Substructural Logical Specifications

Robert J. Simmons

CMU-CS-12-142

November 14, 2012

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA

Thesis Committee

Frank Pfenning, Chair
Robert Harper
André Platzer
Iliano Cervesato, Carnegie Mellon Qatar
Dale Miller, INRIA-Saclay & LIX/Ecole Polytechnique

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Copyright © 2012 Robert J. Simmons

This research was sponsored by the National Science Foundation Graduate Research Fellowship; National Science Foundation under research grant number CNS-0716469; US Army Research Office under grant number DAAD-190210389; Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) via the Carnegie Mellon Portugal Program under grant NGN-44; X10 Innovation Award from IBM; and Siebel Scholar Program.

Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of any sponsoring institution or government.

Keywords: logical frameworks, linear logic, ordered logic, operational semantics

Abstract

A logical framework and its implementation should serve as a flexible tool for specifying, simulating, and reasoning about formal systems. When the formal systems we are interested in exhibit state and concurrency, however, existing logical frameworks fall short of this goal. Logical frameworks based on a rewriting interpretation of substructural logics, ordered and linear logic in particular, can help. To this end, this dissertation introduces and demonstrates four methodologies for developing and using substructural logical frameworks for specifying and reasoning about stateful and concurrent systems.

Structural focalization is a synthesis of ideas from Andreoli's focused sequent calculi and Watkins's hereditary substitution. We can use structural focalization to take a logic and define a restricted form of derivations, the *focused derivations*, that form the basis of a logical framework. We apply this methodology to define SLS, a logical framework for substructural logical specifications, as a fragment of ordered linear lax logic.

Logical correspondence is a methodology for relating and inter-deriving different styles of programming language specification in SLS. The styles we connect range from very high-level specification styles like natural semantics, which do not fully specify the control structure of programs, to low-level specification styles like destination-passing, which provide detailed control over concurrency and control flow. We apply this methodology to systematically synthesize a low-level destination-passing semantics for a Mini-ML language extended with stateful and concurrent primitives. The specification is mostly high-level except for the relatively few rules that actually deal with concurrency.

Linear logical approximation is a methodology for deriving program analyses by performing abstract analysis on the SLS encoding of the language's operational semantics. We demonstrate this methodology by deriving a control flow analysis and an alias analysis from suitable programming language specifications.

Generative invariants are a powerful generalization of both context-free grammars and LF's regular worlds that allow us to express invariants of SLS specifications in SLS. We show that generative invariants can form the basis of progress-and-preservation-style reasoning about programming languages encoded in SLS.

Acknowledgments

I cannot begin to fully acknowledge the debt of gratitude I owe to to my advisor, Frank Pfenning. Frank has provided me with extraordinary guidance from the moment I arrived in Wean Hall and started asking incredibly naive questions about these curious logical propositions $A \multimap B$ and $!A$. Nearly all of the ideas in this document were either developed or critically refined during our many conversations over the past seven years. I also want to thank my committee, Robert Harper, André Platzer, Iliano Cervesato, and Dale Miller. Iliano's extremely timely and detailed feedback during the summer and fall of 2012 was especially important to making this final document as intelligible as it is.

Many, many people have given me helpful feedback on this document and the various papers that went into it, especially Sean McLaughlin, Jason Reed, Rowan Davies, Chris Martens, Ian Zerny, Beniamino Accattoli, Jonathan Aldrich, Roger Wolff, Lindsey Kuper, and Ben Blum. Beyond this group, the conversations I have had with Noam Zeilberger, Dan Licata, Bernardo Toninho, William Lovas, Carsten Schürmann, Henry DeYoung, Anders Schack-Nielsen, David Baelde, Taus Brock-Nannestad, Karl Crary, Carlo Angiuli, and both the Spring 2012 Linear Logic class and the CMU POP group as a whole were incredibly important as I developed the ideas that are presented here. I'd also like to acknowledge the high-quality feedback I've received through anonymous review during my time in graduate school. Not everyone has a positive experience with peer review, and I appreciate the degree to which most of the positive reviews I've received have been both constructive and fair.

And then there's the bigger picture. So much of who I am is due to my family – my extended family of grandparents, aunts, uncles, and cousins, but especially Mom, Dad, and Elizabeth. I would never have gotten to graduate school in the first place, nor made it through, without their unwavering love, support, and patience (so much patience).

I have been fortunate to have the guidance of many exceptional academic and professional mentors along the way. I want to especially thank Jessica Hunt, Andrew Appel, David Walker, Sriram Rajamani, and Aditya Nori, who have each contributed unique and valuable pieces to my mathematical education.

I also must thank the caring, quirky, and diverse Pittsburgh community that has helped me remember that there are things I care about above and beyond writing funny Greek symbols in notebooks and TeX files. Rachel Gougian, of course, who stuck beside me for the whole ride – I couldn't have done this without you, and even if I could have it wouldn't have been much fun. There was also Jen, Jamie, Pete, Laurie, Carsten, Nawshin, Tom, Jim, Mary, Thomas, Emily, Nick, Mike, and all my fellow Dec/5 troublemakers, espresso elves, talented Yinzerstars, SIGBOVIK organizers, and Words with Friends buddies.

It's been a privilege and a blessing. Thanks, everybody.

Contents

1 Introduction **1**
1.1 Logical frameworks . . . 3
1.2 Substructural operational semantics . . . 6
1.3 Invariants in substructural logic . . . 9
1.4 Contributions . . . 11

I Focusing substructural logics **13**

2 Linear logic **15**
2.1 Introduction to linear logic . . . 15
2.2 Logical frameworks . . . 17
2.3 Focused linear logic . . . 19
2.3.1 Polarity . . . 20
2.3.2 Polarization . . . 21
2.3.3 Focused sequent calculus . . . 23
2.3.4 Suspended propositions . . . 25
2.3.5 Identity expansion . . . 27
2.3.6 Cut admissibility . . . 29
2.3.7 Correctness of focusing . . . 32
2.3.8 Confluent versus fixed inversion . . . 35
2.3.9 Running example . . . 36
2.4 Synthetic inference rules . . . 37
2.5 Hacking the focusing system . . . 39
2.5.1 Atom optimization . . . 41
2.5.2 Exponential optimization . . . 45
2.5.3 Adjoint logic . . . 47
2.5.4 Permeability . . . 49
2.6 Revisiting our notation . . . 53

3 Substructural logic **55**
3.1 Ordered linear lax logic . . . 56
3.1.1 First-order logic . . . 59
3.2 Substructural contexts . . . 60

3.2.1 Fundamental operations on contexts . . . 61
3.2.2 Multiplicative operations . . . 63
3.2.3 Exponential operations . . . 64
3.3 Focused sequent calculus . . . 65
3.3.1 Restrictions on the form of sequents . . . 65
3.3.2 Polarized propositions . . . 66
3.3.3 Derivations and proof terms . . . 67
3.3.4 Variable substitution . . . 70
3.3.5 Focal substitution . . . 72
3.4 Cut admissibility . . . 73
3.4.1 Optimizing the statement of cut admissibility . . . 74
3.4.2 Proof of cut admissibility, Theorem 3.6 . . . 75
3.5 Identity expansion . . . 86
3.6 Correctness of focusing . . . 88
3.6.1 Erasure . . . 88
3.6.2 De-focalization . . . 89
3.6.3 Unfocused admissibility . . . 89
3.6.4 Focalization . . . 92
3.7 Properties of syntactic fragments . . . 92
3.8 The design space of proof terms . . . 93

4 Substructural logical specifications 97
4.1 Spine Form LF as a term language . . . 98
4.1.1 Core syntax . . . 99
4.1.2 Simple types and hereditary substitution . . . 99
4.1.3 Judgments . . . 100
4.1.4 Adequacy . . . 103
4.2 The logical framework SLS . . . 104
4.2.1 Propositions . . . 105
4.2.2 Substructural contexts . . . 107
4.2.3 Process states . . . 109
4.2.4 Patterns . . . 110
4.2.5 Values, terms, and spines . . . 111
4.2.6 Steps and traces . . . 114
4.2.7 Presenting traces . . . 115
4.2.8 Frame properties . . . 116
4.3 Concurrent equality . . . 116
4.3.1 Multifocusing . . . 119
4.4 Adequate encoding . . . 120
4.4.1 Generative signatures . . . 121
4.4.2 Restriction . . . 122
4.4.3 Generative invariants . . . 126
4.4.4 Adequacy of the transition system . . . 130
4.5 The SLS implementation . . . 131

4.6 Logic programming . . . 132
4.6.1 Deductive computation and backward chaining . . . 133
4.6.2 Concurrent computation . . . 135
4.6.3 Integrating deductive and trace computation . . . 136
4.7 Design decisions . . . 137
4.7.1 Pseudo-positive atoms . . . 137
4.7.2 The need for traces . . . 138
4.7.3 LF as a term language . . . 139

II Substructural operational semantics 141

5 On logical correspondence 143
5.1 Logical correspondence . . . 144
5.2 Related work . . . 147
5.3 Transformation and modular extension . . . 151

6 Ordered abstract machines 153
6.1 Logical transformation: operationalization . . . 154
6.1.1 Transformable signatures . . . 157
6.1.2 Basic transformation . . . 158
6.1.3 Tail-recursion . . . 159
6.1.4 Parallelism . . . 161
6.1.5 Correctness . . . 163
6.2 Logical transformation: defunctionalization . . . 169
6.2.1 Defunctionalization . . . 170
6.2.2 Uncurrying . . . 172
6.2.3 From many predicates to many frames . . . 172
6.3 Adequacy with abstract machines . . . 173
6.3.1 Encoding states . . . 174
6.3.2 Preservation and adequacy . . . 175
6.4 Exploring the image of operationalization . . . 176
6.4.1 Arbitrary choice and failure . . . 177
6.4.2 Conditionals and factoring . . . 178
6.4.3 Operationalization and computation . . . 180
6.5 Exploring the richer fragment . . . 181
6.5.1 Mutable storage . . . 182
6.5.2 Call-by-need evaluation . . . 185
6.5.3 Environment semantics . . . 188
6.5.4 Recoverable failure . . . 189
6.5.5 Looking back at natural semantics . . . 191
6.6 Other applications of transformation . . . 192
6.6.1 Binary addition . . . 193
6.6.2 Operationalizing SOS specifications . . . 194

6.6.3 Partial evaluation in $\lambda^{\bigcirc}$ . . . 196

7 Destination-passing **199**
7.1 Logical transformation: destination-adding . . . 200
7.1.1 Vestigial destinations . . . 204
7.1.2 Persistent destination passing . . . 204
7.2 Exploring the richer fragment . . . 205
7.2.1 Alternative semantics for parallelism and failure . . . 206
7.2.2 Synchronization . . . 207
7.2.3 Futures . . . 209
7.2.4 First-class continuations . . . 211

8 Linear logical approximation **213**
8.1 Saturating logic programming . . . 213
8.2 Using approximation . . . 215
8.3 Logical transformation: approximation . . . 217
8.4 Control flow analysis . . . 219
8.4.1 Subexpressions in higher-order abstract syntax . . . 220
8.4.2 Environment semantics . . . 221
8.4.3 Approximation to 0CFA . . . 221
8.4.4 Correctness . . . 224
8.5 Alias analysis . . . 226
8.5.1 Monadic language . . . 226
8.5.2 Approximation and alias analysis . . . 228
8.6 Related work . . . 230

III Reasoning about substructural logical specifications **233**

9 Generative invariants **235**
9.1 Worlds . . . 237
9.1.1 Regular worlds . . . 238
9.1.2 Regular worlds from generative signatures . . . 239
9.1.3 Regular worlds in substructural specifications . . . 239
9.1.4 Generative versus consumptive signatures . . . 239
9.2 Invariants of ordered specifications . . . 240
9.2.1 Inversion . . . 242
9.2.2 Preservation . . . 246
9.3 From well-formed to well-typed states . . . 248
9.3.1 Inversion . . . 250
9.3.2 Preservation . . . 250
9.4 State . . . 252
9.4.1 Inversion . . . 253
9.4.2 Uniqueness . . . 255

9.4.3 Preservation . . . 256
9.4.4 Revisiting pointer inequality . . . 258
9.5 Destination-passing . . . 259
9.5.1 Uniqueness and index sets . . . 260
9.5.2 Inversion . . . 262
9.5.3 Preservation . . . 263
9.5.4 Extensions . . . 264
9.6 Persistent continuations . . . 264
9.7 On mechanization . . . 267

10 Safety for substructural specifications 269
10.1 Backwards and forwards through traces . . . 269
10.2 Progress for ordered abstract machines . . . 270
10.3 Progress with mutable storage . . . 272
10.4 Safety . . . 273

11 Conclusion 275

A Process states summary 277
A.1 Substructural contexts . . . 277
A.2 Steps and traces . . . 278

B A hybrid specification of Mini-ML 279
B.1 Pure Mini-ML . . . 279
B.1.1 Syntax . . . 280
B.1.2 Natural semantics . . . 280
B.2 State . . . 281
B.2.1 Syntax . . . 281
B.2.2 Nested ordered abstract machine semantics . . . 282
B.2.3 Flat ordered abstract machine semantics . . . 282
B.3 Failure . . . 283
B.3.1 Syntax . . . 283
B.3.2 Flat ordered abstract machine semantics . . . 283
B.4 Parallelism . . . 284
B.4.1 Destination-passing semantics . . . 284
B.4.2 Integration of parallelism and exceptions . . . 284
B.5 Concurrency . . . 285
B.5.1 Syntax . . . 285
B.5.2 Natural semantics . . . 285
B.5.3 Destination-passing semantics . . . 286
B.6 Composing the semantics . . . 287

Bibliography 289

List of Figures

1.1 Series of PDA transitions . . . 3
1.2 A Boolean program, encoded as a rewriting system and in SLS . . . 5
1.3 SSOS evaluation of an expression to a value . . . 7
1.4 Evaluation with an imperative counter . . . 8
1.5 Proving well-formedness of one of the states from Figure 1.3 . . . 10

2.1 Intuitionstic linear logic . . . 16
2.2 Proving that a transition is possible (where we let $\Gamma = \mathsf{6bucks} \multimap \mathsf{battery}$) . . . 18
2.3 De-polarizing and polarizing (with minimal shifts) propositions of MELL . . . 21
2.4 Fully-shifting polarization strategy for MELL . . . 22
2.5 Focused intuitionstic linear logic . . . 24
2.6 Identity expansion – restricting η^+ and η^- to atomic propositions . . . 28
2.7 Identity expansion for units and additive connectives . . . 29
2.8 Lifting erasure and polarization (Figure 2.3) to contexts and succeedents . . . 32
2.9 Proving that a focused transition is possible (where we let $\Gamma = \mathsf{6bucks} \multimap \uparrow\mathsf{battery}$) 37
2.10 Our running example, presented with synthetic rules . . . 39
2.11 Substituting A^+ for p^+ in the presence of the atom optimization . . . 43
2.12 A problematic cut that arises from the introduction of the $\Uparrow p^+$ connective . . . 44
2.13 Some relevant sequent calculus rules for adjoint logic . . . 48
2.14 Persistent identity expansion . . . 51
2.15 Alternative presentation of intuitionstic linear logic . . . 54

3.1 Propositional ordered linear lax logic . . . 58
3.2 First-order ordered linear lax logic . . . 59
3.3 Summary of where propositions and judgments appear in OL_3 sequents . . . 65
3.4 Propositions of polarized OL_3 . . . 67
3.5 Multiplicative, exponential fragment of focused OL_3 (contexts Ψ suppressed) . . . 68
3.6 Additive connectives of focused OL_3 (contexts Ψ suppressed) . . . 69
3.7 First-order connectives of focused OL_3 . . . 69
3.8 Erasure in OL_3 . . . 89
3.9 Unfocused admissibility for the multiplicative, exponential fragment of OL_3 . . . 90
3.10 Unfocused admissibility for the additive connectives of OL_3 (omits $\oplus_{R2}$, $\&_{L2}$) . . . 91
3.11 Unfocused admissibility for the first-order connectives of OL_3 . . . 91

4.1 Hereditary substitution on terms, spines, and classifiers . . . 100

4.2 Simultaneous substitution on terms, spines, and classifiers . . . 101
4.3 LF formation judgments ($\tau' = p$ refers to α-equivalence) . . . 102
4.4 Equality constraints (used to support notational definitions) . . . 105
4.5 SLS propositions . . . 106
4.6 SLS contexts . . . 108
4.7 SLS patterns . . . 111
4.8 SLS values . . . 112
4.9 SLS atomic terms . . . 112
4.10 SLS terms . . . 112
4.11 SLS spines . . . 113
4.12 SLS steps . . . 114
4.13 SLS traces . . . 114
4.14 Interaction diagram for a trace $(u_1{:}\langle \mathsf{a} \rangle, u_2{:}\langle \mathsf{a} \rangle, u_3{:}\langle \mathsf{a} \rangle) \leadsto^*_\Sigma (z_1{:}\langle \mathsf{f} \rangle, z_2{:}\langle \mathsf{f} \rangle, z_3{:}\langle \mathsf{f} \rangle)$. 120
4.15 Generative signature for PDA states and an analogous context-free grammar . . . 122
4.16 Mathematical and ASCII representations of propositions, terms, and classifiers . 131
4.17 A rough taxonomy of deductive computation . . . 135

5.1 Major transformations on SLS specifications . . . 145
5.2 Evolution of a nested SLS process state . . . 146
5.3 Classification of existing work on SSOS specifications . . . 147
5.4 Using the logical correspondence for modular language extension . . . 151

6.1 Natural semantics (left) and ordered abstract machine (right) for CBV evaluation 157
6.2 Tail-recursion optimized semantics for CBV evaluation . . . 161
6.3 Parallel, tail-recursion optimized semantics for CBV evaluation . . . 162
6.4 Defunctionalization on a nested SLS signature . . . 170
6.5 Uncurried call-by-value evaluation . . . 172
6.6 A first-order ordered abstract machine semantics for CBV evaluation . . . 173
6.7 The generative signature Σ_{Gen} describing states Δ that equal $\ulcorner s \urcorner$ for some s . . 174
6.8 Semantics of some pure functional features . . . 176
6.9 Semantics of nondeterministic choice . . . 177
6.10 Problematic semantics of case analysis (not defunctionalized) . . . 179
6.11 Problematic semantics of case analysis (defunctionalized) . . . 179
6.12 Revised semantics of case analysis (not defunctionalized) . . . 180
6.13 Revised semantics of case analysis (defunctionalized) . . . 180
6.14 Semantics of mutable storage . . . 182
6.15 A racy process state . . . 184
6.16 Semantics of call-by-need recursive suspensions . . . 186
6.17 Semantics of call-by-need recursive suspensions, refunctionalized . . . 187
6.18 Semantics of lazy call-by-need functions . . . 187
6.19 Environment semantics for call-by-value functions . . . 188
6.20 Semantics of recoverable failure . . . 189
6.21 Backward-chaining logic program for binary addition . . . 192
6.22 Forward-chaining logic program for binary addition . . . 193

6.23 SOS evaluation 194
6.24 The operationalization of evsos from Figure 6.23 195
6.25 This transformation of Figure 6.23 evokes an evaluation context semantics 196
6.26 Semantics of partial evaluation for $\lambda^{\bigcirc}$ (lambda calculus fragment) 197
6.27 Semantics for $\lambda^{\bigcirc}$ (temporal fragment) 198

7.1 Ordered SLS specification of a PDA for parenthesis matching 199
7.2 Linear SLS specification of a PDA for parenthesis matching 200
7.3 Destination-adding transformation 201
7.4 Translation of Figure 6.6 with vestigial destinations 203
7.5 Translation of Figure 6.6 without vestigial destinations 204
7.6 Destination-passing semantics for parallel evaluation of pairs 206
7.7 Integration of parallelism and exceptions; signals failure as soon as possible . . . 207
7.8 Semantics of simple synchronization 208
7.9 Semantics of call-by-future functions 209
7.10 Series of process states in an example call-by-future evaluation 210
7.11 Semantics of first-class continuations (with letcc) 211

8.1 Skolemized approximate version of the PDA specification from Figure 7.2 215
8.2 Approximated PDA specification 217
8.3 Alternative environment semantics for CBV evaluation 221
8.4 A control-flow analysis derived from Figure 8.3 222
8.5 Simplification of Figure 8.4 that eliminates the vestigial argument to eval 223
8.6 Coinductive definition of an acceptable control flow analysis 225
8.7 Semantics of functions in the simple monadic language 227
8.8 Semantics of mutable pairs in the simple monadic language 228
8.9 Alias analysis for the simple monadic language 229

9.1 Ordered abstract machine with parallel evaluation and failure 241
9.2 Generative invariant: well-formed process states 243
9.3 Graphical representation of part 1 of the inversion lemma for $\Sigma_{Gen9.2}$ 244
9.4 Generative invariant: well-typed process states 249
9.5 Back-patching, with judgments (ord and eph) and arguments corresponding to implicit quantifiers elided 254
9.6 Generative invariant: well-typed mutable storage 255
9.7 Generative invariants for cells with unique natural-number tags 258
9.8 Generative invariant: destination-passing ("obvious" formulation) 259
9.9 Generative invariant: destination-passing (modified formulation) 260
9.10 Generative invariant: futures 264
9.11 Generative invariant: persistent destinations and first-class continuations 265

Chapter 1

Introduction

Suppose you find yourself in possession of

* a calculator of unfamiliar design, or
* a new board game, or
* the control system for an army of robots, or
* an implementation of a security protocol, or
* the interface to a high-frequency trading system.

The fundamental questions are the same: *What does it do? What are the rules of the game?* The answer to this question, whether it comes in the form of an instruction manual, a legal document, or an ISO standard, is a *specification*.

Specifications must be *formal*, because any room for misinterpretation could (respectively) lead to incorrect calculations, accusations of cheating, a robot uprising, a security breach, or bankruptcy. At the same time, specifications must be *clear*: while clarity is in the eye of the beholder, a specification that one finds hopelessly confusing or complex is no more useful than one that is hopelessly vague. Clarity is what allows us to communicate with each other, to use specifications to gain a common understanding of what some system does and to think about how that system might be changed. Formality is what allows specifications to interact with the world of computers, to say with confidence that the *implementation* of the calculator or high-frequency trading system obeys the specification. Formality also allows specifications to interact with the world of mathematics, and this, in turn, enables us to make precise and accurate statements about what may or may not happen to a given system.

The specification of many (too many!) critical systems still remains in the realm of English text, and the inevitable lack of formality can and does make formal reasoning about these specifications difficult or impossible. Notably, this is true about most of the programming languages used to implement our calculators, program our robot army control systems, enforce our security protocols, and interact with our high-frequency trading systems. In the last decade, however, we have finally begun to see the emergence of operational semantics specifications (the "rules of the game" for a programming language) for real-world programming languages that are truly formal. A notable aspect of this recent work is that the formalization effort is not done simply for formalization's sake. Ellison and Roşu's formal semantics of C can be used to check individual

programs for undefined behavior, unsafe situations where the rules of the game no longer apply and the compiler is free to do anything, including unleashing the robot army [ER12]. Lee, Crary, and Harper's formalization of Standard ML has been used to formally prove – using a computer to check all the proof's formal details – a much stronger safety property: that *every* program accepted by the compiler is free of undefined behavior [LCH07].

Mathematics, by contrast, has a century-long tradition of insisting on absolute formality (at least in principle: practice often falls far short). Over time, this tradition has become a collaboration between practicing mathematicians and practicing computer scientists, because while humans are reasonable judges of clarity, computers have absolutely superhuman patience when it comes to checking all the formal details of an argument. One aspect of this collaboration has been the development of *logical frameworks*. In a logical framework, the language of specifications is derived from the language of logic, which gives specifications in a logical framework an independent meaning based on the logic from which the logical framework was derived. To be clear, the language of logic is not a single, unified entity: logics are formal systems that satisfy certain internal coherence properties, and we study many of them. For example, the logical framework Coq is based on the Calculus of Inductive Constructions [Coq10], the logical framework Agda is based on a variant of Martin-Löf's type theory called UTT_{Σ} [Nor07], and the logical framework Twelf is based on the dependent type theory λ^{Π}, also known as LF [PS99b]. Twelf was the basis of Lee, Crary, and Harper's formalization of Standard ML.

Why is there not a larger tradition of formally specifying the programming languages that people actually use? Part of the answer is that most languages that people actually use have lots of features – like mutable state, or exception handling, or synchronization and communication, or lazy evaluation – that are not particularly pleasant to specify using existing logical frameworks. Dealing with a few unpleasant features at a time might not be much trouble, but the combinations that appear in actual programming languages cause formal programming language specifications to be both unclear for humans to read and inconvenient for formal tools to manipulate. A more precise statement is that the addition of the aforementioned features is *non-modular*, because handling a new feature requires reconsidering and revising the rest of the specification. Some headway on this problem has been made by frameworks like the K semantic framework that are formal but not logically derived; the K semantic framework is based on a formal system of rewriting rules [RŞ10]. Ellison and Roşu's formalization of C was done in the K semantic framework.

This dissertation considers the specification of systems, particularly programming languages, in logical frameworks. We consider a particular family of logics, called *substructural logics*, in which logical propositions can be given an interpretation as rewriting rules as detailed by Cervesato and Scedrov [CS09]. We seek to support the following:

> **Thesis Statement:** *Logical frameworks based on a rewriting interpretation of substructural logics are suitable for modular specification of programming languages and formal reasoning about their properties.*[1]

Part I of the dissertation covers the design of logical frameworks that support this rewriting interpretation and the design of the logical framework SLS in particular. Part II considers the

[1]The original thesis proposal used the phrase "forward reasoning in substructural logics" instead of the phrase "a rewriting interpretation of substructural logics," but these are synonymous, as discussed in Section 4.6.

$$
\begin{array}{lllllllllll}
\mathsf{hd} & < & < & > & < & < & > & > & > & \rightsquigarrow \\
< & \mathsf{hd} & < & > & < & < & > & > & > & \rightsquigarrow \\
< & < & \mathsf{hd} & > & < & < & > & > & > & \rightsquigarrow \\
 & & < & \mathsf{hd} & < & < & > & > & > & \rightsquigarrow \\
 & & < & < & \mathsf{hd} & < & > & > & > & \rightsquigarrow \\
 & & < & < & < & \mathsf{hd} & > & > & > & \rightsquigarrow \\
 & & & & < & < & \mathsf{hd} & > & > & \rightsquigarrow \\
 & & & & & & < & \mathsf{hd} & > & \rightsquigarrow \\
 & & & & & & & & \mathsf{hd} &
\end{array}
$$

Figure 1.1: Series of PDA transitions

modular specification of programming language features in SLS and the methodology by which we organize and relate styles of specification. Part III discusses formal reasoning about properties of SLS specifications, with an emphasis on establishing invariants.

1.1 Logical frameworks

Many interesting stateful systems have a natural notion of *ordering* that is fundamental to their behavior. A very simple example is a push-down automaton (PDA) that reads a string of symbols left-to-right while maintaining and manipulating a separate stack of symbols. We can represent a PDA's internal configuration as a sequence with three regions:

[the stack] [the head] [the string being read]

where the symbols closest to the head are the top of the stack and the symbol waiting to be read from the string. If we represent the head as a token hd, we can describe the behavior (the rules of the game) for the PDA that checks a string for correct nesting of angle braces by using two rewriting rules:

$$\mathsf{hd} < \;\rightsquigarrow\; < \mathsf{hd} \qquad \text{(push)}$$

$$< \mathsf{hd} > \;\rightsquigarrow\; \mathsf{hd} \qquad \text{(pop)}$$

The distinguishing feature of these rewriting rules is that they are *local* – they do not mention the entire stack or the entire string, just the relevant fragment at the beginning of the string and the top of the stack. Execution of the PDA on a particular string of tokens then consists of (1) appending the token hd to the beginning of the string, (2) repeatedly performing rewritings until no more rewrites are possible, and (3) checking to see if only a single token hd remains. One possible series of transitions that this rewriting system can take is shown in Figure 1.1

Because our goal is to use a framework that is both simple and logically motivated, we turn to a substructural logic called *ordered logic*, a fragment of which was originally proposed by

Lambek for applications in computational linguistics [Lam58]. In ordered logic, hypotheses are ordered relative to one another and cannot be rearranged. The rewriting rules we considered above can be expressed as propositions in ordered logic, where the tokens hd, $>$, and $<$ are all treated as *atomic propositions*:

$$\begin{aligned} \mathsf{push}: &\quad \mathsf{hd} \bullet < \;\rightarrowtail\; \{< \bullet \mathsf{hd}\} \\ \mathsf{pop}: &\quad < \bullet \mathsf{hd} \bullet > \;\rightarrowtail\; \{\mathsf{hd}\} \end{aligned}$$

The symbol $\bullet$ (pronounced "fuse") is the binary connective for ordered conjunction (i.e. concatenation); it binds more tightly than $\rightarrowtail$, a binary connective for ordered implication. The curly braces $\{\ldots\}$ can be ignored for now.

The propositional fragment of ordered logic is Turing complete: it is in fact a simple exercise to specify a Turing machine! Nevertheless, first-order quantification helps us write specifications that are short and clear. For example, by using first-order quantification we can describe a a more general push-down automaton in a generic way. In this generic specification, we use $\mathsf{left}(X)$ and $\mathsf{right}(X)$ to describe left and right angle braces ($X = \mathsf{an}$), square braces ($X = \mathsf{sq}$), and parentheses ($X = \mathsf{pa}$). The string $[\ <\ >\ (\ [\ ]\ )\ ]$ is then represented by the following sequence of ordered atomic propositions:

$$\mathsf{left}(\mathsf{sq})\ \ \mathsf{left}(\mathsf{an})\ \ \mathsf{right}(\mathsf{an})\ \ \mathsf{left}(\mathsf{pa})\ \ \mathsf{left}(\mathsf{sq})\ \ \mathsf{right}(\mathsf{sq})\ \ \mathsf{right}(\mathsf{pa})\ \ \mathsf{right}(\mathsf{sq})$$

The following rules describe the more general push-down automaton:

$$\begin{aligned} \mathsf{push}: &\quad \forall x.\, \mathsf{hd} \bullet \mathsf{left}(x) \;\rightarrowtail\; \{\mathsf{stack}(x) \bullet \mathsf{hd}\} \\ \mathsf{pop}: &\quad \forall x.\, \mathsf{stack}(x) \bullet \mathsf{hd} \bullet \mathsf{right}(x) \;\rightarrowtail\; \{\mathsf{hd}\} \end{aligned}$$

(This specification would still be possible in propositional ordered logic; we would just need one copy of the push rule and one copy of the pop rule for each pair of braces.) Note that while we use the fuse connective to indicate adjacent tokens in the rules above, no fuses appear in Figure 1.1. That is because the intermediate states are not propositions in the same way rules are propositions. Rather, the intermediate states in Figure 1.1 are *contexts* in ordered logic, which we will refer to as *process states*.

The most distinctive characteristic of these transition systems is that the intermediate stages of computation are encoded in the structure of a substructural context (a process state). This general idea dates back to Miller [Mil93] and his Ph.D. student Chirimar [Chi95], who encoded the intermediate states of a π-calculus and of a low-level RISC machine (respectively) as contexts in focused classical linear logic. Part I of this dissertation is concerned with the design of logical frameworks for specifying transition systems. In this respect, Part I follows in the footsteps of Miller's Forum [Mil96], Cervesato and Scedrov's multiset rewriting language ω [CS09], and Watkins et al.'s CLF [WCPW02].

As an extension to CLF, the logical framework we develop is able to specify systems like the π-calculus, security protocols, and Petri nets that can be encoded in CLF [CPWW02]. The addition of ordered logic allows us to easily incorporate specifications that are naturally expressed as string rewriting systems. An example from the verification domain, taken from Bouajjani and Esparza [BE06], is shown in Figure 1.2. The left-hand side of the figure is a simple Boolean

bool function $foo(l)$ f_0: if l then f_1: return ff else f_2: return tt fi	$\langle b\rangle\,\langle \mathsf{tt}, \mathsf{f}_0\rangle \;\rightarrow\; \langle b\rangle\,\langle \mathsf{tt}, \mathsf{f}_1\rangle$ $\langle b\rangle\,\langle \mathsf{ff}, \mathsf{f}_0\rangle \;\rightarrow\; \langle b\rangle\,\langle \mathsf{ff}, \mathsf{f}_2\rangle$ $\langle b\rangle\,\langle l, \mathsf{f}_1\rangle \;\rightarrow\; \langle \mathsf{ff}\rangle$ $\langle b\rangle\,\langle l, \mathsf{f}_2\rangle \;\rightarrow\; \langle \mathsf{tt}\rangle$	$\forall b.\, \mathsf{gl}(b) \bullet \mathsf{foo}(\mathsf{tt}, \mathsf{f}_0) \rightarrowtail \{\mathsf{gl}(b) \bullet \mathsf{foo}(\mathsf{tt}, \mathsf{f}_1)\}$ $\forall b.\, \mathsf{gl}(b) \bullet \mathsf{foo}(\mathsf{ff}, \mathsf{f}_0) \rightarrowtail \{\mathsf{gl}(b) \bullet \mathsf{foo}(\mathsf{ff}, \mathsf{f}_1)\}$ $\forall b.\, \mathsf{gl}(b) \bullet \mathsf{foo}(l, \mathsf{f}_1) \rightarrowtail \{\mathsf{gl}(\mathsf{ff})\}$ $\forall b.\, \mathsf{gl}(b) \bullet \mathsf{foo}(l, \mathsf{f}_2) \rightarrowtail \{\mathsf{gl}(\mathsf{tt})\}$
procedure $main()$ global b m_0: while b do m_1: $b := foo(b)$ od m_2: return	$\langle \mathsf{tt}\rangle\,\langle \mathsf{m}_0\rangle \;\rightarrow\; \langle \mathsf{tt}\rangle\,\langle \mathsf{m}_1\rangle$ $\langle \mathsf{ff}\rangle\,\langle \mathsf{m}_0\rangle \;\rightarrow\; \langle \mathsf{ff}\rangle\,\langle \mathsf{m}_2\rangle$ $\langle b\rangle\,\langle \mathsf{m}_1\rangle \;\rightarrow\; \langle b\rangle\,\langle b, \mathsf{f}_0\rangle\,\langle \mathsf{m}_0\rangle$ $\langle b\rangle\,\langle \mathsf{m}_2\rangle \;\rightarrow\; \epsilon$	$\mathsf{gl}(\mathsf{tt}) \bullet \mathsf{main}(\mathsf{m}_0) \rightarrowtail \{\mathsf{gl}(\mathsf{tt}) \bullet \mathsf{main}(\mathsf{m}_1)\}$ $\mathsf{gl}(\mathsf{ff}) \bullet \mathsf{main}(\mathsf{m}_0) \rightarrowtail \{\mathsf{gl}(\mathsf{tt}) \bullet \mathsf{main}(\mathsf{m}_2)\}$ $\forall b.\, \mathsf{gl}(b) \bullet \mathsf{main}(\mathsf{m}_1) \rightarrowtail \{\mathsf{gl}(b) \bullet \mathsf{foo}(b, \mathsf{f}_0) \bullet \mathsf{main}(\mathsf{m}_0)\}$ $\forall b.\, \mathsf{gl}(b) \bullet \mathsf{main}(\mathsf{m}_2) \rightarrowtail \{\mathbf{1}\}$

Figure 1.2: A Boolean program, encoded as a rewriting system and in SLS

program: the procedure foo has one local variable and the procedure $main$ has no local variables but mentions a global variable b. Bouajjani and Esparza represented Boolean programs like this one as *canonical systems* like the one shown in the middle of Figure 1.2. Canonical systems are rewriting systems where only the left-most tokens are ever rewritten: the left-most token in this canonical system always has the form $\langle b\rangle$, where b is either true (tt) or false (ff), representing the valuation of the global variables – there is only one, b. The token to the right of the global variables contains the current program counter and the value of the current local variables. The token to the right of *that* contains the program counter and local variables of the calling procedure, and so on, forming a call stack that grows off to the right (in contrast to the PDA's stack, which grew off to the left). Canonical systems can be directly represented in ordered logic, as shown on the right-hand side of Figure 1.2. The atomic proposition $\mathsf{gl}(b)$ contains the global variables (versus $\langle b\rangle$ in the middle column), the atomic proposition $\mathsf{foo}(l, f)$ contains the local variables and program counter within the procedure foo (versus $\langle l, f\rangle$ in the middle column), and the atomic proposition $\mathsf{main}(m)$ contains the program counter within the procedure main (versus $\langle m\rangle$ in the middle column).

The development of SLS, a CLF-like framework of **s**ubstructural **l**ogical **s**pecifications that includes an intrinsic notion of order, is a significant development of Part I of the dissertation. However, the principal contribution of these three chapters is the development of *structural focalization*, which unifies Andreoli's work on focused logics [And92] with the *hereditary substitution* technique that Watkins developed in the context of CLF [WCPW02]. Chapter 2 explains structural focalization in the context of linear logic, Chapter 3 establishes focalization for a richer substructural logic OL_3, and Chapter 4 takes focused OL_3 and carves out the SLS framework as a fragment of the focused logic.

1.2 Substructural operational semantics

Existing logical frameworks are perfectly capable of representing simple systems like PDAs, and while applications in the verification domain like the rewriting semantics of Boolean programs are an interesting application of SLS, they will not be a focus of this dissertation. Instead, in Part II, we will concentrate on specifying the operational semantics of programming languages in SLS. We can represent operational semantics in SLS in many ways, but we are particularly interested in a broad specification style called *substructural operational semantics*, or SSOS [Pfe04, PS09].[2] SSOS is a synthesis of structural operational semantics, abstract machines, and logical specifications.

One of our running examples will be a call-by-value operational semantics for the untyped lambda calculus, defined by the BNF grammar:

$$e ::= x \mid \lambda x.e \mid e_1\, e_2$$

Taking some liberties with our representation of terms,[3] we can describe call-by-value evaluation for this language with the same rewriting rules we used to describe the PDA and the Boolean program's semantics. Our specification uses three atomic propositions: one, $\mathsf{eval}(e)$, carries an unevaluated expression e, and another, $\mathsf{retn}(v)$, carries an evaluated value v. The third atomic proposition, $\mathsf{cont}(f)$, contains a *continuation frame* f that represents some partially evaluated value: $f = \Box\, e_2$ contains an expression e_2 waiting on the evaluation of e_1 to a value, and $f = (\lambda x.e)\, \Box$ contains an function $\lambda x.e$ waiting on the evaluation of e_2 to a value. These frames are arranged in a stack that grows off to the right (like the Boolean program's stack).

The evaluation of a function is simple, as a function is already a fully evaluated value, so we replace $\mathsf{eval}(\lambda x.e)$ in-place with $\mathsf{retn}(\lambda x.e)$:

$$\mathsf{ev/lam}: \quad \mathsf{eval}\,(\lambda x.e) \rightarrowtail \{\mathsf{retn}\,(\lambda x.e)\}$$

The evaluation of an application $e_1\, e_2$, on the other hand, requires us to push a new element onto the stack. We evaluate $e_1\, e_2$ by evaluating e_1 and leaving behind a frame $\Box\, e_2$ that suspends the argument e_2 while e_1 is being evaluated to a value.

$$\mathsf{ev/app}: \quad \mathsf{eval}\,(e_1\, e_2) \rightarrowtail \{\mathsf{eval}\,(e_1) \bullet \mathsf{cont}\,(\Box\, e_2)\}$$

When a function is returned to a waiting $\Box\, e_2$ frame, we switch to evaluating the function argument while storing the returned function in a frame $(\lambda x.e)\, \Box$.

$$\mathsf{ev/app1}: \quad \mathsf{retn}\,(\lambda x.e) \bullet \mathsf{cont}\,(\Box\, e_2) \rightarrowtail \{\mathsf{eval}\,(e_2) \bullet \mathsf{cont}\,((\lambda x.e)\, \Box)\}$$

Finally, when an evaluated function argument is returned to the waiting $(\lambda x.e)\, \Box$ frame, we substitute the value into the body of the function and evaluate the result.

$$\mathsf{ev/app2}: \quad \mathsf{retn}\,(v_2) \bullet \mathsf{cont}\,((\lambda x.e)\, \Box) \rightarrowtail \{\mathsf{eval}\,([v_2/x]e)\}$$

$$
\begin{array}{rrrcl}
 & & \mathsf{eval}\,((\lambda x.x)\,((\lambda y.y)\,(\lambda z.e))) & \rightsquigarrow & \text{(by rule ev/app)} \\
 & \mathsf{eval}\,(\lambda x.x) & \mathsf{cont}\,(\square\,((\lambda y.y)\,(\lambda z.e))) & \rightsquigarrow & \text{(by rule ev/lam)} \\
 & \mathsf{retn}\,(\lambda x.x) & \mathsf{cont}\,(\square\,((\lambda y.y)\,(\lambda z.e))) & \rightsquigarrow & \text{(by rule ev/app1)} \\
 & \mathsf{eval}\,((\lambda y.y)\,(\lambda z.e)) & \mathsf{cont}\,((\lambda x.x)\,\square) & \rightsquigarrow & \text{(by rule ev/app)} \\
\mathsf{eval}\,(\lambda y.y) & \mathsf{cont}\,(\square\,(\lambda z.e)) & \mathsf{cont}\,((\lambda x.x)\,\square) & \rightsquigarrow & \text{(by rule ev/lam)} \\
\mathsf{retn}\,(\lambda y.y) & \mathsf{cont}\,(\square\,(\lambda z.e)) & \mathsf{cont}\,((\lambda x.x)\,\square) & \rightsquigarrow & \text{(by rule ev/app1)} \\
\mathsf{eval}\,(\lambda z.e) & \mathsf{cont}\,((\lambda y.y)\,\square) & \mathsf{cont}\,((\lambda x.x)\,\square) & \rightsquigarrow & \text{(by rule ev/lam)} \\
\mathsf{retn}\,(\lambda z.e) & \mathsf{cont}\,((\lambda y.y)\,\square) & \mathsf{cont}\,((\lambda x.x)\,\square) & \rightsquigarrow & \text{(by rule ev/app2)} \\
 & \mathsf{eval}\,(\lambda z.e) & \mathsf{cont}\,((\lambda x.x)\,\square) & \rightsquigarrow & \text{(by rule ev/lam)} \\
 & \mathsf{retn}\,(\lambda z.e) & \mathsf{cont}\,((\lambda x.x)\,\square) & \rightsquigarrow & \text{(by rule ev/app2)} \\
 & & \mathsf{eval}\,(\lambda z.e) & \rightsquigarrow & \text{(by rule ev/lam)} \\
 & & \mathsf{retn}\,(\lambda z.e) & \not\rightsquigarrow &
\end{array}
$$

Figure 1.3: SSOS evaluation of an expression to a value

These four rules constitute an SSOS specification of call-by-value evaluation; an example of evaluating the expression $(\lambda x.x)\,((\lambda y.y)\,(\lambda z.e))$ to a value under this specification is given in Figure 1.3. Again, each intermediate state is represented by a process state or ordered context.

The SLS framework admits many styles of specification. The SSOS specification above resides in the *concurrent* fragment of SLS. (This rewriting-like fragment is called concurrent because rewriting specifications are naturally concurrent – we can just as easily seed the process state with two propositions $\mathsf{eval}(e)$ and $\mathsf{eval}(e')$ that will evaluate to values concurrently and independently, side-by-side in the process state.) Specifications in the concurrent fragment of SLS can take many different forms, a point that we will discuss further in Chapter 5.

On the other end of the spectrum, the *deductive* fragment of SLS supports the specification of inductive definitions by the same methodology used to represent inductive definitions in LF [HHP93]. We can therefore use the deductive fragment of SLS to specify a big-step operational semantics for call-by-value evaluation by inductively defining the judgment $e \Downarrow v$, which expresses that the expression e evaluates to the value v. On paper, this big-step operational semantics is expressed with two inference rules:

$$
\frac{}{\lambda x.e \Downarrow \lambda x.e}
\qquad
\frac{e_1 \Downarrow \lambda x.e \quad e_2 \Downarrow v_2 \quad [v_2/x]e_2 \Downarrow v}{e_1\,e_2 \Downarrow v}
$$

Big-step operational semantics specifications are compact and elegant, but they are not particularly *modular*. As a (contrived but illustrative) example, consider the addition of a incrementing

[2]The term *substructural operational semantics* merges structural operational semantics [Plo04], which we seek to generalize, and substructural logic, which forms the basis of our specification framework.

[3]In particular, we are leaving the first-order quantifiers implicit in this section and using an informal *object language* representation of syntax. The actual representation of syntax uses LF terms that adequately encode this object language, as discussed in Section 4.1.4.

$$
\begin{array}{rrrrcl}
\mathsf{store}\,\underline{5} & \mathsf{eval}\,(((\lambda x.\lambda y.y)\,\mathsf{count})\,\mathsf{count}) & & & \rightsquigarrow & \text{(by rule ev/app)} \\
\mathsf{store}\,\underline{5} & \mathsf{eval}\,((\lambda x.\lambda y.y)\,\mathsf{count}) & \mathsf{cont}\,(\square\,\mathsf{count}) & & \rightsquigarrow & \text{(by rule ev/app)} \\
\mathsf{store}\,\underline{5} & \mathsf{eval}\,(\lambda x.\lambda y.y) & \mathsf{cont}\,(\square\,\mathsf{count}) & \mathsf{cont}\,(\square\,\mathsf{count}) & \rightsquigarrow & \text{(by rule ev/lam)} \\
\mathsf{store}\,\underline{5} & \mathsf{retn}\,(\lambda x.\lambda y.y) & \mathsf{cont}\,(\square\,\mathsf{count}) & \mathsf{cont}\,(\square\,\mathsf{count}) & \rightsquigarrow & \text{(by rule ev/app1)} \\
\mathsf{store}\,\underline{5} & \mathsf{eval}\,(\mathsf{count}) & \mathsf{cont}\,((\lambda x.\lambda y.y)\,\square) & \mathsf{cont}\,(\square\,\mathsf{count}) & \rightsquigarrow & \text{(by rule ev/count)} \\
\mathsf{store}\,\underline{6} & \mathsf{retn}\,(\underline{5}) & \mathsf{cont}\,((\lambda x.\lambda y.y)\,\square) & \mathsf{cont}\,(\square\,\mathsf{count}) & \rightsquigarrow & \text{(by rule ev/app2)} \\
\mathsf{store}\,\underline{6} & \mathsf{eval}\,(\lambda y.y) & & \mathsf{cont}\,(\square\,\mathsf{count}) & \rightsquigarrow & \text{(by rule ev/lam)} \\
\mathsf{store}\,\underline{6} & \mathsf{retn}\,(\lambda y.y) & & \mathsf{cont}\,(\square\,\mathsf{count}) & \rightsquigarrow & \text{(by rule ev/app2)} \\
\mathsf{store}\,\underline{6} & \mathsf{eval}\,(\mathsf{count}) & & \mathsf{cont}\,((\lambda y.y)\,\square) & \rightsquigarrow & \text{(by rule ev/count)} \\
\mathsf{store}\,\underline{7} & \mathsf{retn}\,(\underline{6}) & & \mathsf{cont}\,((\lambda y.y)\,\square) & \rightsquigarrow & \text{(by rule ev/app2)} \\
 & & & \mathsf{store}\,\underline{7} \quad \mathsf{eval}\,(\underline{6}) & \rightsquigarrow & \text{(by rule ev/lam)} \\
 & & & \mathsf{store}\,\underline{7} \quad \mathsf{retn}\,(\underline{6}) & \not\rightsquigarrow &
\end{array}
$$

Figure 1.4: Evaluation with an imperative counter

counter count to the language of expressions e. The counter is a piece of runtime state, and every time count is evaluated, our runtime must return the value of the counter and then increments the counter.[4] To extend the big-step operational semantics with this new feature, we have to revise all the existing rules so that they mention the running counter:

$$
\frac{}{(\mathsf{count}, \underline{n}) \Downarrow (\underline{n}, \underline{n}+1)} \qquad \frac{}{(\lambda x.e, \underline{n}) \Downarrow (\lambda x.e, \underline{n})}
$$

$$
\frac{(e_1, \underline{n}) \Downarrow (\lambda x.e, \underline{n}_1) \quad (e_2, \underline{n}_1) \Downarrow (v_2, \underline{n}_2) \quad ([v_2/x]e_2, \underline{n}_2) \Downarrow (v, \underline{n}')}{(e_1\, e_2, \underline{n}) \Downarrow (v, \underline{n}')}
$$

The simple elegance of our big-step operational semantics has been tarnished by the need to deal with state, and each new stateful feature requires a similar revision. In contrast, our SSOS specification can tolerate the addition of a counter without revision to the existing rules; we just store the counter's value in an atomic proposition $\mathsf{store}(\underline{n})$ to the left of the $\mathsf{eval}(e)$ or $\mathsf{retn}(v)$ proposition in the ordered context. Because the rules ev/lam, ev/app, ev/app1, and ev/app2 are local, they will ignore this extra proposition, which only needs to be accessed by the rule ev/count.

$$
\mathsf{ev/count}: \quad \mathsf{store}\,\underline{n} \bullet \mathsf{eval}\,\mathsf{count} \rightarrowtail \{\mathsf{store}\,(\underline{n}+1) \bullet \mathsf{retn}\,\underline{n}\}
$$

In Figure 1.4, we give an example of evaluating $(((\lambda x.\lambda y.y)\,\mathsf{count})\,\mathsf{count})$ to a value with a starting counter value of $\underline{5}$. This specific solution – adding a counter proposition to the left of the eval or retn – is rather contrived. We want, in general, to be able to add arbitrary state, and this technique only allows us to add *one* piece of runtime state easily: if we wanted to

[4] To keep the language small, we can represent numerals $\underline{n}$ as Church numerals: $\underline{0} = (\lambda f.\lambda x.x)$, $\underline{1} = (\lambda f.\lambda x.fx)$, $\underline{2} = (\lambda f.\lambda x.f(fx))$, and so on. Then, $\underline{n}+1 = \lambda f.\lambda x.fe$ if $\underline{n} = \lambda f.\lambda x.e$.

introduce a *second* counter, where would it go? Nevertheless, the example does foreshadow how, in Part II of this dissertation, we will show that SSOS specifications in SLS allow for the modular specification of many programming language features.

An overarching theme of Part II is that we can have our cake and eat it too by deploying the *logical correspondence*, an idea that was developed jointly with Ian Zerny and that is explained in Chapter 5. In Chapter 6, we show how we can use the logical correspondence to directly connect the big-step semantics and SSOS specifications above; in fact, we can automatically and mechanically derive the latter from the former. As our example above showed, big-step operational semantics do not support combining the specification of pure features (like call-by-value evaluation) with the specification of a stateful feature (like the counter) – or, at least, doing so requires more than concatenating the specifications. Using the automatic transformations described in Chapter 6, we can specify pure features (like call-by-value evaluation) as a simpler big-step semantics specification, and then we can compose that specification with an SSOS specification of stateful features (like the counter) by mechanically transforming the big-step semantics part of the specification into SSOS. In SSOS, the extension is modular: the call-by-value specification can be extended by just adding new rules for the counter. Further transformations, developed in joint work with Pfenning [SP11a], create new opportunities for modular extension; this is the topic of Chapter 7.

Appendix B puts the logical correspondence to work by demonstrating that we can create a single coherent language specification by composing four different styles of specification. Pure features are given a natural semantics, whereas stateful, concurrent, and control features are specified at the most "high-level" SSOS specification style that is appropriate. The automatic transformations that are the focus of Part II then transform the specifications into a single coherent specification.

Transformations on SLS specifications also allow us to derive abstract analyses (such as control flow and alias analysis) directly from SSOS specifications. This methodology for program abstraction, *linear logical approximation*, is the focus of Chapter 8.

1.3 Invariants in substructural logic

A prominent theme in work on model checking and rewriting logic is expressing invariants in terms of temporal logics like LTL and verifying these properties with exhaustive state-space exploration [CDE+11, Chapter 10]. In Part III of this dissertation we offer an approach to invariants that is complementary to this model checking approach. From a programming languages perspective, invariants are often associated with *types*. Type invariants are well-formedness criteria on programs that are weak enough to be preserved by state transitions (a property called *preservation*) but strong enough to allow us to express the properties we expect to hold of all well-formed program states. In systems free of deadlock, a common property we want to hold is *progress* – a well-typed state is either final or it can evolve to some other state with a state transition. (Even in systems where deadlock is a possibility, progress can be handled by stipulating that a deadlocked state is final.) Progress and preservation together imply the safety property that a language is free of unspecified behavior.

Chapter 9 discusses the use of *generative signatures* to describe well-formedness invariants

$$
\begin{array}{rrrcl}
 & & \mathsf{gen\_state} & \rightsquigarrow & \text{(by rule gen/app2)} \\
 & \mathsf{gen\_state} & \mathsf{cont}\,((\lambda x.x)\,\Box) & \rightsquigarrow & \text{(by rule gen/app1)} \\
\mathsf{gen\_state} & \mathsf{cont}\,(\Box\,(\lambda z.e)) & \mathsf{cont}\,((\lambda x.x)\,\Box) & \rightsquigarrow & \text{(by rule gen/retn)} \\
\mathsf{retn}\,(\lambda y.y) & \mathsf{cont}\,(\Box\,(\lambda z.e)) & \mathsf{cont}\,((\lambda x.x)\,\Box) & \not\rightsquigarrow &
\end{array}
$$

Figure 1.5: Proving well-formedness of one of the states from Figure 1.3

of specifications. Generative signatures look like a generalization of context-free grammars, and they allow us to characterize contexts by a describing rewriting rules that generate legal or well-formed process states in the same way that context-free grammars characterize grammatical strings by describing rules that generate all grammatical strings.

In our example SSOS specification, a process state that consists of only a single $\mathsf{retn}(v)$ proposition is final, and a well-formed state is any state that consists of an atomic proposition $\mathsf{eval}(e)$ (where e is a closed expression) or $\mathsf{retn}(\lambda x.e)$ (where $\lambda x.e$ is a closed expression) to the left of a series of continuation frames $\mathsf{cont}(\Box\,e)$ or $\mathsf{cont}((\lambda x.e)\,\Box)$. We can characterize all such states as being generated from an initial atomic proposition gen_state under the following generative signature:

$$
\begin{array}{rl}
\mathsf{gen/eval}: & \mathsf{gen\_state} \rightarrowtail \{\mathsf{eval}(e)\} \\
\mathsf{gen/retn}: & \mathsf{gen\_state} \rightarrowtail \{\mathsf{retn}(\lambda x.e)\} \\
\mathsf{gen/app1}: & \mathsf{gen\_state} \rightarrowtail \{\mathsf{gen\_state} \bullet \mathsf{cont}(\Box\,e_2)\} \\
\mathsf{gen/app2}: & \mathsf{gen\_state} \rightarrowtail \{\mathsf{gen\_state} \bullet \mathsf{cont}((\lambda x.e)\,\Box)\}
\end{array}
$$

The derivation of one of the intermediate process states from Figure 1.3 is shown in Figure 1.5.

Well-formedness is a global property of specifications. Therefore, if we add state to the specification, we have to change the description of what counts as a final state and extend the grammar of well-formed process states. In the case of our counter extension, final states have a single $\mathsf{store}(\underline{n})$ proposition to the left of a single $\mathsf{retn}(v)$ proposition, and well-formed states are generated from an initial atomic proposition gen under the following extension to the previous generative signature:

$$
\begin{array}{rl}
\mathsf{gen/all}: & \mathsf{gen} \rightarrowtail \{\mathsf{gen\_store} \bullet \mathsf{gen\_state}\} \\
\mathsf{gen/store}: & \mathsf{gen\_store} \rightarrowtail \{\mathsf{store}(\underline{n})\}
\end{array}
$$

The grammar above describes a very coarse invariant of our SSOS specification, and it is possible to prove that specifications preserve more expressive invariants. An important class of examples are invariants about the types of expressions and process states, which will be considered in Chapter 9. For almost any SSOS specification more complicated than the one given above, type invariants are necessary for proving the progress theorem and concluding that the specification is safe – that is, free from undefined behavior. Chapter 10 will consider the use of generative invariants for proving safety properties of specifications.

1.4 Contributions

The three parts of this dissertation support three different aspects of our central thesis, which we can state as refined thesis statements that support the central thesis. We will presently discuss these supporting thesis statements along with the major contributions associated with each of the refinements.

> **Thesis (Part I):** *The methodology of structural focalization facilitates the derivation of logical frameworks as fragments of focused logics.*

The first major contribution of Part I of the dissertation is the development of *structural focalization* and its application to linear logic (Chapter 2) and ordered linear lax logic (Chapter 3). The second major contribution is the justification of the logical framework SLS as a fragment of a focused logic, generalizing the *hereditary substitution* methodology of Watkins [WCPW02].

> **Thesis (Part II):** *A logical framework based on a rewriting interpretation of substructural logic supports many styles of programming language specification. These styles can be formally classified and connected by considering general transformations on logical specifications.*

The major contribution of Part II is the development of the *logical correspondence*, a methodology for extending, classifying, inter-deriving, and modularly extending operational semantics specifications that are encoded in SLS, with an emphasis on SSOS specifications. The transformations in Chapter 6 connect big-step operational semantics specifications and the ordered abstract machine-style SSOS semantics that we introduced in Section 1.2. The destination-adding transformation given in Chapter 7 connects these specifications with the older *destination-passing style* of SSOS specification. In both chapters the transformations we discuss add new opportunities for modular extension – that is, new opportunities to add features to the language specification without revising existing rules. The transformations in these chapters are implemented in the SLS prototype, as demonstrated by the development in Appendix B.

> **Thesis (Part III):** *The SLS specification of the operational semantics of a programming language is a suitable basis for formal reasoning about properties of the specified language.*

We discuss two techniques for formal reasoning about the properties of SSOS specifications in SLS. In Chapter 8 we discuss the logical approximation methodology and show that it can be used to take SSOS specifications and derive known control flow and alias analyses that are correct by construction. The use of generative signatures to describe invariants is discussed in Chapter 9, and the use of these invariants to prove safety properties of programming languages is discussed in Chapter 10.

Part I

Focusing substructural logics

Chapter 2

Linear logic

In this chapter, we present linear logic as a logic with the ability to express aspects of state and state transition in a natural way. In Chapter 3 we will repeat the development from this chapter in a much richer and more expressive setting, and in Chapter 4 we will carve out a fragment of this logic to use as the basis of SLS, our logical framework of substructural logical specifications. These three chapters contribute to the overall thesis by focusing on the design of logical frameworks:

> **Thesis (Part I):** *The methodology of structural focalization facilitates the derivation of logical frameworks as fragments of focused logics.*

The purpose of this chapter is to introduce the methodology of *structural focalization*; this development is one of the major contributions of this work. Linear logic is a fairly simple logic that nevertheless allows us to consider many of the issues that will arise in richer substructural logics like the one considered in Chapter 3.

In Section 2.1 we motivate and discuss a traditional account of linear logic, and in Section 2.2 we discuss why this account is insufficient as a *logical framework* – derivations in linear logic suffice to establish the existence of a series of state transitions but do not adequately capture the structure of those transitions. Our remedy for this insufficiency comes in the form of *focusing*, Andreoli's restricted normal form for derivations in linear logic. We discuss focusing for a polarized presentation of linear logic in Section 2.3.

With focusing, we can describe *synthetic inference rules* (Section 2.4) that succinctly capture the structure of focused transitions. In Section 2.5 we discuss a number of ways of modifying the design of our focused logic to increase the expressiveness of synthetic inference rules; one of the alternatives we present, the introduction of *permeable atomic propositions*, will be generalized and incorporated into the focused presentation of ordered linear lax logic that we discuss in Chapter 3.

2.1 Introduction to linear logic

Logic as it has been traditionally understood and studied – both in its classical and intuitionistic varieties – treats the truth of a proposition as a *persistent resource*. That is, if we have evidence

$$
\begin{array}{lll}
A & ::= p \mid !A \mid \mathbf{1} \mid A \otimes B \mid A \multimap B & \\
\Gamma & ::= \cdot \mid \Gamma, A & \textit{(multiset)} \\
\Delta & ::= \cdot \mid \Delta, A & \textit{(multiset)}
\end{array}
$$

$\boxed{\Gamma; \Delta \longrightarrow A}$

$$
\frac{}{\Gamma; p \longrightarrow p}\ \mathit{id} \qquad \frac{\Gamma, A; \Delta, A \longrightarrow C}{\Gamma, A; \Delta \longrightarrow C}\ \mathit{copy}
$$

$$
\frac{\Gamma; \cdot \longrightarrow A}{\Gamma; \cdot \longrightarrow !A}\ !_R \qquad \frac{\Gamma, A; \Delta \longrightarrow C}{\Gamma; \Delta, !A \longrightarrow C}\ !_L \qquad \frac{}{\Gamma; \cdot \longrightarrow \mathbf{1}}\ \mathbf{1}_R \qquad \frac{\Gamma; \Delta \longrightarrow C}{\Gamma; \Delta, \mathbf{1} \longrightarrow C}\ \mathbf{1}_L
$$

$$
\frac{\Gamma; \Delta_1 \longrightarrow A \quad \Gamma; \Delta_2 \longrightarrow B}{\Gamma; \Delta_1, \Delta_2 \longrightarrow A \otimes B}\ \otimes_R \qquad \frac{\Gamma; \Delta, A, B \longrightarrow C}{\Gamma; \Delta, A \otimes B \longrightarrow C}\ \otimes_L
$$

$$
\frac{\Gamma; \Delta, A \longrightarrow B}{\Gamma; \Delta \longrightarrow A \multimap B}\ \multimap_R \qquad \frac{\Gamma; \Delta_1 \longrightarrow A \quad \Gamma; \Delta_2, B \longrightarrow C}{\Gamma; \Delta_1, \Delta_2, A \multimap B \longrightarrow C}\ \multimap_L
$$

Figure 2.1: Intuitionstic linear logic

for the truth of a proposition, we can ignore that evidence if it is not needed and reuse the evidence as many times as we need to. Throughout this document, "logic as it has been traditionally understood as studied" will be referred to as *persistent* logic to emphasize this treatment of evidence.

Linear logic, which was studied and popularized by Girard [Gir87], treats evidence as an *ephemeral* resource; the use of an ephemeral resource consumes it, at which point it is unavailable for further use. Linear logic, like persistent logic, comes in classical and intuitionistic flavors. We will favor intuitionistic linear logic in part because the propositions of intuitionistic linear logic (written A, B, C, ...) have a more natural correspondence with our physical intuitions about consumable resources. Linear conjunction $A \otimes B$ ("A tensor B") represents the resource built from the resources A and B; if you have both a bowl of soup *and* a sandwich, that resource can be represented by the proposition $\mathsf{soup} \otimes \mathsf{sandwich}$. Linear implication $A \multimap B$ ("A lolli B") represents a resource that can interact with another resource A to produce a resource B. One robot with batteries not included could be represented as the linear resource $(\mathsf{battery} \multimap \mathsf{robot})$, and the linear resource $(\mathsf{6bucks} \multimap \mathsf{soup} \otimes \mathsf{sandwich})$ represents the ability to use \$6 to obtain lunch – but only once.[1] Linear logic also has a connective $!A$ ("bang A" or "of course A") representing a persistent resource that can be used to generate any number of A resources, including zero. Your local Panera, which allows six dollars to be exchanged for both soup and a sandwich any number of times, can be represented as the resource $!(\mathsf{6bucks} \multimap \mathsf{soup} \otimes \mathsf{sandwich})$.

Figure 2.1 presents a standard sequent calculus for linear logic, in particular the *multiplica-*

[1]Conjunction will always bind more tightly than implication, so this is equivalent to the proposition $\mathsf{6bucks} \multimap (\mathsf{soup} \otimes \mathsf{sandwich})$.

tive, exponential fragment of intuitionistic linear logic (or *MELL*), so called because the connectives $\mathbf{1}$, $A \otimes B$, and $A \multimap B$ are considered to be the *multiplicative* connectives, and the connective $!A$ is the *exponential* connective of intuitionistic linear logic.[2] It corresponds most closely to Barber's dual intuitionistic linear logic [Bar96], but also to Andreoli's dyadic system [And92] and Chang et al.'s judgmental analysis of intuitionistic linear logic [CCP03].

The propositions of intuitionistic linear logic, and linear implication in particular, capture a notion of state change: we can *transition* from a state where we have both a battery and the battery-less robot (represented, as before, by the linear implication $\mathsf{battery} \multimap \mathsf{robot}$) to a state where we have the battery-endowed (and therefore presumably functional) robot (represented by the proposition robot). In other words, the proposition

$$\mathsf{battery} \otimes (\mathsf{battery} \multimap \mathsf{robot}) \multimap \mathsf{robot}$$

is provable in linear logic. These transitions can be chained together as well: if we start out with $\mathsf{6bucks}$ instead of $\mathsf{battery}$ but we also have the persistent ability to turn $\mathsf{6bucks}$ into a $\mathsf{battery}$ – just like we turned $6 into a bowl of soup and a salad at Panera – then we can ultimately get our working robot as well. Written as a series of transitions, the picture looks like this:

$6 (1)		*battery (1)*		*robot (1)*
battery-less robot (1)	$\rightsquigarrow$	*battery-less robot (1)*	$\rightsquigarrow$	*turn $6 into a battery (all you want)*
turn $6 into a battery (all you want)		*turn $6 into a battery (all you want)*		

In linear logic, these transitions correspond to the provability of the proposition

$$!(\mathsf{6bucks} \multimap \mathsf{battery}) \otimes \mathsf{6bucks} \otimes (\mathsf{battery} \multimap \mathsf{robot}) \multimap \mathsf{robot}.$$

A derivation of this proposition is given in Figure 2.2.[3]

It is precisely because linear logic contains this intuitive notion of state and state transition that a rich line of work, dating back to Chirimar's 1995 dissertation, has sought to use linear logic as a *logical framework* for describing stateful systems [Chi95, CP02, CPWW02, Pfe04, Mil09, PS09, CS09].

2.2 Logical frameworks

Generally speaking, logical frameworks use the *structure* of proofs in a logic (like linear logic) to describe the structures we're interested in (like the process of obtaining a robot). There are

[2] In this chapter we will mostly ignore the *additive* connectives of intuitionistic linear logic $\mathbf{0}$, $A \oplus B$, $\top$, and $A \,\&\, B$ and will entirely ignore the *first-order* connectives $\exists x.A$ and $\forall x.A$. The "why not" connective $?A$ from classical linear logic is sometimes treated as a second exponential connective in intuitionistic linear logic [CCP03], but we will never ask "why not?" in the context of this dissertation.

[3] In Chapter 4 (and Section 4.7.2 in particular) we see that this view isn't quite precise enough, and that the "best" representation of state change from the state A to the state B isn't really captured by derivations of the proposition $A \multimap B$ or by derivations of the sequent $\cdot; A \longrightarrow B$. However, this view remains a simple and useful one; Cervesato and Scedrov cover it thoroughly in the context of intuitionistic linear logic [CS09].

$$
\begin{array}{c}
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma; \mathsf{6bucks} \longrightarrow \mathsf{6bucks}}\,id \quad \dfrac{}{\Gamma; \mathsf{battery} \longrightarrow \mathsf{battery}}\,id}{\Gamma; \mathsf{6bucks}, \mathsf{6bucks} \multimap \mathsf{battery} \longrightarrow \mathsf{battery}}\,\multimap_L}{\Gamma; \mathsf{6bucks} \longrightarrow \mathsf{battery}}\,copy \quad \dfrac{}{\Gamma; \mathsf{robot} \longrightarrow \mathsf{robot}}\,id}{\Gamma; \mathsf{6bucks}, \mathsf{battery} \multimap \mathsf{robot} \longrightarrow \mathsf{robot}}\,\multimap_L}{\Gamma; \mathsf{6bucks} \otimes (\mathsf{battery} \multimap \mathsf{robot}) \longrightarrow \mathsf{robot}}\,\otimes_L}{\cdot; !(\mathsf{6bucks} \multimap \mathsf{battery}), \mathsf{6bucks} \otimes (\mathsf{battery} \multimap \mathsf{robot}) \longrightarrow \mathsf{robot}}\,!_L}{\cdot; !(\mathsf{6bucks} \multimap \mathsf{battery}) \otimes \mathsf{6bucks} \otimes (\mathsf{battery} \multimap \mathsf{robot}) \longrightarrow \mathsf{robot}}\,\otimes_L}{\cdot; \cdot \longrightarrow !(\mathsf{6bucks} \multimap \mathsf{battery}) \otimes \mathsf{6bucks} \otimes (\mathsf{battery} \multimap \mathsf{robot}) \multimap \mathsf{robot}}\,\multimap_R
\end{array}
$$

Figure 2.2: Proving that a transition is possible (where we let $\Gamma = \mathsf{6bucks} \multimap \mathsf{battery}$)

two related reasons why linear logic as described in Figure 2.1 is not immediately useful as a logical framework. First, the structure of the derivation in Figure 2.2 doesn't really match the intuitive two-step transition that we sketched out above. Second, there are *lots* of derivations of our example proposition according to the rules in Figure 2.1, even though there's only one "real" series of transitions that get us to a working robot. The use of $!L$, for instance, could be permuted up past the $\otimes L$ and then past the $\multimap L$ into the left branch of the proof. These differences represent inessential nondeterminism in proof construction – they just get in the way of the structure that we are trying to capture.

This is a general problem in the construction of logical frameworks. We'll discuss two solutions in the context of LF, a logical framework based on dependent type theory that has proved to be a suitable means of encoding a wide variety of deductive systems, such as logics and programming languages [HHP93]. The first solution is to define an appropriate equivalence class of proofs, and the second solution is to define a complete set of canonical proofs.

Defining an appropriate equivalence relation on proofs can be an effective way of handling this inessential nondeterminism. In linear logic as presented above, if the permutability of rules like $!_L$ and $\otimes_L$ is problematic, we can instead reason about *equivalence classes* of derivations. Derivations that differ only in the ordering of $!_L$ and $\otimes_L$ rules belong in the same equivalence class (which means we treat them as equivalent):

$$
\begin{array}{c}
\dfrac{\dfrac{\overset{\mathcal{D}}{\Gamma, A; \Delta, B, C \longrightarrow D}}{\Gamma, A; \Delta, B \otimes C \longrightarrow D}\,\otimes_L}{\Gamma; \Delta, !A, B \otimes C \longrightarrow D}\,!_L
\end{array}
\quad \equiv \quad
\begin{array}{c}
\dfrac{\dfrac{\overset{\mathcal{D}}{\Gamma, A; \Delta, B, C \longrightarrow D}}{\Gamma; \Delta, !A, B, C \longrightarrow D}\,!_L}{\Gamma; \Delta, !A, B \otimes C \longrightarrow D}\,\otimes_L
\end{array}
$$

In LF, lambda calculus terms (which correspond to derivations by the Curry-Howard correspondence) are considered modulo the least equivalence class that includes

* α-equivalence ($\lambda x.N \equiv \lambda y.N[y/x]$ if $y \notin FV(N)$),
* β-equivalence ($(\lambda x.\, M)N \equiv M[N/x]$ if $x \notin FV(N)$), and
* η-equivalence ($N \equiv \lambda x.N\, x$).

The weak normalization property for LF establishes that, given any typed LF term, we can find an equivalent term that is β-normal (no β-redexes of the form $(\lambda x.M)N$ exist) and η-long (replacing N with $\lambda x.N\, x$ anywhere would introduce a β-redex or make the term ill-typed). In any

given equivalence class of typed LF terms, all the β-normal and η-long terms are α-equivalent. Therefore, because α-equivalence is decidable, the equivalence of typed LF terms is decidable.

The uniqueness of β-normal and η-long terms within an equivalence class of lambda calculus terms (modulo α-equivalence, which we will henceforth take for granted) makes these terms useful as canonical representatives of equivalence classes. In Harper, Honsell, and Plotkin's original formulation of LF, a deductive system is said to be *adequately encoded* as an LF type family in the case that there is a compositional bijection between the formal objects in the deductive system and these β-normal, η-long representatives of equivalence classes [HHP93]. (Adequacy is a topic we will return to in Section 4.1.4.)

Modern presentations of LF, such as Harper and Licata's [HL07], follow the approach developed by Watkins et al. [WCPW02] and define the logical framework so that it only contains these β-normal, η-long *canonical forms* of LF. This presentation of LF is called Canonical LF to distinguish it from the original presentation of LF in which the β-normal, η-long terms are just a refinement of terms. A central component in this approach is *hereditary substitution*; in Chapter 3, we will make the connections between hereditary substitution and the focused cut admissibility property we prove in this chapter more explicit. Hereditary substitution also establishes a normalization property for LF. Using hereditary substitution we can easily take a regular LF term and transform it into a Canonical LF term. By a separate theorem, we can prove that the normalized term will be equivalent to the original term [MC12].

Our analogue to the canonical forms of LF will be the *focused derivations* of linear logic that are presented in the next section. In Section 2.3 below, we present focused linear logic and see that there is exactly one focused derivation of the proposition

$$!(\mathsf{6bucks} \multimap \mathsf{battery}) \otimes \mathsf{6bucks} \otimes (\mathsf{battery} \multimap \mathsf{robot}) \multimap \mathsf{robot}.$$

We will furthermore see that the structure of this derivation matches the intuitive transition interpretation, a point that is reinforced by the discussion of *synthetic inference rules* in Section 2.4.

2.3 Focused linear logic

Andreoli's original motivation for introducing focusing was not to describe a logical framework, it was to describe a foundational logic programming paradigm based on proof search in classical linear logic [And92]. The existence of multiple proofs that differ in inessential ways is particularly problematic for proof search, as inessential differences between derivations correspond to unnecessary choice points that a proof search procedure will need to backtrack over.

The development in this section introduces *structural focalization*, a methodology for deriving the correctness of a focused sequent calculus (Theorem 2.5 and Theorem 2.6, Section 2.3.7) as a consequence of the internal completeness (identity expansion, Theorem 2.3, Section 2.3.5) and internal soundness (cut admissibility, Theorem 2.4, Section 2.3.6) of the focused system. This methodology is a substantial refinement of the method used by Chaudhuri to establish the correctness of focused intuitionistic linear logic [Cha06], and because it relies on structural methods, structural focalization is more amenable to mechanized proof [Sim11]. Our focused sequent calculus also departs from Chaudhuri's by treating asynchronous rules as confluent rather than fixed, a point that will be discussed in Section 2.3.8.

2.3.1 Polarity

The first step in describing a focused sequent calculus is to classify connectives into two groups [And92]. Some connectives, such as linear implication $A \multimap B$, are called *asynchronous* because their right rules can always be applied eagerly, without backtracking, during bottom-up proof search. Other connectives, such as multiplicative conjunction $A \otimes B$, are called *synchronous* because their right rules cannot be applied eagerly. For instance, if we are trying to prove the sequent $A \otimes B \longrightarrow B \otimes A$, the $\otimes R$ rule cannot be applied eagerly; we first have to decompose $A \otimes B$ on the left using the $\otimes L$ rule. The terms asynchronous and synchronous make a bit more sense in a one-sided classical sequent calculus; in intuitionistic logics, it is common to call asynchronous connectives *right*-asynchronous and *left*-synchronous. Similarly, it is common to call synchronous connectives *right*-synchronous and *left*-asynchronous. We will instead use a different designation, calling the (right-)synchronous connectives *positive* ($!$, $\mathbf{0}$, $\oplus$, $\mathbf{1}$, and $\otimes$ in full propositional linear logic) and calling the (right-)asynchronous connectives *negative* ($\multimap$, $\top$ and $\&$ in full propositional linear logic); this assignment is called the proposition's *polarity*. Each atomic proposition must be assigned to have only one polarity, though this assignment can be made arbitrarily.

The nontrivial result of focusing is that it is possible to separate a proof into two strictly alternating phases. In *inversion* phases, positive propositions on the left and negative propositions on the right are eagerly and exhaustively decomposed using invertible rules.[4] In *focused* phases, a single proposition is selected (the proposition *in focus*, which is either a positive proposition in right focus or a negative proposition in left focus). This proposition is then decomposed repeatedly and exhaustively using rules that are mostly non-invertible.

If we consider this discipline applied to our robot example where all atoms have been assigned positive polarity, we would begin with an inversion phase, decomposing the negative implication on the right and the positive tensor and exponential on the left:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\mathsf{6bucks} \multimap \mathsf{battery}; \mathsf{6bucks}, \mathsf{battery} \multimap \mathsf{robot} \longrightarrow \mathsf{robot}}}{\mathsf{6bucks} \multimap \mathsf{battery}; \mathsf{6bucks} \otimes (\mathsf{battery} \multimap \mathsf{robot}) \longrightarrow \mathsf{robot}}\,\otimes_L}{\cdot; !(\mathsf{6bucks} \multimap \mathsf{battery}), \mathsf{6bucks} \otimes (\mathsf{battery} \multimap \mathsf{robot}) \longrightarrow \mathsf{robot}}\,!_L}{\cdot; !(\mathsf{6bucks} \multimap \mathsf{battery}) \otimes \mathsf{6bucks} \otimes (\mathsf{battery} \multimap \mathsf{robot}) \longrightarrow \mathsf{robot}}\,\otimes_L}{\cdot; \cdot \longrightarrow !(\mathsf{6bucks} \multimap \mathsf{battery}) \otimes \mathsf{6bucks} \otimes (\mathsf{battery} \multimap \mathsf{robot}) \multimap \mathsf{robot}}\,\multimap_R$$

Once we reach the topmost sequent in the above fragment, we have to pick a negative proposition on the left or a positive proposition on the right as our focus in order to proceed. The correct choice in this context is to pick the negative proposition $\mathsf{6bucks} \multimap \mathsf{battery}$ in the persistent context and decompose it using the non-invertible rule $\multimap_L$. Because the subformula $\mathsf{6bucks}$ is

[4]Synchronicity or polarity, a property of connectives, is closely connected to (and sometimes conflated with) a property of rules called *invertibility*; a rule is invertible if the conclusion of the rule implies the premises. So $\multimap R$ is invertible ($\Gamma; \Delta \longrightarrow A \multimap B$ implies $\Gamma; \Delta, A \longrightarrow B$) but $\multimap L$ is not ($\Gamma; \Delta, A \multimap B \longrightarrow C$ does not imply that $\Delta = \Delta_1, \Delta_2$ such that $\Gamma; \Delta_1 \longrightarrow A$ and $\Gamma; \Delta_2, B \longrightarrow C$). Rules that can be applied eagerly need to be invertible, so asynchronous connectives have invertible right rules and synchronous connectives have invertible left rules. Therefore, in the literature a common synonym for asynchronous/negative is *right-invertible*, and the analogous synonym for synchronous/positive is *left-invertible*.

$$
\begin{array}{rcl|rcl|rcl}
(\downarrow A^-)^\circ & = & (A^-)^\circ & & & & & & \\
(p^+)^\circ & = & p^+ & (p^+)^\oplus & = & p^+ & (p^+)^\ominus & = & \uparrow p^+ \\
(!A^-)^\circ & = & !(A^-)^\circ & (!A)^\oplus & = & !A^\ominus & (!A)^\ominus & = & \uparrow(!A^\ominus) \\
(\mathbf{1})^\circ & = & \mathbf{1} & (\mathbf{1})^\oplus & = & \mathbf{1} & (\mathbf{1})^\ominus & = & \uparrow\mathbf{1} \\
(A^+ \otimes B^+)^\circ & = & (A^+)^\circ \otimes (B^+)^\circ & (A \otimes B)^\oplus & = & A^\oplus \otimes B^\oplus & (A \otimes B)^\ominus & = & \uparrow(A^\oplus \otimes B^\oplus) \\
(\uparrow A^+)^\circ & = & (A^+)^\circ & & & & & & \\
(p^-)^\circ & = & p^- & (p^-)^\oplus & = & \downarrow p^- & (p^-)^\ominus & = & p^- \\
(A^+ \multimap B^-)^\circ & = & (A^+)^\circ \multimap (B^-)^\circ & (A \multimap B)^\oplus & = & \downarrow(A^\oplus \multimap B^\ominus) & (A \multimap B)^\ominus & = & A^\oplus \multimap B^\ominus
\end{array}
$$

Figure 2.3: De-polarizing and polarizing (with minimal shifts) propositions of MELL

positive and ends up on the right side in the subderivation, the focusing discipline requires that we prove it immediately with the id rule. Letting $\Gamma = \mathsf{6bucks} \multimap \mathsf{battery}$, this looks like this:

$$
\dfrac{\dfrac{\dfrac{}{\Gamma; \mathsf{6bucks} \longrightarrow \mathsf{6bucks}}\,id \qquad \begin{array}{c}\vdots\\ \Gamma; \mathsf{battery} \multimap \mathsf{robot}, \mathsf{battery} \longrightarrow \mathsf{robot}\end{array}}{\Gamma; \mathsf{6bucks}, \mathsf{battery} \multimap \mathsf{robot}, \mathsf{6bucks} \multimap \mathsf{battery} \longrightarrow \mathsf{robot}}\,\multimap_L}{\Gamma; \mathsf{6bucks}, \mathsf{battery} \multimap \mathsf{robot} \longrightarrow \mathsf{robot}}\,copy
$$

$$\vdots$$

The trace (that is, the pair of a single bottom sequent and a set of unproved top sequents) of an inversion phase stacked on top of a focused phase is called a *synthetic inference rule* by Chaudhuri, a point we will return to in Section 2.4.

2.3.2 Polarization

At this point, there is an important choice to make. One way forward is to treat positive and negative propositions as syntactic refinements of the set of all propositions, and to develop a focused presentation for intuitionistic linear logic with the connectives and propositions that we have already considered, as Chaudhuri did in [Cha06]. The other way forward is to treat positive and negative propositions as distinct syntactic classes A^+ and A^- with explicit inclusions, called *shifts*, between them. This is called *polarized* linear logic. The positive proposition $\downarrow A^-$, pronounced "downshift A" or "down A," has a subterm that is a negative proposition; the negative proposition $\uparrow A^+$, pronounced "upshift A" or "up A," has a subterm that is a positive proposition.

$$
\begin{array}{l}
A^+ ::= p^+ \mid \downarrow A^- \mid !A^- \mid \mathbf{1} \mid A^+ \otimes B^+ \\
A^- ::= p^- \mid \uparrow A^+ \mid A^+ \multimap B^-
\end{array}
$$

The relationship between unpolarized and polarized linear logic is given by two erasure functions $(A^+)^\circ$ and $(A^-)^\circ$ that wipe away all the shifts; this function is defined in Figure 2.3. In the other direction, every proposition in unpolarized linear logic has an polarized analogue with a minimal number of shifts, given by the functions $A^\oplus$ and $A^\ominus$ in Figure 2.3. Both of these functions are partial inverses of erasure, since $(A^\oplus)^\circ = (A^\ominus)^\circ = A$; we will generally refer to partial

$$
\begin{array}{rcl|rcl}
(p^+)^{m+} & = & \downarrow\uparrow p^+ & (p^+)^{m-} & = & \uparrow p^+ \\
(!A)^{m+} & = & \downarrow\uparrow!(A)^{m-} & (!A)^{m-} & = & \uparrow!(A)^{m-} \\
(\mathbf{1})^{m+} & = & \downarrow\uparrow\mathbf{1} & (\mathbf{1})^{m-} & = & \uparrow\mathbf{1} \\
(A \otimes B)^{m+} & = & \downarrow\uparrow((A)^{m+} \otimes (B)^{m+}) & (A \otimes B)^{m-} & = & \uparrow(A^{\oplus} \otimes B^{\oplus}) \\
 & & & & & \\
(p^-)^{m+} & = & \downarrow p^- & (p^-)^{m-} & = & \uparrow\downarrow p^- \\
(A \multimap B)^{m+} & = & \downarrow((A)^{m+} \multimap (B)^{m-}) & (A \multimap B)^{m-} & = & \uparrow\downarrow((A)^{m+} \multimap (B)^{m-})
\end{array}
$$

Figure 2.4: Fully-shifting polarization strategy for MELL

inverses of erasure as *polarization strategies*. The strategies $A^{\oplus}$ and $A^{\ominus}$ are minimal, avoiding shifts wherever possible, but there are many other possible strategies, such as the fully-shifting strategy that always adds either one or two shifts between every connective, which we can write as $(A)^{m+} = B^+$ and $(A)^{m-} = B^-$, defined in Figure 2.4.

Shifts turn out to have a profound impact on the structure of focused proofs, though erasure requires that they have no impact on *provability*. For instance, the proofs of A in Chaudhuri's focused presentation of linear logic are isomorphic to the proofs of $(A)^{\oplus}$ in the polarized logic discussed below,[5] whereas the proofs of $(A)^{m+}$ in polarized logic are isomorphic to the *unfocused* proofs of linear logic as described in Figure 2.1. Other polarization strategies correspond to different focused logics, as explored by Liang and Miller in [LM09], so the presentation of polarized linear logic below, like Liang and Miller's LJF, can be seen in two ways: as a focused logic in its own right, and as a framework for defining many focused logics (one per polarization strategy). As such, the strongest statement of the correctness of focusing is based on erasure: there is an unfocused derivation of $(A^+)^\circ$ or $(A^-)^\circ$ if and only if there is a focused derivation of A^+ or A^-. Most existing proofs of the completeness of focusing only verify a weaker property: that there is an unfocused derivation of A if and only if there is a focused derivation of $A^\bullet$, where $A^\bullet$ is some polarization strategy. The only exception seems to be Zeilberger's proof for classical persistent logic [Zei08b].

In this dissertation, we will be interested only in the structure of focused proofs, which corresponds to using the polarization strategy given by $A^{\oplus}$ and $A^{\ominus}$. Therefore, following Chaudhuri, it would be possible to achieve our objectives without the use of polarization. Our choice is largely based on practical considerations: the use of polarized logic simplifies the proof of identity expansion in Section 2.3.5 and the proof of completeness in Section 2.3.7. That said, polarized logic is an independently significant and currently active area of research. For instance, the Curry-Howard interpretation of polarized persistent logic has been studied by Levy as Call-by-Push-Value [Lev04]. The erasable influence of the shifts on the structure (but not the existence) of proofs is also important in the context of theorem proving. For instance, a theorem prover for polarized logic can imitate focused proof search by using the $(A)^{\oplus}$ polarization strategy and unfocused proof search by using the $(A)^{m+}$ polarization strategy [MP09].

[5]This isomorphism holds for Chaudhuri's focused presentation of linear logic precisely because his treatment of atomic propositions differs from Andreoli's. This isomorphism does not hold relative to focused systems that follow Andreoli's design, a point we will return to in Section 2.5.

2.3.3 Focused sequent calculus

Usually, focused logics are described as having multiple sequent forms. For intuitionistic logics, there need to be at least three sequent forms:

* $\Gamma; \Delta \vdash [A^+]$ (the *right focus* sequent, where the proposition A^+ is in focus),
* $\Gamma; \Delta \vdash C$ (the *inversion* sequent), and
* $\Gamma; \Delta, [A^-] \vdash C$ (the *left focus* sequent, where the proposition A^- is in focus).

It is also possible to distinguish a fourth sequent form, the *stable* sequents, inversion sequents $\Gamma; \Delta \vdash C$ where no asynchronous inversion remains to be done. A sufficient condition for stability is that the context Δ contains only negative propositions A^- and the succedent C is a positive proposition A^+. However, this cannot be a *necessary* condition for stability due to the presence of atomic propositions. If the process of inversion reaches a positive atomic proposition p^+ on the left or a negative atomic proposition p^- on the right, the proposition can be decomposed no further. When we reach an atomic proposition, we are therefore forced to *suspend* decomposition, either placing a suspended positive atomic proposition $\langle p^+ \rangle$ in Δ or placing a suspended negative proposition $\langle p^- \rangle$ as the succedent. For technical reasons discussed below in Section 2.3.4, our sequent calculus can handle arbitrary suspended propositions, not just suspended atomic propositions, and suspended propositions are always treated as stable, so $\Gamma; A^-, B^-, C^- \vdash D^+$ and $\Gamma; \langle A^+ \rangle, B^-, \langle C^+ \rangle \vdash \langle D^- \rangle$ are both stable sequents.

Another reasonable presentation of linear logic, and the one we will adopt in this section, uses only one sequent form, $\Gamma; \underline{\Delta} \vdash \underline{U}$, that generalizes what is allowed to appear in the linear context $\underline{\Delta}$ or in the succedent $\underline{U}$. We will use this interpretation to understand the logic described in Figure 2.5. In addition to propositions A^+, A^- and positive suspended positive propositions $\langle A^+ \rangle$, the grammar of contexts $\underline{\Delta}$ allows them to contain left focuses $[A^-]$. Likewise, a succedent $\underline{U}$ can be a stable positive proposition A^+, a suspended negative proposition $\langle A^- \rangle$, a focused positive proposition $[A^+]$, or an inverting negative proposition A^-. We will henceforth write Δ and U to indicate the refinements of $\underline{\Delta}$ and $\underline{U}$ that do not contain any focus.

By adding a side condition to the three rules $focus_R$, $focus_L$, and $copy$ that neither the context Δ nor the succedent U can contain an in-focus proposition $[A^+]$ or $[A^-]$, derivations can maintain the invariant that there is always at most one proposition in focus in any sequent, effectively restoring the situation in which there are three distinct judgments. Therefore, from this point on, we will only consider sequents $\Gamma; \underline{\Delta} \vdash \underline{U}$ with at most one focus. Pfenning, who developed this construction in [Pfe12c], calls this invariant the *focusing constraint*. The focusing constraint alone gives us what Pfenning calls a *chaining* logic [Pfe12c] and which Laurent calls a *weakly focused* logic [Lau04].[6] We obtain a fully focused logic by further restricting the three critical rules $focus_R$, $focus_L$, and $copy$ so that they only apply when the sequent below the line is stable. In light of this additional restriction, whenever we consider a focused sequent $\Gamma; \Delta, [A^-] \vdash U$ or $\Gamma; \Delta \vdash [A^+]$, we can assume that Δ and U are stable.

[6]Unfortunately, I made the meaning of "weak focusing" less precise by calling a different sort of logic weakly focused in [SP11b]. That weakly focused system had an additional restriction that invertible rules could *not* be applied when any other proposition was in focus, which is what Laurent called a strongly $+$-focused logic.

$$
\begin{array}{lll}
A^+ & ::= p^+ \mid \downarrow A^- \mid !A^- \mid \mathbf{1} \mid A^+ \otimes B^+ & \\
A^- & ::= p^- \mid \uparrow A^+ \mid A^+ \multimap B^- & \\
\Gamma & ::= \cdot \mid \Gamma, A^- & \textit{(multiset)} \\
\underline{\Delta} & ::= \cdot \mid \underline{\Delta}, A^+ \mid \underline{\Delta}, A^- \mid \underline{\Delta}, [A^-] \mid \underline{\Delta}, \langle A^+ \rangle & \textit{(multiset)} \\
\underline{U} & ::= A^- \mid A^+ \mid [A^+] \mid \langle A^- \rangle &
\end{array}
$$

$\boxed{\Gamma; \underline{\Delta} \vdash \underline{U}}$

$$
\frac{\Gamma; \Delta \vdash [A^+]}{\Gamma; \Delta \vdash A^+}\ \mathit{focus}^*_R \quad
\frac{\Gamma; \Delta, [A^-] \vdash U}{\Gamma; \Delta, A^- \vdash U}\ \mathit{focus}^*_L \quad
\frac{\Gamma, A^-; \Delta, [A^-] \vdash U}{\Gamma, A^-; \Delta \vdash U}\ \mathit{copy}^*
$$

$$
\frac{\Gamma; \Delta, \langle p^+ \rangle \vdash U}{\Gamma; \Delta, p^+ \vdash U}\ \eta^+ \quad
\frac{}{\Gamma; \langle A^+ \rangle \vdash [A^+]}\ \mathit{id}^+ \quad
\frac{\Gamma; \Delta \vdash \langle p^- \rangle}{\Gamma; \Delta \vdash p^-}\ \eta^- \quad
\frac{}{\Gamma; [A^-] \vdash \langle A^- \rangle}\ \mathit{id}^-
$$

$$
\frac{\Gamma; \Delta \vdash A^+}{\Gamma; \Delta \vdash \uparrow A^+}\ \uparrow_R \quad
\frac{\Gamma; \Delta, A^+ \vdash U}{\Gamma; \Delta, [\uparrow A^+] \vdash U}\ \uparrow_L \quad
\frac{\Gamma; \Delta \vdash A^-}{\Gamma; \Delta \vdash [\downarrow A^-]}\ \downarrow_R \quad
\frac{\Gamma; \Delta, A^- \vdash U}{\Gamma; \Delta, \downarrow A^- \vdash U}\ \downarrow_L
$$

$$
\frac{\Gamma; \cdot \vdash A^-}{\Gamma; \cdot \vdash [!A^-]}\ !_R \quad
\frac{\Gamma, A^-; \Delta \vdash U}{\Gamma; \Delta, !A^- \vdash U}\ !_L \quad
\frac{}{\Gamma; \cdot \vdash [\mathbf{1}]}\ \mathbf{1}_R \quad
\frac{\Gamma; \Delta \vdash U}{\Gamma; \Delta, \mathbf{1} \vdash U}\ \mathbf{1}_L
$$

$$
\frac{\Gamma; \Delta_1 \vdash [A^+] \quad \Gamma; \Delta_2 \vdash [B^+]}{\Gamma; \Delta_1, \Delta_2 \vdash [A^+ \otimes B^+]}\ \otimes_R \quad
\frac{\Gamma; \Delta, A^+, B^+ \vdash U}{\Gamma; \Delta, A^+ \otimes B^+ \vdash U}\ \otimes_L
$$

$$
\frac{\Gamma; \Delta, A^+ \vdash B^-}{\Gamma; \Delta \vdash A^+ \multimap B^-}\ \multimap_R \quad
\frac{\Gamma; \Delta_1 \vdash [A^+] \quad \Gamma; \Delta_2, [B^-] \vdash U}{\Gamma; \Delta_1, \Delta_2, [A^+ \multimap B^-] \vdash U}\ \multimap_L
$$

Figure 2.5: Focused intuitionstic linear logic

The persistent context of a focused derivation can always be weakened by adding more persistent resources. This weakening property can be phrased as an admissible rule, which we indicate using a dashed line:

$$
\frac{\Gamma; \underline{\Delta} \vdash \underline{U}}{\Gamma, \Gamma'; \underline{\Delta} \vdash \underline{U}}\ \mathit{weaken}
$$

In developments following Pfenning's structural cut admissibility methodology [Pfe00], it is critical that the weakening theorem *does not* change the structure of proofs: that the structure of the derivation $\Gamma; \underline{\Delta} \vdash \underline{U}$ is unchanged when we weaken it to $\Gamma, \Gamma'; \underline{\Delta} \vdash \underline{U}$. It turns out that the development in this chapter does not rely on this property.

Suspended propositions ($\langle A^+ \rangle$ and $\langle A^- \rangle$) and the four rules that interact with suspended propositions (id^+, id^-, η^+, and η^-) are the main nonstandard aspect of this presentation. The η^+ and η^- rules, which allow us to stop decomposing a proposition that we are eagerly de-

composing with invertible rules, are restricted to atomic propositions, and there is no other way for suspended propositions to be introduced into the context with rules. It seems reasonable to restrict the two rules that capture the identity principles, id^+ and id^-, to atomic propositions as well. However, the seemingly unnecessary generality of these two identity rules makes it much easier to establish the standard metatheory of this sequent calculus. To see why this is the case, we will turn our attention to suspended propositions and the four admissible rules (two focal substitution principles and two identity expansion principles) that interact with suspended propositions.

2.3.4 Suspended propositions

In unfocused sequent calculi, it is generally possible to restrict the id rule to atomic propositions (as shown in Figure 2.1). The general id rule, which concludes $\Gamma; A \longrightarrow A$ for all propositions A, is admissible just as the cut rule is admissible. But while the cut rule can be eliminated completely, the atomic id rule must remain. This is related to the logical interpretation of atomic propositions as stand-ins for unknown propositions. All sequent calculi, focused or unfocused, have the subformula property: every rule breaks down a proposition, either on the left or the right of the turnstile "$\vdash$", when read from bottom to top. We are unable to break down atomic propositions any further (they are unknown), thus the id rule is necessary at atomic propositions. If we substitute a concrete proposition for some atomic proposition, the structure of the proof stays exactly the same, except that instances of initial sequents become admissible instances of the identity theorem.

To my knowledge, all published proof systems for focused logic have incorporated a focused version of the id rule that also applies only to atomic propositions. This treatment is not incorrect and is obviously analogous to the id rule from the unfocused system. Nevertheless, I believe this to be a design error, and it is one that has historically made it unnecessarily difficult to prove the identity theorem for focused systems. The alternative developed in this chapter is the use of suspensions. Suspended positive propositions $\langle A^+ \rangle$ only appear in the linear context Δ, and suspended negative propositions $\langle A^- \rangle$ only appear as succedents. They are treated as stable (we never break down a suspended proposition) and are only used to immediately prove a proposition in focus with one of the identity rules id^+ or id^-. The rules id^+ and id^- are more general focused versions of the unfocused id rule. This extra generality does not influence the structure of proofs because suspended propositions can only be introduced into the context or the succedent by the η^+ and η^- rules, and those rules *are* restricted to atomic propositions.

Suspended positive propositions act much like regular variables in a natural deduction system. The positive identity rule id^+ allows us to prove any positive proposition given that the positive proposition appears suspended in the context. There is a corresponding substitution principle for focal substitutions that has a natural-deduction-like flavor: we can substitute a derivation right-focused on A^+ for a suspended positive proposition $\langle A^+ \rangle$ in a context.

Theorem 2.1 (Focal substitution (positive))**.**
If $\Gamma; \Delta \vdash [A^+]$ *and* $\Gamma; \underline{\Delta'}, \langle A^+ \rangle \vdash \underline{U}$*, then* $\Gamma; \underline{\Delta'}, \Delta \vdash \underline{U}$*.*

Proof. Straightforward induction over the second given derivation, as in a proof of regular substitution in a natural deduction system. If the second derivation is the axiom id^+, the result follows immediately using the first given derivation. □

As discussed above in Section 2.3.3, because we only consider focused sequents that are otherwise stable, we assume that Δ in the statement of Theorem 2.1 is stable by virtue of it appearing in the focused sequent $\Gamma; \Delta \vdash [A^+]$. The second premise $\Gamma; \underline{\Delta'}, \langle A^+ \rangle \vdash \underline{U}$, on the other hand, may be a right-focused sequent $\Gamma; \Delta', \langle A^+ \rangle \vdash [B^+]$, a left-focused sequent $\Gamma; \Delta'', [B^-], \langle A^+ \rangle \vdash U$, or an inverting sequent.

Suspended negative propositions are a bit less intuitive than suspended positive propositions. While a derivation of $\Gamma; \underline{\Delta'}, \langle A^+ \rangle \vdash \underline{U}$ is missing a premise that can be satisfied by a derivation of $\Gamma; \Delta \vdash [A^+]$, a derivation of $\Gamma; \underline{\Delta} \vdash \langle A^- \rangle$ is missing a *continuation* that can be satisfied by a derivation of $\Gamma; \Delta', [A^-] \vdash U$. The focal substitution principle, however, still takes the basic form of a substitution principle.

Theorem 2.2 (Focal substitution (negative))**.**
If $\Gamma; \underline{\Delta} \vdash \langle A^- \rangle$ *and* $\Gamma; \Delta', [A^-] \vdash U$*, then* $\Gamma; \Delta', \underline{\Delta} \vdash U$*.*

Proof. Straightforward induction over the *first* given derivation; if the first derivation is the axiom id^-, the result follows immediately using the second given derivation. □

Unlike cut admissibility, which we discuss in Section 2.3.6, both of the focal substitution principles are straightforward inductions over the structure of the derivation containing the suspended proposition. As an aside, when we encode the focused sequent calculus for persistent logic in LF, a suspended positive premise can be naturally encoded as a hypothetical right focus. This encoding makes the id^+ rule an instance of the hypothesis rule provided by LF and establishes Theorem 2.1 "for free" as an instance of LF substitution. This is possible to do for negative focal substitution as well, but it is counterintuitive and relies on a peculiar use of LF's uniform function space [Sim11].

The two substitution principles can be phrased as admissible rules for building derivations, like the $weaken$ rule above:

$$\frac{\Gamma; \Delta \vdash [A^+] \quad \Gamma; \underline{\Delta'}, \langle A^+ \rangle \vdash \underline{U}}{\Gamma; \underline{\Delta'}, \Delta \vdash \underline{U}}\ subst^+ \qquad \frac{\Gamma; \underline{\Delta} \vdash \langle A^- \rangle \quad \Gamma; \Delta', [A^-] \vdash U}{\Gamma; \Delta', \underline{\Delta} \vdash U}\ subst^-$$

Note the way in which these admissible substitution principles generalize the logic: $subst^+$ or $subst^-$ are the only rules we have discussed that allow us to introduce non-atomic suspended propositions, because only *atomic* suspended propositions are introduced explicitly by rules η^+ and η^-.

2.3.5 Identity expansion

Suspended propositions appear in Figure 2.5 in two places: in the identity rules, which we have just discussed and connected with the focal substitution principles, and in the rules marked η^+ and η^-, which are also the only mention of atomic propositions in the presentation. It is here that we need to make a critical shift of perspective from unfocused to focused logic. In an unfocused logic, the rules nondeterministically break down propositions, and the initial rule id puts an end to this process when an atomic proposition is reached. In a focused logic, the focus and inversion phases must break down a proposition *all the way* until a shift is reached. The two η rules are what put an end to this when an atomic proposition is reached, and they work hand-in-glove with the two id rules that allow these necessarily suspended propositions to successfully conclude a right or left focus.

Just as the id rule is a particular instance of the admissible identity sequent $\Gamma; A \longrightarrow A$ in unfocused linear logic, the atomic suspension rules η^+ and η^- are instances of an admissible *identity expansion* rule in focused linear logic:

$$\dfrac{\Gamma; \Delta, \langle A^+ \rangle \vdash U}{\Gamma; \Delta, A^+ \vdash U}\ \eta^+ \qquad \dfrac{\Gamma; \Delta \vdash \langle A^- \rangle}{\Gamma; \Delta \vdash A^-}\ \eta^-$$

In other words, the admissible identity expansion rules allow us to act as if the η^+ and η^- rules apply to *arbitrary* propositions, not just atomic propositions. The atomic propositions must be handled by an explicit rule, but the general principle is admissible.

The two admissible identity expansion rules above can be rephrased as an identity expansion theorem:

Theorem 2.3 (Identity expansion)**.**

* *If* $\Gamma; \Delta, \langle A^+ \rangle \vdash U$*, then* $\Gamma; \Delta, A^+ \vdash U$.
* *If* $\Gamma; \Delta \vdash \langle A^- \rangle$*, then* $\Gamma; \Delta \vdash A^-$.

Proof. Mutual induction over the structure of the proposition A^+ or A^-, with a critical use of focal substitution in each case.

Most of the cases of this proof are represented in Figure 2.6. The remaining case (for the multiplicative unit $\mathbf{1}$) is presented in Figure 2.7 along with the cases for the additive connectives $\mathbf{0}$, $\oplus$, $\top$, and $\&$, which are neglected elsewhere in this chapter. (Note that in Figures 2.6 and 2.7 we omit polarity annotations from propositions as they are always clear from the context.) $\square$

The admissible identity expansion rules fit with an interpretation of positive atomic propositions as stand-ins for arbitrary positive propositions and of negative atomic propositions as stand-ins for negative atomic propositions: if we substitute a proposition for some atomic proposition, all the instances of atomic suspension corresponding to that rule become admissible instances of identity expansion.

The usual identity principles are corollaries of identity expansion:

$$\dfrac{\dfrac{\overline{\Gamma; \langle A^+ \rangle \vdash [A^+]}\ id^+}{\Gamma; \langle A^+ \rangle \vdash A^+}\ focus_R}{\Gamma; A^+ \vdash A^+}\ \eta^+ \qquad \dfrac{\dfrac{\overline{\Gamma; [A^-] \vdash \langle A^- \rangle}\ id^-}{\Gamma; A^- \vdash \langle A^- \rangle}\ focus_L}{\Gamma; A^- \vdash A^-}\ \eta^-$$

$$\dfrac{\overset{\mathcal{D}}{\Gamma;\Delta,\langle{\downarrow}A\rangle \vdash U}}{\Gamma;\Delta,{\downarrow}A \vdash U}\,\eta^+ \quad\Longrightarrow\quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma;[A]\vdash\langle A\rangle}\,id^-}{\Gamma;A\vdash\langle A\rangle}\,focus_L}{\Gamma;A\vdash A}\,\eta^-}{\Gamma;A\vdash[{\downarrow}A]}\,{\downarrow}_R \qquad \overset{\mathcal{D}}{\Gamma;\Delta,\langle{\downarrow}A\rangle\vdash U}}{\Gamma;\Delta,A\vdash U}\,subst^+}{\Gamma;\Delta,{\downarrow}A\vdash U}\,{\downarrow}_L$$

$$\dfrac{\overset{\mathcal{D}}{\Gamma;\Delta,\langle !A\rangle \vdash U}}{\Gamma;\Delta,!A \vdash U}\,\eta^+ \quad\Longrightarrow\quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma,A;[A]\vdash\langle A\rangle}\,id^-}{\Gamma,A;\cdot\vdash\langle A\rangle}\,copy}{\Gamma,A;\cdot\vdash A}\,\eta^-}{\Gamma,A;\cdot\vdash[!A]}\,!_R \qquad \dfrac{\overset{\mathcal{D}}{\Gamma;\Delta,\langle !A\rangle\vdash U}}{\Gamma,A;\Delta,\langle !A\rangle\vdash U}\,weaken}{\Gamma,A;\Delta\vdash U}\,subst^+}{\Gamma;\Delta,!A\vdash U}\,!_L$$

$$\dfrac{\overset{\mathcal{D}}{\Gamma;\Delta,\langle A\otimes B\rangle \vdash U}}{\Gamma;\Delta,A\otimes B \vdash U}\,\eta^+ \quad\Longrightarrow\quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma;\langle A\rangle\vdash[A]}\,id^+ \quad \dfrac{}{\Gamma;\langle B\rangle\vdash[B]}\,id^+}{\Gamma;\langle A\rangle,\langle B\rangle\vdash[A\otimes B]}\,\otimes_R \qquad \overset{\mathcal{D}}{\Gamma;\Delta,\langle A\otimes B\rangle\vdash U}}{\Gamma;\Delta,\langle A\rangle,\langle B\rangle\vdash U}\,subst^+}{\Gamma;\Delta,\langle A\rangle,B\vdash U}\,\eta^+}{\Gamma;\Delta,A,B\vdash U}\,\eta^+}{\Gamma;\Delta,A\otimes B\vdash U}\,\otimes_L$$

$$\dfrac{\overset{\mathcal{D}}{\Gamma;\Delta\vdash\langle{\uparrow}A\rangle}}{\Gamma;\Delta\vdash{\uparrow}A}\,\eta^- \quad\Longrightarrow\quad \dfrac{\dfrac{\overset{\mathcal{D}}{\Gamma;\Delta\vdash\langle{\uparrow}A\rangle} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma;\langle A\rangle\vdash[A]}\,id^+}{\Gamma;\langle A\rangle\vdash A}\,focus_R}{\Gamma;A\vdash A}\,\eta^+}{\Gamma;[{\uparrow}A]\vdash A}\,{\uparrow}_L}{\Gamma;\Delta\vdash A}\,subst^-}{\Gamma;\Delta\vdash{\uparrow}A}\,{\uparrow}_R$$

$$\dfrac{\overset{\mathcal{D}}{\Gamma;\Delta\vdash\langle A\multimap B\rangle}}{\Gamma;\Delta\vdash A\multimap B}\,\eta^- \quad\Longrightarrow\quad \dfrac{\dfrac{\dfrac{\dfrac{\overset{\mathcal{D}}{\Gamma;\Delta\vdash\langle A\multimap B\rangle} \qquad \dfrac{\dfrac{}{\Gamma;\langle A\rangle\vdash[A]}\,id^+ \quad \dfrac{}{\Gamma;[B]\vdash\langle B\rangle}\,id^-}{\Gamma;\langle A\rangle,[A\multimap B]\vdash\langle B\rangle}\,\multimap_L}{\Gamma;\Delta,\langle A\rangle\vdash\langle B\rangle}\,subst^-}{\Gamma;\Delta,\langle A\rangle\vdash B}\,\eta^-}{\Gamma;\Delta,A\vdash B}\,\eta^+}{\Gamma;\Delta\vdash A\multimap B}\,\multimap_R$$

Figure 2.6: Identity expansion – restricting η^+ and η^- to atomic propositions

$$\dfrac{\overset{\mathcal{D}}{\Gamma;\Delta,\langle\mathbf{1}\rangle\vdash U}}{\Gamma;\Delta,\mathbf{1}\vdash U}\;\eta^+ \quad\Longrightarrow\quad \dfrac{\dfrac{\dfrac{}{\Gamma;\cdot\vdash[\mathbf{1}]}\;\mathbf{1}_R \quad \overset{\mathcal{D}}{\Gamma;\Delta,\langle\mathbf{1}\rangle\vdash U}}{\Gamma;\Delta\vdash U}\;subst^+}{\Gamma;\Delta,\mathbf{1}\vdash U}\;\mathbf{1}_L$$

$$\dfrac{\overset{\mathcal{D}}{\Gamma;\Delta,\langle\mathbf{0}\rangle\vdash U}}{\Gamma;\Delta,\mathbf{0}\vdash U}\;\eta^+ \quad\Longrightarrow\quad \dfrac{}{\Gamma;\Delta,\mathbf{0}\vdash U}\;\mathbf{0}_L$$

$$\dfrac{\overset{\mathcal{D}}{\Gamma;\Delta,\langle A\oplus B\rangle\vdash U}}{\Gamma;\Delta,A\oplus B\vdash U}\;\eta^+ \quad\Longrightarrow\quad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma;\langle A\rangle\vdash[A]}\;id^+}{\Gamma;\Delta,\langle A\rangle\vdash[A\oplus B]}\;\oplus_{R1} \quad \overset{\mathcal{D}}{\Gamma;\Delta,\langle A\oplus B\rangle\vdash U}}{\Gamma;\Delta,\langle A\rangle\vdash U}\;subst^+}{\Gamma;\Delta,A\vdash U}\;\eta^+ \quad \begin{array}{c}\vdots\\ \Gamma;\Delta,B\vdash U\end{array}}{\Gamma;\Delta,A\oplus B\vdash U}\;\oplus_L$$

$$\dfrac{\overset{\mathcal{D}}{\Gamma;\Delta\vdash\langle\top\rangle}}{\Gamma;\Delta\vdash\top}\;\eta^- \quad\Longrightarrow\quad \dfrac{}{\Gamma;\Delta\vdash\top}\;\top_R$$

$$\dfrac{\overset{\mathcal{D}}{\Gamma;\Delta\vdash\langle A\,\&\,B\rangle}}{\Gamma;\Delta\vdash A\,\&\,B}\;\eta^- \quad\Longrightarrow\quad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma;[A]\vdash\langle A\rangle}\;id^-}{\Gamma;[A\,\&\,B]\vdash\langle A\rangle}\;\&_{L1} \quad \overset{\mathcal{D}}{\Gamma;\Delta\vdash\langle A\,\&\,B\rangle}}{\Gamma;\Delta\vdash\langle A\rangle}\;subst^-}{\Gamma;\Delta\vdash A}\;\eta^- \quad \begin{array}{c}\vdots\\ \Gamma;\Delta\vdash B\end{array}}{\Gamma;\Delta\vdash A\,\&\,B}\;\&_R$$

Figure 2.7: Identity expansion for units and additive connectives

2.3.6 Cut admissibility

Cut admissibility, Theorem 2.4 below, mostly follows the well-worn contours of a structural cut admissibility argument [Pfe00]. A slight inelegance of the proof given here is that some very similar cases must be considered more than once in different parts of the proof. The right commutative cases – cases in which the last rule in the second given derivation is an invertible rule that is not decomposing the principal cut formula A^+ – must be repeated in parts 1 and 4, for instance. (Pfenning's classification of the cases of cut admissibility into principal, left commutative, and right commutative cuts is discussed in Section 3.4.) In addition to this duplication, the proof of part 4 is almost identical in form to the proof of part 5. The proof of cut admissibility in the next chapter will eliminate both forms of duplication.

The most important caveat about cut admissibility is that it is only applicable in the absence of any non-atomic suspended propositions. If we did not make this restriction, then in Theorem 2.4, part 1, we might encounter a derivation of $\Gamma;\langle A\otimes B\rangle\vdash[A\otimes B]$ that concludes with id^+ being

cut into the derivation

$$\dfrac{\begin{array}{c}\mathcal{E}\\ \Gamma; \Delta', A, B \vdash U\end{array}}{\Gamma; \Delta', A \otimes B \vdash U}\ \otimes_R$$

in which case there is no clear way to proceed and prove $\Gamma; \Delta', \langle A \otimes B \rangle \vdash U$.

Theorem 2.4 (Cut admissibility)**.** *For all Γ, A^+, A^-, Δ, Δ', and U that do not contain any non-atomic suspended propositions:*

1. *If $\Gamma; \Delta \vdash [A^+]$ and $\Gamma; \Delta', A^+ \vdash U$ (where Δ is stable), then $\Gamma; \Delta', \Delta \vdash U$.*
2. *If $\Gamma; \Delta \vdash A^-$ and $\Gamma; \Delta', [A^-] \vdash U$ (where Δ, Δ', and U are stable), then $\Gamma; \Delta', \Delta \vdash U$.*
3. *If $\Gamma; \underline{\Delta} \vdash A^+$ and $\Gamma; \Delta', A^+ \vdash U$, (where Δ' and U are stable), then $\Gamma; \Delta', \underline{\Delta} \vdash U$.*
4. *If $\Gamma; \Delta \vdash A^-$ and $\Gamma; \underline{\Delta'}, A^- \vdash \underline{U}$, (where Δ is stable), then $\Gamma; \underline{\Delta'}, \Delta \vdash \underline{U}$.*
5. *If $\Gamma; \cdot \vdash A^-$ and $\Gamma, A^-; \underline{\Delta'} \vdash \underline{U}$, then $\Gamma; \underline{\Delta'} \vdash \underline{U}$.*

Parts 1 and 2 are where most of the action happens, but there is a sense in which the *necessary* cut admissibility property is contained in structure of parts 3, 4, and 5 – these are the cases used to prove the completeness of focusing (Theorem 2.6). The discrepancy between the stability restrictions demanded for part 1 and part 2 is discussed below; this peculiarity is justified by the fact that these two parts need only be general enough to prove parts 3, 4, and 5.

Proof. The proof is by induction: in each invocation of the induction hypothesis, either the principal cut formula A^+ or A^- gets smaller or else it stays the same and the "part size" (1-5) gets smaller. When the principal cut formula and the part size remain the same, either the first given derivation gets smaller (part 3) or the second given derivation gets smaller (parts 1, 4 and 5).

This termination argument is a refinement of the standard structural termination argument for cut admissibility in unfocused logics [Pfe00] – in part 3, we don't need to know that the second given derivation stays the same size, and in parts 1, 4, and 5 we don't need to know that the first given derivation stays the same size. This refined termination argument is the reason that we do not need to prove that admissible weakening preserves the structure of proofs.

We schematically present one or two illustrative cases for each part of the proof.

Part 1 (positive principal cuts, right commutative cuts)

(Δ_1, Δ_2 stable are stable by assumption)

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}_1\\ \Gamma; \Delta_1 \vdash [A_1^+]\end{array} \quad \begin{array}{c}\mathcal{D}_2\\ \Gamma; \Delta_2 \vdash [A_2^+]\end{array}}{\Gamma; \Delta_1, \Delta_2 \vdash [A_1^+ \otimes A_2^+]}\ \otimes_R \quad \dfrac{\begin{array}{c}\mathcal{E}'\\ \Gamma; \Delta', A_1^+, A_2^+ \vdash U\end{array}}{\Gamma; \Delta', A_1^+ \otimes A_2^+ \vdash U}\ \otimes_L}{\Gamma; \Delta', \Delta_1, \Delta_2 \vdash U}\ cut(1)$$

$$\Longrightarrow \quad \dfrac{\begin{array}{c}\mathcal{D}_1\\ \Gamma; \Delta_1 \vdash [A_1^+]\end{array} \quad \dfrac{\begin{array}{c}\mathcal{D}_2\\ \Gamma; \Delta_2 \vdash [A_2^+]\end{array} \quad \begin{array}{c}\mathcal{E}'\\ \Gamma; \Delta', A_1^+, A_2^+ \vdash U\end{array}}{\Gamma; \Delta', A_1^+, \Delta_2 \vdash U}\ cut(1)}{\Gamma; \Delta', \Delta_1, \Delta_2 \vdash U}\ cut(1)$$

(Δ is stable by assumption)

$$\dfrac{\overset{\mathcal{D}}{\Gamma;\Delta\vdash[A^+]}\quad \dfrac{\overset{\mathcal{E}'}{\Gamma;\Delta',B_1^+,B_2^+,A^+\vdash U}}{\Gamma;\Delta',B_1^+\otimes B_2^+,A^+\vdash U}\otimes_L}{\Gamma;\Delta',B_1^+\otimes B_2^+,\Delta\vdash U}\,cut(1) \quad\Longrightarrow\quad \dfrac{\dfrac{\overset{\mathcal{D}}{\Gamma;\Delta\vdash[A^+]}\quad \overset{\mathcal{E}'}{\Gamma;\Delta',B_1^+,B_2^+,A^+\vdash U}}{\Gamma;\Delta',B_1^+,B_2^+,\Delta\vdash U}\,cut(1)}{\Gamma;\Delta',B_1^+\otimes B_2^+,\Delta\vdash U}\otimes_L$$

Part 2 (negative principal cuts)

(Δ, Δ', Δ'_A, and U are stable by assumption)

$$\dfrac{\dfrac{\overset{\mathcal{D}'}{\Gamma;\Delta,A_1^+\vdash A_2^-}}{\Gamma;\Delta\vdash A_1^+\multimap A_2^-}\multimap_R\quad \dfrac{\overset{\mathcal{E}_1}{\Gamma;\Delta'_A\vdash[A_1^+]}\quad \overset{\mathcal{E}_2}{\Gamma;\Delta',[A_2^-]\vdash U}}{\Gamma;\Delta',\Delta'_A,[A_1^+\multimap A_2^-]\vdash U}\multimap_L}{\Gamma;\Delta',\Delta'_A,\Delta\vdash U}\,cut(2)$$

$$\Longrightarrow\quad \dfrac{\dfrac{\overset{\mathcal{E}_1}{\Gamma;\Delta'_A\vdash[A_1^+]}\quad \overset{\mathcal{D}'}{\Gamma;\Delta,A_1^+\vdash A_2^-}}{\Gamma;\Delta'_A,\Delta\vdash A_2^-}\,cut(1)\quad \overset{\mathcal{E}_2}{\Gamma;\Delta',[A_2^-]\vdash U}}{\Gamma;\Delta',\Delta'_A,\Delta\vdash U}\,cut(2)$$

Part 3 (left commutative cuts)

(Δ' and U are stable by assumption, Δ is stable by the side condition on rule $focus_R$)

$$\dfrac{\dfrac{\overset{\mathcal{D}'}{\Gamma;\Delta\vdash[A^+]}}{\Gamma;\Delta\vdash A^+}\,focus_R\quad \overset{\mathcal{E}}{\Gamma;\Delta',A^+\vdash U}}{\Gamma;\Delta',\Delta\vdash U}\,cut(3) \quad\Longrightarrow\quad \dfrac{\overset{\mathcal{D}'}{\Gamma;\Delta\vdash[A^+]}\quad \overset{\mathcal{E}}{\Gamma;\Delta',A^+\vdash U}}{\Gamma;\Delta',\Delta\vdash U}\,cut(1)$$

(Δ' and U are stable by assumption)

$$\dfrac{\dfrac{\overset{\mathcal{D}'}{\Gamma;\Delta,B_1^+,B_2^+\vdash A^+}}{\Gamma;\Delta,B_1^+\otimes B_2^+\vdash A^+}\otimes_L\quad \overset{\mathcal{E}}{\Gamma;\Delta',A^+\vdash U}}{\Gamma;\Delta',\Delta,B_1^+\otimes B_2^+\vdash A^+}\,cut(3) \quad\Longrightarrow\quad \dfrac{\dfrac{\overset{\mathcal{D}'}{\Gamma;\Delta,B_1^+,B_2^+\vdash A^+}\quad \overset{\mathcal{E}}{\Gamma;\Delta',A^+\vdash U}}{\Gamma;\Delta',\Delta,B_1^+,B_2^+\vdash A^+}\,cut(3)}{\Gamma;\Delta',\Delta,B_1^+\otimes B_2^+\vdash A^+}\otimes_L$$

$$
\begin{array}{rcl}
(\Gamma)^\circ & & \\
(\cdot)^\circ & = & \cdot \\
(\Gamma, A^-)^\circ & = & (\Gamma)^\circ, (A^-)^\circ
\end{array}
\quad\Bigg|\quad
\begin{array}{rcl}
(\underline{\Delta})^\circ & & \\
(\cdot)^\circ & = & \cdot \\
(\Delta, A^+)^\circ & = & (\Delta)^\circ, (A^+)^\circ \\
(\Delta, A^-)^\circ & = & (\Delta)^\circ, (A^-)^\circ \\
(\Delta, [A^-])^\circ & = & (\Delta)^\circ, (A^-)^\circ \\
(\Delta, \langle p^+ \rangle)^\circ & = & (\Delta)^\circ, p^+
\end{array}
\quad\Bigg|\quad
\begin{array}{rcl}
(\underline{U})^\circ & & \\
(A^-)^\circ & = & (A^-)^\circ \\
(A^+)^\circ & = & (A^+)^\circ \\
([A^+])^\circ & = & (A^+)^\circ \\
(\langle p^- \rangle)^\circ & = & p^-
\end{array}
$$

Figure 2.8: Lifting erasure and polarization (Figure 2.3) to contexts and succeedents

Part 4 (right commutative cuts)

(Δ is stable by assumption, Δ' and U are stable by the side condition on rule $focus_R$)

$$
\dfrac{\overset{\mathcal{D}}{\Gamma; \Delta \vdash A^-} \quad \dfrac{\overset{\mathcal{E}'}{\Gamma; \Delta', [A^-] \vdash U}}{\Gamma; \Delta', A^- \vdash U}\, focus_R}{\Gamma; \Delta', \Delta \vdash U}\, cut(4)
\quad \Longrightarrow \quad
\dfrac{\overset{\mathcal{D}}{\Gamma; \Delta \vdash A^-} \quad \overset{\mathcal{E}'}{\Gamma; \Delta', [A^-] \vdash U}}{\Gamma; \Delta', \Delta \vdash U}\, cut(2)
$$

Part 5 (persistent right commutative cuts)

$$
\dfrac{\overset{\mathcal{D}}{\Gamma; \cdot \vdash A^-} \quad \dfrac{\overset{\mathcal{E}'}{\Gamma, A^-; \cdot \vdash B^-}}{\Gamma, A^-; \cdot \vdash [!B^-]}\, !_R}{\Gamma; \cdot \vdash [!B^-]}\, cut(5)
\quad \Longrightarrow \quad
\dfrac{\dfrac{\overset{\mathcal{D}}{\Gamma; \cdot \vdash A^-} \quad \overset{\mathcal{E}'}{\Gamma, A^-; \cdot \vdash B^-}}{\Gamma; \cdot \vdash B^-}\, cut(5)}{\Gamma; \cdot \vdash [!B^-]}\, !_R
$$

All the other cases follow the same pattern. □

As noted above, there is a notable asymmetry between part 1 of the theorem, which does not require stability of Δ' and U in the second given derivation $\Gamma; \Delta', A^+ \vdash U$, and part 2 of the theorem, which does require stability of Δ in the first given derivation $\Gamma; \Delta \vdash A^-$. The theorem would still hold for non-stable Δ, but we do not need the more general theorem, and the less general theorem is easier to prove – it allows us to avoid duplicating the left commutative cuts between parts 2 and 3. On the other hand, we cannot make the theorem more specific, imposing extra stability conditions on part 1, without fixing the order in which invertible rules are applied. Fixing the order in which invertible rules are applied has some other advantages as well; this is a point we will return to in Section 2.3.8.

2.3.7 Correctness of focusing

Now we will prove the correctness property for the focused, polarized logic that we discussed in Section 2.3.1: that there is an unfocused derivation of $(A^+)^\circ$ or $(A^-)^\circ$ if and only if there is a focused derivation of A^+ or A^-. The proof requires us to lift our erasure function to contexts and succedents, which is done in Figure 2.8. Note that erasure is only defined on focused sequents

$\Gamma; \underline{\Delta} \vdash \underline{U}$ when all suspended propositions are atomic. We are justified in making this restriction because non-atomic suspended propositions cannot arise in the process of proving a proposition A^+ or A^- in an empty context, and we are required to make this restriction due to the analogous restrictions on cut admissibility (Theorem 2.4).

Theorems 2.5 and 2.6 therefore implicitly carry the same extra condition that we put on the cut admissibility theorem: that $\underline{\Delta}$ and $\underline{U}$ must contain only atomic suspended propositions.

Theorem 2.5 (Soundness of focusing)**.** *If* $\Gamma; \underline{\Delta} \vdash \underline{U}$*, then* $\Gamma^\circ; \underline{\Delta}^\circ \longrightarrow \underline{U}^\circ$.

Proof. By straightforward induction on the given derivation; in each case, the result either follows directly by invoking the induction hypothesis (in the case of rules like $\uparrow_R$) or by invoking the induction hypothesis and applying one rule from Figure 2.1 (in the case of rules like $\otimes_R$). $\square$

Theorem 2.6 (Completeness of focusing)**.** *If* $\Gamma^\circ; \Delta^\circ \longrightarrow U^\circ$*, where* Δ *and* U *are stable, then* $\Gamma; \Delta \vdash U$.

Proof. By induction on the first given derivation. Each rule in the unfocused system (Figure 2.1) corresponds to one *unfocused admissibility lemma*, plus a some extra steps.

These extra steps arise are due to the generality of erasure. If we know that $!A = (C^+)^\circ$ (as in the case for $!_R$ below), then by case analysis on the structure of C^+, C^+ must be either $!B^-$ (for some B^-) or $\downarrow C_1^-$ (for some C_1^-). In the latter case, by further case analysis on C_1^- we can see that C_1^- must equal $\uparrow C_2^+$ (for some C_2^+). But then C_2^+ can be either $!B_2^-$ or $\downarrow C_3^-$; in the latter case $C^+ = \downarrow\uparrow\downarrow C_3^-$, and this can go on arbitrarily long (but not forever, because C^- is a finite term). So we say that, by induction on the structure of C^+, there exists an A^- such that $C^+ = \downarrow\uparrow \ldots \downarrow\uparrow !A^-$ and $A = (A^-)^\circ$. Depending on the case, we then repeatedly apply either the $\uparrow\downarrow_R$ rule or the $\downarrow\uparrow_L$ rule, both of which are derived below, to eliminate all the extra shifts. (Zero or more instances of a rule are indicated by a double-ruled inference rule.)

$$\frac{\Gamma; \Delta \vdash A^+}{\Gamma; \Delta \vdash \downarrow\uparrow A^+}\ \downarrow\uparrow_R = \frac{\dfrac{\dfrac{\Gamma; \Delta \vdash A^+}{\Gamma; \Delta \vdash \uparrow A^+}\ \uparrow_R}{\Gamma; \Delta \vdash [\downarrow\uparrow A^+]}\ \downarrow_R}{\Gamma; \Delta \vdash \downarrow\uparrow A^+}\ focus_R \qquad \frac{\Gamma; \Delta, A^- \vdash U}{\Gamma; \Delta, \uparrow\downarrow A^- \vdash U}\ \uparrow\downarrow_L = \frac{\dfrac{\dfrac{\Gamma; \Delta, A^- \vdash U}{\Gamma; \Delta, \downarrow A^- \vdash U}\ \downarrow_R}{\Gamma; \Delta, [\uparrow\downarrow A^-] \vdash U}\ \uparrow_L}{\Gamma; \Delta, \uparrow\downarrow A^- \vdash U}\ focus_L$$

We will describe a few cases to illustrate how unfocused admissibility lemmas work.

Rule *copy*: We are given $\Gamma^\circ, A; \Delta^\circ, A \longrightarrow U^\circ$, which is used to derive $\Gamma^\circ, A; \Delta^\circ \longrightarrow U^\circ$. We know $A = (A^-)^\circ$. By the induction hypothesis, we have $\Gamma, A^-; \Delta, A^- \vdash U$, and we conclude with the unfocused admissibility lemma $copy_u$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, A^-; [A^-] \vdash \langle A^- \rangle}\ id^-}{\Gamma, A^-; \cdot \vdash \langle A^- \rangle}\ copy}{\Gamma, A^-; \cdot \vdash A^-}\ \eta^- \qquad \Gamma, A^-; \Delta, A^- \vdash U}{\Gamma, A^-; \Delta \vdash U}\ cut(5)$$

Rule $!_L$: We are given $\Gamma^\circ, A; \Delta^\circ \longrightarrow U^\circ$, which is used to derive $\Gamma^\circ; \Delta^\circ, !A \longrightarrow U^\circ$. We know $!A = (C^-)^\circ$; by induction on the structure of C^- there exists A^- such that $C^- =$

$\uparrow\downarrow \ldots \downarrow\uparrow !A^-$. By the induction hypothesis, we have $\Gamma, A^-; \Delta \vdash U$, and we conclude by the unfocused admissibility lemma $!_{uL}$, which is derivable:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, A^-; \Delta \vdash U}{\Gamma; \Delta, !A^- \vdash U}\ !_L}{\Gamma; \Delta, [\uparrow !A^-] \vdash U}\ \uparrow_L}{\Gamma; \Delta, \uparrow !A^- \vdash U}\ \mathit{focus}_L}{\Gamma; \Delta, \uparrow\downarrow \ldots \downarrow\uparrow !A^- \vdash U}\ \uparrow\downarrow_L$$

Rule $!_R$: We are given $\Gamma^\circ; \cdot \longrightarrow A$, which is used to derive $\Gamma^\circ; \cdot \longrightarrow !A$. We know $!A = (C^+)^\circ$; by induction on the structure of C^+ there exists A^- such that $C^+ = \downarrow\uparrow \ldots \downarrow\uparrow !A^-$. By the induction hypothesis, we have $\Gamma; \cdot \vdash \downarrow A^-$, and we conclude by the unfocused admissibility lemma $!_{uR}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma; \cdot \vdash \downarrow A^-}{\Gamma; \cdot \vdash \uparrow\downarrow A^-}\ \uparrow_R \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, \uparrow\downarrow A^-; [A^-] \vdash \langle A^- \rangle}\ id^-}{\Gamma, \uparrow\downarrow A^-; A^- \vdash \langle A^- \rangle}\ \mathit{focus}_L}{\Gamma, \uparrow\downarrow A^-; \downarrow A^- \vdash \langle A^- \rangle}\ \downarrow_L}{\Gamma, \uparrow\downarrow A^-; [\uparrow\downarrow A^-] \vdash \langle A^- \rangle}\ \uparrow_L}{\Gamma, \uparrow\downarrow A^-; \cdot \vdash \langle A^- \rangle}\ \mathit{copy}}{\Gamma, \uparrow\downarrow A^-; \cdot \vdash A^-}\ \eta^-}{\Gamma, \uparrow\downarrow A^-; \cdot \vdash [!A^-]}\ !_R}{\Gamma, \uparrow\downarrow A^-; \cdot \vdash !A^-}\ \mathit{focus}_R}{\Gamma; \cdot \vdash !A^-}\ \mathit{cut}(5)}{\Gamma; \cdot \vdash \downarrow\uparrow \ldots \downarrow\uparrow !A^-}\ \downarrow\uparrow_R$$

Rule $\multimap_L$: We are given $\Gamma^\circ; \Delta_A^\circ \longrightarrow A$ and $\Gamma^\circ; \Delta^\circ, B \longrightarrow U^\circ$, which are used to derive $\Gamma^\circ; \Delta_A^\circ, \Delta^\circ, A \multimap B \longrightarrow U$. We know $A \multimap B = (C^-)^\circ$; by induction on the structure of C^- there exist A^+ and B^- such that $A = (A^+)^\circ$, $B = (B^-)^\circ$, and $C^- = \uparrow\downarrow \ldots \uparrow\downarrow (A^+ \multimap B^-)$. By the induction hypothesis, we have $\Gamma; \Delta_A \vdash A^+$ and $\Gamma; \Delta, B^- \vdash U$, and we conclude by the unfocused admissibility lemma $\multimap_{uL}$:

$$\dfrac{\dfrac{\dfrac{\Gamma; \Delta_A \vdash A^+ \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma; \langle A^+ \rangle \vdash [A^+]}\ id^+ \qquad \dfrac{}{\Gamma; [B^-] \vdash \langle B^- \rangle}\ id^-}{\Gamma; \langle A^+ \rangle, [A^+ \multimap B^-] \vdash \langle B^- \rangle}\ \multimap_L}{\Gamma; \langle A^+ \rangle, A^+ \multimap B^- \vdash \langle B^- \rangle}\ \mathit{focus}_L}{\Gamma; \langle A^+ \rangle, A^+ \multimap B^- \vdash B^-}\ \eta^-}{\Gamma; \langle A^+ \rangle, A^+ \multimap B^- \vdash [\downarrow B^-]}\ \downarrow_R}{\Gamma; \langle A^+ \rangle, A^+ \multimap B^- \vdash \downarrow B^-}\ \mathit{focus}_R}{\Gamma; A^+, A^+ \multimap B^- \vdash \downarrow B^-}\ \eta^+}{\Gamma; \Delta_A, A^+ \multimap B^- \vdash \downarrow B^-}\ \mathit{cut}(3) \qquad \dfrac{\Gamma; \Delta, B^- \vdash U}{\Gamma; \Delta, \downarrow B^- \vdash U}\ B^-}{\Gamma; \Delta_A, \Delta, A^+ \multimap B^- \vdash U}\ \mathit{cut}(3)}{\Gamma; \Delta_A, \Delta, \uparrow\downarrow \ldots \uparrow\downarrow (A^+ \multimap B^-) \vdash U}\ \uparrow\downarrow_L$$

Rule $\multimap_R$: We are given $\Gamma^\circ; \Delta^\circ, A \longrightarrow B$, which is used to derive $\Gamma^\circ; \Delta^\circ \longrightarrow A \multimap B$. We know $A \multimap B = (C^+)^\circ$; by induction on the structure of C^+ there exist A^+ and B^- such that $A = (A^+)^\circ$, $B = (B^-)^\circ$, and $C^+ = {\downarrow}{\uparrow} \ldots {\uparrow}{\downarrow}(A^+ \multimap B^-)$. By the induction hypothesis, we have $\Gamma; \Delta, {\uparrow}A^+ \vdash {\downarrow}B^+$, and we conclude by the unfocused admissibility lemma $\multimap_{uR}$:

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{\Gamma; \Delta, {\uparrow}A^+ \vdash {\downarrow}B^-}{\Gamma; \Delta, {\uparrow}A^+ \vdash {\uparrow}{\downarrow}B^-}\,{\uparrow_R}
}{\Gamma; \Delta, {\downarrow}{\uparrow}A^+ \vdash {\uparrow}{\downarrow}B^-}\,{\downarrow_L}
}{\Gamma; \Delta \vdash {\downarrow}{\uparrow}A^+ \multimap {\uparrow}{\downarrow}B^-}\,{\multimap_R}
\qquad
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{\dfrac{}{\Gamma; \langle A^+\rangle \vdash [A^+]}\,id^+}{\Gamma; \langle A^+\rangle \vdash A^+}\,focus_R
}{\Gamma; \langle A^+\rangle \vdash {\uparrow}A^+}\,{\uparrow_R}
}{\Gamma; \langle A^+\rangle \vdash [{\downarrow}{\uparrow}A^+]}\,{\downarrow_R}
\quad
\dfrac{
\dfrac{
\dfrac{\dfrac{}{\Gamma; [B^-] \vdash \langle B^-\rangle}\,id^-}{\Gamma; B^- \vdash \langle B^-\rangle}\,focus_L
}{\Gamma; {\downarrow}B^- \vdash \langle B^-\rangle}\,{\downarrow_L}
}{\Gamma; [{\uparrow}{\downarrow}B^-] \vdash \langle B^-\rangle}\,{\uparrow_L}
}{\Gamma; [{\downarrow}{\uparrow}A^+ \multimap {\uparrow}{\downarrow}B^-], \langle A^+\rangle \vdash \langle B^-\rangle}\,{\multimap_L}
}{\Gamma; {\downarrow}{\uparrow}A^+ \multimap {\uparrow}{\downarrow}B^-, \langle A^+\rangle \vdash \langle B^-\rangle}\,focus_L
}{\Gamma; {\downarrow}{\uparrow}A^+ \multimap {\uparrow}{\downarrow}B^-, \langle A^+\rangle \vdash B^-}\,\eta^-
}{\Gamma; {\downarrow}{\uparrow}A^+ \multimap {\uparrow}{\downarrow}B^-, A^+ \vdash B^-}\,\eta^+
}{\Gamma; {\downarrow}{\uparrow}A^+ \multimap {\uparrow}{\downarrow}B^- \vdash A^+ \multimap B^-}\,{\multimap_R}
}{\Gamma; {\downarrow}{\uparrow}A^+ \multimap {\uparrow}{\downarrow}B^- \vdash [{\downarrow}(A^+ \multimap B^-)]}\,{\downarrow_R}
}{\Gamma; {\downarrow}{\uparrow}A^+ \multimap {\uparrow}{\downarrow}B^- \vdash {\downarrow}(A^+ \multimap B^-)}\,focus_R
}{
\dfrac{\Gamma; \Delta \vdash {\downarrow}(A^+ \multimap B^-)}{\Gamma; \Delta \vdash {\downarrow}{\uparrow} \ldots {\uparrow}{\downarrow}(A^+ \multimap B^-)}\,{\downarrow}{\uparrow_R}
}\,cut(4)
$$

All the other cases follow the same pattern. □

2.3.8 Confluent versus fixed inversion

A salient feature of this presentation of focusing is that invertible, non-focused rules need not be applied in any particular order. Therefore, the last step in a proof of $\Gamma; \Delta, A \otimes B, \mathbf{1}, !C \vdash D \multimap E$ could be $\otimes_L$, $\mathbf{1}_L$ $!_L$, or $\multimap_R$. The style is exemplified by Liang and Miller's LJF [LM09], and the confluent presentation in this chapter is closely faithful to Pfenning's course notes on linear logic [Pfe12c].

Allowing for this inessential nondeterminism simplifies the presentation a bit, but it also gets in the way of effective proof search and canonical derivations if we do not address it in some way. The different possibilities for addressing this nondeterminism within an inversion phase echo the discussion of nondeterminism in LF from the beginning of the chapter. We can, as suggested in that discussion, declare that all proofs which differ only by the order of their invertible, non-focused rules be treated as equivalent. It is possible to establish that all possible inversion orderings will lead to the same set of stable sequents, which lets us know that all of these reorderings do not fundamentally change the structure of the rest of the proof. This property already seems to be necessary to prove *unfocused cut* as expressed by this admissible rule:

$$
\dfrac{\Gamma; \Delta \vdash A \qquad \Gamma; \Delta', A \vdash U}{\Gamma; \Delta', \Delta \vdash U}\,cut
$$

(where A is A^+ or A^- and Δ, Δ', and U contain no focus but may not be stable). If A is A^+, proving the admissibility of this rule involves permuting invertible rules in the second given

derivation, $\Gamma; \Delta', A^+ \vdash U$, until A^+ is the only unstable part of the second sequent, at which point part 3 of Theorem 2.4 applies. Similarly, if A is A^-, we must permute invertible rules in the first given derivation until A^- is the only unstable part of the first sequent, at which point part 4 of Theorem 2.4 applies.

By proving and using this more general cut property, it would be possible to prove a more general completeness theorem: if $\Gamma^\circ; \Delta^\circ \longrightarrow U^\circ$, then $\Gamma; \Delta \vdash U$ (Theorem 2.6 as stated also requires that Δ and U be stable). The cases of this new theorem corresponding to the unfocused rules $!_R$, $\multimap_R$, and $\otimes_L$, which required the use of doubly-shifted side derivations in our presentation, are trivial in this modified presentation. Unfortunately, the proof of unfocused cut, while simple, is tedious and long. Gentzen's original proof of cut admissibility [Gen35] and Pfenning's mechanization [Pfe00] both scale linearly with the number of connectives and rules in the logic; the proofs of identity expansion, cut admissibility, soundness of focusing, and completeness of focusing presented in this chapter do too. There is no known proof of the unfocused admissibly of the rule cut above that scales linearly in this way: all known proofs grow quadratically with the number of connectives and rules in the logic.

Once we equate all proofs that differ only on the order in which inference rules are applied within an inversion phase, we can pick some member of each equivalence class to serve as a canonical representative; this will suffice to solve the problems with proof search, as we can search for the canonical representatives of focused proofs rather than searching within the larger set of all focused proofs. The most common canonical representatives force invertible rules to decompose propositions in a depth-first ordering.

Then, reminiscent of the move from LF to Canonical LF, the logic itself can be restricted so that only the canonical representatives are admitted. The most convenient way of forcing a left-most, depth-first ordering is to isolate the invertible propositions (A^+ on the left and A^- on the right) in separate, ordered inversion contexts, and then to only work on the left-most proposition in the context. This is the way most focused logics are defined, including those by Andreoli, Chaudhuri, and myself in the next chapter. This style of presenting a focusing logic can be called a *fixed* presentation, as the inversion phase is fixed in a particular, though fundamentally arbitrary, shape.

The completeness of focusing for a fixed presentation of focusing is implied by the completeness of focusing for a confluent presentation of the same logic along with the appropriate confluence property for that logic, whereas the reverse is not true. In this sense, the confluent presentation allows us to prove a stronger theorem than the fixed presentation does, though the fixed presentation will be sufficient for our purposes here and in later chapters. We will not prove confluence in this chapter, though doing so is a straightforward exercise.

2.3.9 Running example

Figure 2.9 gives the result of taking our robot example, Figure 2.2, through the polarization process and then running the result through Theorem 2.6. There is indeed only one proof of this focused proposition up to the reordering of invertible rules, and only one proof period if we always decompose invertible propositions in a left-most (i.e., depth-first) ordering as we do in Figure 2.9.

$$
\begin{array}{c}
\dfrac{\dfrac{\dfrac{

a bipole is a monopole surrounded by positive (or, respectively, negative) propositions [And01]. In a polarized setting, bipoles capture the outermost structure of a proposition up to the second occurrence of a shift or an exponential.

The first idea behind synthetic inference rules is that the most important sequents in a polarized sequent calculus are stable sequents where all suspended propositions are atomic. This was reflected by our proof of the completeness of focusing (Theorem 2.6), which was restricted to stable sequents.[7] The second idea is that the bottom-most rule in the proof of a stable sequent must be one of the following:

* $copy$ on some proposition A^- from Γ,
* $focus_L$ on some proposition A^- in Δ, or
* $focus_R$ on the succedent A^+

Once we know which proposition we have focused on, the bipole structure of that proposition (that is, the outermost structure of the proposition up through the second occurrence of a shift of exponential) completely (though not uniquely) dictates the structure of the proof up to the next occurrences of stable sequents.

For example, consider the act of focusing on the proposition $a^+ \multimap \uparrow b^+$ in Γ using the $copy$ rule, where a^+ and b^+ are positive atomic propositions. This must mean that a suspended atomic proposition a^+ appears suspended in the context Δ, or else the proof could not be completed:

$$\dfrac{\dfrac{\dfrac{}{\Gamma, a^+ \multimap \uparrow b^+; \langle a^+ \rangle \vdash [a^+]}\, id^+ \quad \dfrac{\dfrac{\Gamma, a^+ \multimap \uparrow b^+; \Delta, \langle b^+ \rangle \vdash U}{\Gamma, a^+ \multimap \uparrow b^+; \Delta, b^+ \vdash U}\, \eta^+}{\Gamma, a^+ \multimap \uparrow b^+; \Delta, [\uparrow b^+] \vdash U}\, \uparrow_L}{\Gamma, a^+ \multimap \uparrow b^+; \Delta, \langle a^+ \rangle, [a^+ \multimap \uparrow b^+] \vdash U}\, \multimap_L}{\Gamma, a^+ \multimap \uparrow b^+; \Delta, \langle a^+ \rangle \vdash U}\, copy$$

The non-stable sequents in the middle are not interesting parts of the structure of the proof, as they are fully determined by the choice of focus, so we can collapse this series of transitions into a single synthetic rule:

$$\dfrac{\Gamma, a^+ \multimap \uparrow b^+; \Delta, \langle b^+ \rangle \vdash U}{\Gamma, a^+ \multimap \uparrow b^+; \Delta, \langle a^+ \rangle \vdash U}\, \mathsf{CP}$$

For the MELL fragment of linear logic, we can associate exactly one rule with every positive proposition (corresponding to a right-focus on that proposition) and two rules with every negative proposition (corresponding to left focus on a negative proposition in the persistent context and left focus on that negative proposition in the positive context). Here are three examples:

$$\dfrac{\Gamma; \Delta, \langle b^+ \rangle \vdash U}{\Gamma; \Delta, \langle a^+ \rangle, a^+ \multimap \uparrow b^+ \vdash U}\, \mathsf{LF}$$

$$\dfrac{\Gamma, A^-; \Delta, \langle b^+ \rangle, C^- \vdash D^+}{\Gamma; \Delta \vdash \downarrow(!A^- \otimes b^+ \otimes \downarrow C^- \multimap \uparrow D^+)}\, \mathsf{RF} \qquad \dfrac{}{\Gamma; \langle a^+ \rangle \vdash a^+}\, \mathsf{RF}'$$

[7] If we had established the unfocused cut rule discussed in Section 2.3.8 and had then proven the completeness of focusing (Theorem 2.6) for arbitrary inverting sequents, it would have enabled an interpretation that puts all unfocused sequents on similar footing, but that is not our goal here.

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{6bucks} \multimap \uparrow\mathsf{battery} \;;\; \langle\mathsf{robot}\rangle \vdash \mathsf{robot}}\,\mathsf{RF}'}{\mathsf{6bucks} \multimap \uparrow\mathsf{battery} \;;\; \mathsf{battery} \multimap \uparrow\mathsf{robot},\ \langle\mathsf{battery}\rangle \vdash \mathsf{robot}}\,\mathsf{LF}}{\mathsf{6bucks} \multimap \uparrow\mathsf{battery} \;;\; \langle\mathsf{6bucks}\rangle,\ \mathsf{battery} \multimap \uparrow\mathsf{robot} \vdash \mathsf{robot}}\,\mathsf{CP}}{\cdot;\cdot \vdash \downarrow(!(\mathsf{6bucks} \multimap \uparrow\mathsf{battery}) \otimes \mathsf{6bucks} \otimes \downarrow(\mathsf{battery} \multimap \uparrow\mathsf{robot}) \multimap \uparrow\mathsf{robot})}\,\mathsf{RF}
$$

Figure 2.10: Our running example, presented with synthetic rules

This doesn't mean that there are no choices to be made within focused phases, just that, in MELL, those choices are limited to the way the resources – propositions in Δ – are distributed among the branches of the proof. If we also consider additive connectives, we can identify some number of synthetic rules for each right focus, left focus, or copy. This may be zero, as there's no way to successfully right focus on a proposition like $\mathbf{0} \otimes \downarrow\uparrow A^+$, and therefore zero synthetic inference rules are associated with this proposition. It may be more than one: there are three ways to successfully right focus on the proposition $a^+ \oplus b^+ \oplus c^+$, and so three synthetic inference rules are associated with this proposition:

$$
\dfrac{}{\Gamma; \langle a^+\rangle \vdash a^+ \oplus b^+ \oplus c^+} \quad \dfrac{}{\Gamma; \langle b^+\rangle \vdash a^+ \oplus b^+ \oplus c^+} \quad \dfrac{}{\Gamma; \langle c^+\rangle \vdash a^+ \oplus b^+ \oplus c^+}
$$

Focused proofs of stable sequents are, by definition, in a 1-to-1 correspondence with proofs using synthetic inference rules. If we look at our running example as a derivation using the example synthetic inference rules presented above (as demonstrated in Figure 2.10), we see that the system takes four steps. The middle two steps, furthermore, correspond precisely to the two steps in our informal description of the robot-battery-store system.

2.5 Hacking the focusing system

Despite the novel treatment of suspended propositions in Section 2.3, the presentation of linear logic given there is equivalent to the presentation in Chaudhuri's dissertation [Cha06], in the sense that the logic gives rise to the same synthetic inference rules. It is *not* a faithful intuitionistic analogue to Andreoli's original presentation of focusing [And92], though the presentation in Pfenning's course notes is [Pfe12c].[8] Nor does it have the same synthetic inference rules as the focused presentation used in the framework of ordered logical specifications that we presented in [PS09].

In this section, we will discuss four different presentations of focused sequent calculi that are closely connected to the logic we have just presented. Each system differs significantly in its treatment of positive atomic propositions, the exponential $!A$, and the interaction between them.

- Andreoli's original system, which I name the *atom optimization*, complicates the interpretation of atomic propositions as stand-ins for arbitrary propositions.

[8] We will blur the lines, in this section, between Andreoli's original presentation of focused classical linear logic and Pfenning's adaptation to intuitionistic linear logic. In particular, we will mostly use the notation of Pfenning's presentation, but the observations are equally applicable in Andreoli's focused triadic system.

* A further change to the atom optimization, the *exponential optimization*, complicates the relationship between the focused logic and the unfocused logic.
* The *adjoint logic* of Benton and Wadler [BW96] introduces a new syntactic class of persistent propositions, restricting linear propositions to the linear context and persistent propositions to the persistent context.
* The introduction of *permeable atomic propositions*, a notion (which dates at least back to Girard's LU [Gir93]) that some propositions can be treated as *permeable* between the persistent and linear contexts and that permeable atomic propositions can be introduced to stand for this class of permeable propositions.

The reason we survey these different systems is that they all provide a solution to a pervasive problem encountered when using focused sequent calculi as logical frameworks: the need to allow for synthetic inference rules of the form

$$\frac{\Gamma, p; \Delta, r \vdash C}{\Gamma, p; \Delta, q \vdash C}$$

where p is an atomic proposition in the persistent context that is observed (but not consumed), q is an atomic proposition that is consumed in the transition, and r is an atomic proposition that is generated as the result of the transition. In the kinds of specifications we will be dealing with, the ability to form these synthetic inference rules is critical. In some uses, the persistent resource acts as *permission* to consume q and produce r. In other uses, p represents knowledge that we must currently possess in order to enact a transition. As a concrete example, America's 2010 health care reform law introduced a requirement that restaurant menus include calorie information. This means that, in the near future, we can exchange six bucks for a soup and salad at Panera, but only if we know how many calories are in the meal. The six bucks, soup, and salad remain ephemeral resources like q and r, but the calorie count is persistent. A calorie count is scientific knowledge, which is a resource that is not consumed by the transition.

My justification for presenting Chaudhuri's system as the canonical focusing system for linear logic in Section 2.3 is because it most easily facilitates reasoning about the focused sequent calculus *as a logic*. Internal soundness and completeness properties are established by the cut admissibility and identity expansion theorems (Theorems 2.4 and 2.3), and these theorems are conceptually prior to the soundness and completeness of the focused system relative to the unfocused system (Theorems 2.5 and 2.6). The various modifications we discuss in this section complicate the treatment of focused logics as independently justifiable sequent calculi for linear logic. I suggest in Section 2.5.4 that the last option, the incorporation of permeable atomic propositions, is the most pleasing mechanism for incorporating the structure we desire into a focused presentation of linear logic.

All of the options discussed in this section are compatible with a fifth option, discussed in Section 4.7.1, of avoiding positive propositions altogether and instead changing our view of stable sequents. The proposition $\downarrow a^- \multimap \downarrow b^- \multimap c^-$ is associated with this synthetic inference rule:

$$\frac{\Gamma; \Delta \vdash \langle a^- \rangle \quad \Gamma; \Delta' \vdash \langle b^- \rangle}{\Gamma; \Delta, \Delta', \downarrow a^- \multimap \downarrow b^- \multimap c^- \vdash \langle c^- \rangle}$$

If we can prove a general theorem that the sequent $\Gamma; \Delta \vdash \langle a^- \rangle$ can only be proven if $\Delta = a^-$ or if $\Delta = \cdot$ and $a^- \in \Gamma$, then a^- is a *pseudo-positive* atomic proposition. Proving the succedent $\langle a^- \rangle$ where a^- is pseudo-positive is functionally very similar to proving $[a^+]$ in focus for a positive atomic proposition. This gives us license to treat stable sequents that prove a pseudo-positive proposition not as a stable sequent that appears in synthetic inference rules but as an immediate subgoal that gets folded into the synthetic inference rule. If a^- is pseudo-positive, the persistent proposition $\downarrow a^- \multimap \downarrow b^- \multimap c^-$ can be associated with these two synthetic inference rules:

$$\frac{\Gamma; \Delta \vdash \langle b^- \rangle}{\Gamma; \Delta, \downarrow a^- \multimap \downarrow b^- \multimap c^-, a^- \vdash \langle c^- \rangle} \qquad \frac{\Gamma, a^-; \Delta \vdash \langle b^- \rangle}{\Gamma, a^-; \Delta, \downarrow a^- \multimap \downarrow b^- \multimap c^- \vdash \langle c^- \rangle}$$

The machinery of lax logic introduced in Chapter 3 and the fragment of this logic that forms a logical framework in Chapter 4 make it feasible, in practice, to observe when negative atomic propositions are pseudo-positive.

2.5.1 Atom optimization

Andreoli's original focused system isn't polarized, so propositions that are syntactically invalid in a polarized presentation, like $!(p^+ \otimes q^+)$ or $!p^+$, are valid in his system (we would have to write $!\uparrow(p^+ \otimes q^+)$ and $!\uparrow p^+$). It's therefore possible, in an unpolarized presentation, to use the $copy$ rule to copy a positive proposition out of the context and into left focus, but the focus immediately blurs, as in this (intuitionistic) proof fragment:[9]

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ p^+ \otimes q^+; p^+, q^+ \Vdash q^+ \otimes p^+\end{array}}{p^+ \otimes q^+; p^+ \otimes q^+ \Vdash q^+ \otimes p^+}\,\otimes_L}{p^+ \otimes q^+; [p^+ \otimes q^+] \Vdash q^+ \otimes p^+}\,blur_L}{p^+ \otimes q^+; \cdot \Vdash q^+ \otimes p^+}\,copy$$

Note that, in the polarized setting, the effect of the $blur_L$ rule is accomplished by the $\downarrow_L$ rule.

Andreoli's system makes a single restriction to the $copy$ rule: it cannot apply to a positive atomic proposition in the persistent context. On its own, this restriction would make the system incomplete with respect to unfocused linear logic – there would be no focused proof of $!p^+ \multimap p^+$ – and so Andreoli-style focusing systems restore completeness by creating a second initial sequent for positive atomic propositions that allows a positive right focus on an atomic proposition to succeed if the atomic proposition appears in the persistent context:

$$\frac{}{\Gamma; p^+ \Vdash [p^+]}\,id_1^+ \qquad \frac{}{\Gamma, p^+; \cdot \Vdash [p^+]}\,id_2^+$$

With the second initial rule, we can once again prove $!p^+ \multimap p^+$, and the system becomes

[9] We will use the sequent form $\Gamma; \Delta \Vdash C$ in this section for focused but unpolarized systems. Again, we frequently reference Pfenning's presentation of focused linear logic [Pfe12c] as a faithful intuitionistic analogue of Andreoli's system.

complete with respect to unfocused linear logic again.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p^+; \cdot \Vdash [p^+]}\ id_2^+}{p^+; \cdot \Vdash p^+}\ focus_R}{\cdot; !p^+ \Vdash p^+}\ !_L}{\cdot; \cdot \Vdash !p^+ \multimap p^+}\ \multimap_R$$

This modified treatment of positive atoms will be called the *atom optimization*, as it reduces the number of focusing steps that need to be applied: it takes only one right focus to prove $!p^+ \multimap p^+$ in Andreoli's system, but it would take two focusing steps to prove the same proposition in Chaudhuri's system (or to prove $!{\uparrow}p^+ \multimap {\uparrow}p^+$ in the focusing system we have presented).

There seem to be three ways of adapting the atom optimization to a polarized setting. The first approach is to add an initial sequent that directly mimics the one in Andreoli's system, while adding an additional requirement to the *copy* rule that A^- is not a shifted positive atomic proposition:

$$\dfrac{}{\Gamma; \langle A^+ \rangle \vdash [A^+]}\ id^+ \quad \dfrac{}{\Gamma, {\uparrow}p^+; \cdot \vdash [p^+]}\ id_2^+ \quad \dfrac{A \neq {\uparrow}p^+ \quad \Gamma, A^-; \Delta, [A^-] \vdash U}{\Gamma, A^-; \Delta \vdash U}\ copy^*$$

The second approach is to extend suspended propositions to the persistent context, add a corresponding rule for right focus, and modify the left rule for ! to notice the presence of a positive atomic proposition:

$$\dfrac{A^- \neq {\uparrow}p^+ \quad \Gamma, A^-; \Delta \vdash U}{\Gamma; \Delta, !A^- \vdash U}\ !_{L1} \quad \dfrac{\Gamma, \langle p^+ \rangle; \Delta \vdash U}{\Gamma; \Delta, !{\uparrow}p^+ \vdash U}\ !_{L2}$$

$$\dfrac{}{\Gamma; \langle A^+ \rangle \vdash [A^+]}\ id_1^+ \quad \dfrac{}{\Gamma, \langle A^+ \rangle; \cdot \vdash [A^+]}\ id_2^+$$

The third approach is to introduce a new connective, $\Uparrow$, that can only be applied to positive atomic propositions, just as ! can only be applied to negative propositions. We can initially view this option as equivalent to the previous one by defining $\Uparrow p^+$ as a notational abbreviation for $!{\uparrow}p^+$ and styling rules according to the second approach above:

$$\dfrac{\Gamma; \cdot \vdash p^+}{\Gamma; \cdot \vdash [\Uparrow p^+]}\ \Uparrow_R \quad \dfrac{\Gamma, \langle p^+ \rangle; \Delta \vdash U}{\Gamma; \Delta, \Uparrow p^+ \vdash U}\ \Uparrow_L \quad \dfrac{}{\Gamma; \langle A^+ \rangle \vdash [A^+]}\ id_1^+ \quad \dfrac{}{\Gamma, \langle A^+ \rangle; \cdot \vdash [A^+]}\ id_2^+$$

All three of these options are similar; we will go with the last, as it allows us to preserve the original meaning of $!{\uparrow}p^+$ if that is our actual intent. Introducing the atom optimization as a new connective also allows us to isolate the effects that this new connective has on cut admissibility, identity expansion, and the correctness of focusing; we will consider each in turn.

$$
\begin{array}{c}
\dfrac{\dfrac{\dfrac{\dfrac{}{\langle p^+\rangle; \cdot \vdash [p^+]}\,id_2^+ \quad \dfrac{}{\langle p^+\rangle; \cdot \vdash [p^+]}\,id_2^+}{\langle p^+\rangle; \cdot \vdash [p^+ \otimes p^+]}\,\otimes_R}{\langle p^+\rangle; \cdot \vdash p^+ \otimes p^+}\,focus_R}{\cdot; \Uparrow p^+ \vdash p^+ \otimes p^+}\,\Uparrow_L
\end{array}
\quad \text{vs.} \quad
\begin{array}{c}
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\uparrow A^+; \langle A^+\rangle \vdash [A^+]}\,id_1^+ \quad \dfrac{}{\uparrow A^+; \langle A^+\rangle \vdash [A^+]}\,id_1^+}{\uparrow A^+; \langle A^+\rangle, \langle A^+\rangle \vdash [A^+ \otimes A^+]}\,\otimes_R}{\uparrow A^+; \langle A^+\rangle, \langle A^+\rangle \vdash A^+ \otimes A^+}\,focus_R}{\uparrow A^+; \langle A^+\rangle, A^+ \vdash A^+ \otimes A^+}\,\eta^+}{\uparrow A^+; \langle A^+\rangle, [\uparrow A^+] \vdash A^+ \otimes A^+}\,\uparrow_L}{\uparrow A^+; \langle A^+\rangle \vdash A^+ \otimes A^+}\,copy}{\uparrow A^+; A^+ \vdash A^+ \otimes A^+}\,\eta^+}{\uparrow A^+; [\uparrow A^+] \vdash A^+ \otimes A^+}\,\uparrow_L}{\uparrow A^+; \cdot \vdash A^+ \otimes A^+}\,copy}{\cdot; !\uparrow A^+ \vdash A^+ \otimes A^+}\,!_L
\end{array}
$$

Figure 2.11: Substituting A^+ for p^+ in the presence of the atom optimization

Identity expansion There is one new case of identity expansion, which is unproblematic:

$$
\begin{array}{c}
\mathcal{D} \\
\Gamma; \Delta, \langle \Uparrow p^+\rangle \vdash U \\
\hline
\Gamma; \Delta, \Uparrow p^+ \vdash U
\end{array}\,\eta^+
\quad \Longrightarrow \quad
\begin{array}{c}
\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, \langle p^+\rangle; \cdot \vdash [p^+]}\,id_2^+}{\Gamma, \langle p^+\rangle; \cdot \vdash p^+}\,focus_R}{\Gamma, \langle p^+\rangle; \cdot \vdash [\Uparrow p^+]}\,\Uparrow_R \quad \begin{array}{c}\mathcal{D}\\ \Gamma; \Delta, \langle \Uparrow p^+\rangle \vdash U\end{array}}{\Gamma, \langle p^+\rangle; \Delta, \langle \Uparrow p^+\rangle \vdash U}\,weaken}{\Gamma, \langle p^+\rangle; \Delta \vdash U}\,subst^+}{\Gamma; \Delta, \Uparrow p^+ \vdash U}\,\Uparrow_L
\end{array}
$$

Even though the identity expansion theorem is unproblematic, we can illuminate one problem with the atom optimization by considering the substitution of arbitrary propositions for atomic propositions. Previously, when we substituted a positive proposition for an atomic proposition, the proof's structure remained fundamentally unchanged – instances of the η^+ rule on p^+ turned into admissible instances of the general identity expansion rule η^+ on A^+. Now, we have to explain what it even means to substitute A^+ for p^+ in $\Uparrow p^+$, since $\Uparrow A^+$ is not a syntactically valid proposition; the only obvious candidate seems to be $!\uparrow A^+$. That substitution may require us to change the structure of proofs in a significant way, as shown in Figure 2.11. Immediately before entering into any focusing phase where the id_2^+ rule is used n times on the hypothesis $\langle p^+\rangle$, we need to left-focus on $\uparrow A^+$ n times with the *copy* rule to get n copies of $\langle A^+\rangle$ into the linear context, each of which can be used to replace one of the id_2^+ instances with an instance of id_1^+.

Cut admissibility While we might be willing to sacrifice the straightforward interpretation of atomic propositions as stand-ins for arbitrary propositions, another instance of the same problematic pattern arises when we try to establish the critical cut admissibility theorem for the logic with $\Uparrow p^+$. Most of the new cases are unproblematic, but trouble arises in part 1 when we cut a

$$
\dfrac{\dfrac{\dfrac{}{\Gamma;\cdot\vdash[q^+]}\,id_2^+ \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma;\langle p^+\rangle\vdash[p^+]}\,id_1^+}{\Gamma;\langle p^+\rangle\vdash p^+}\,focus_R}{\Gamma;p^+\vdash p^+}\,\eta^+}{\Gamma;[\uparrow p^+]\vdash p^+}\,\uparrow_L}{\Gamma;q^+\multimap\uparrow p^+\vdash p^+}\,\multimap_L}{\Gamma;\cdot\vdash p^+}\,copy \qquad \dfrac{\dfrac{\dfrac{}{\Gamma,\langle p^+\rangle;\cdot\vdash[p^+]}\,id_2^+ \qquad \dfrac{}{\Gamma,\langle p^+\rangle;\cdot\vdash[p^+]}\,id_2^+}{\Gamma,\langle p^+\rangle;\cdot\vdash[p^+\otimes p^+]}\,\otimes_R}{\Gamma,\langle p^+\rangle;\cdot\vdash p^+\otimes p^+}\,focus_R}{\Gamma;\cdot\vdash p^+\otimes p^+}\,cut(6b)
$$

$$\Longrightarrow$$

$$
\dfrac{\begin{array}{c}\vdots\\ \Gamma;\cdot\vdash p^+\end{array} \qquad \dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma;\cdot\vdash p^+\end{array} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma;\langle p^+\rangle\vdash[p^+]}\,id_1^+ \qquad \dfrac{}{\Gamma;\langle p^+\rangle\vdash[p^+]}\,id_1^+}{\Gamma;\langle p^+\rangle,\langle p^+\rangle\vdash[p^+\otimes p^+]}\,\otimes_R}{\Gamma;\langle p^+\rangle,\langle p^+\rangle\vdash p^+\otimes p^+}\,focus_R}{\Gamma;\langle p^+\rangle,p^+\vdash p^+\otimes p^+}\,\eta^+}{\Gamma;\langle p^+\rangle\vdash p^+\otimes p^+}\,cut(3)}{\Gamma;p^+\vdash p^+\otimes p^+}\,\eta^+}{\Gamma;\cdot\vdash p^+\otimes p^+}\,cut(3)
$$

(where $\Gamma = q^+ \multimap \uparrow p^+, \langle q^+\rangle$)

Figure 2.12: A problematic cut that arises from the introduction of the $\overset{\uparrow}{\cdot}p^+$ connective

right-focused proof of $\overset{\uparrow}{\cdot}p^+$ against a proof that is decomposing $\overset{\uparrow}{\cdot}p^+$ on the left:

$$
\dfrac{\dfrac{\Gamma;\cdot\vdash p^+}{\Gamma;\cdot\vdash[\overset{\uparrow}{\cdot}p^+]}\,\overset{\uparrow}{\cdot}_R \qquad \dfrac{\Gamma,\langle p^+\rangle;\Delta\vdash U}{\Gamma;\Delta,\overset{\uparrow}{\cdot}p^+\vdash U}\,\overset{\uparrow}{\cdot}_L}{\Gamma;\Delta\vdash U}\,cut(1)
$$

We are left needing to prove that $\Gamma;\cdot\vdash p^+$ and $\Gamma,\langle p^+\rangle;\Delta\vdash U$ proves $\Gamma;\Delta\vdash U$, which does not fit the structure of any of our existing cut principles. It is similar to the statement of part 5 of Theorem 2.4 (if $\Gamma;\cdot\vdash A^-$ and $\Gamma,A^-;\underline{\Delta}\vdash\underline{U}$, then $\Gamma;\underline{\Delta}\vdash\underline{U}$), but the proof is not so straightforward.

To see why this cut is more complicated to prove than part 5 of Theorem 2.4, consider what it will take to reduce the cut in the top half of Figure 2.12. We cannot immediately call the induction hypothesis on the sub-derivation in the right branch, as there is no way to prove $p^+\otimes p^+$ in focus when $\langle p^+\rangle$ does not appear (twice) in the linear context. We need to get two suspended $\langle p^+\rangle$ antecedents in the linear context; then we can replace all the instances of id_2^+ with instances of id_1^+ that use freshly-minted $\langle p^+\rangle$ antecedents. This can be achieved with repeated application of part 3 of Theorem 2.4, as shown in the bottom half of Figure 2.12.

The minimal extension to cut admissibility (Theorem 2.4) that justifies the atom optimization appears to be the following, where $\langle p^+\rangle^n$ denotes n copies of the suspended positive proposition $\langle p^+\rangle$.

Theorem 2.7 (Extra cases of cut admissibility (Theorem 2.4))**.**

6a. If $\Gamma; \cdot \vdash p^+$ *and* $\Gamma, \langle p^+ \rangle; \Delta \vdash [B^+]$*, then there exists* n *such that* $\Gamma; \Delta, \langle p^+ \rangle^n \vdash [B^+]$.

6b. If $\Gamma; \cdot \vdash p^+$ *and* $\Gamma, \langle p^+ \rangle; \Delta \vdash U$*, then* $\Gamma; \Delta \vdash U$.

6c. If $\Gamma; \cdot \vdash p^+$ *and* $\Gamma, \langle p^+ \rangle; \Delta, [B^-] \vdash U$*, then there exists* n *such that* $\Gamma; \Delta, \langle p^+ \rangle^n, [B^-] \vdash U$.

Proof. Induction on the second given derivation; whenever $focus_R$, $focus_L$ or $copy$ are the last rule in part $6b$, we need to make n calls to part 3 of the cut admissibility lemma, each one followed by a use of the η^+ rule, where n is determined by the inductive call to part $6a$ (for $focus_R$) or $6c$ (for $focus_L$ and $copy$).

The calls to part 3 are justified by the existing induction metric: the principal cut formula p^+ stays the same and the part number gets smaller. □

Correctness of focusing The obvious way of extending erasure for our extended logic is to let $(\Uparrow p^+)^\circ = !p^+$ and to let $(\Gamma, \langle p^+ \rangle)^\circ = (\Gamma)^\circ, p^+$. Under this interpretation, the soundness of $\Uparrow_L$ and $\Uparrow_R$ has the same structure as the soundness of $!_L$ and $!_R$, and the soundness of id_2^+ in the focused system is established with $copy$ and id in the unfocused system:

$$\dfrac{\overline{\Gamma^\circ, p^+; p^+ \longrightarrow p^+}\ id}{\Gamma^\circ, p^+; \cdot \longrightarrow p^+}\ copy$$

The extension to the proof of completeness requires two additional cases to deal with $\Uparrow$, both of which are derivable...

$$\dfrac{\Gamma; \cdot \vdash p^+}{\Gamma; \cdot \vdash \downarrow\uparrow \ldots \downarrow\uparrow\Uparrow p^+}\ \Uparrow_{uR} \qquad \dfrac{\Gamma, \langle p^+ \rangle; \Delta \vdash U}{\Gamma; \Delta, \uparrow\downarrow \ldots \downarrow\uparrow\Uparrow p^+ \vdash U}\ \Uparrow_{uL}$$

...as well as a case dealing with the situation where we apply $copy$ to the erasure of a persistent suspended proposition. This case reduces to a case of ordinary focal substitution:

$$\dfrac{\Gamma, \langle p^+ \rangle; \Delta, \langle p^+ \rangle \vdash U}{\Gamma, \langle p^+ \rangle; \Delta \vdash U}\ \langle copy \rangle_u \quad = \quad \dfrac{\overline{\Gamma, \langle p^+ \rangle; \cdot \vdash [p^+]}\ id_2^+ \qquad \Gamma, \langle p^+ \rangle; \Delta, \langle p^+ \rangle \vdash U}{\Gamma, \langle p^+ \rangle; \Delta \vdash U}\ subst^+$$

For such a seemingly simple change, the atom optimization adds a surprising amount of complexity to the cut admissibility theorem for focused linear logic. What's more, the three extra cases of cut that we had to introduce were all for the purpose of handling a single problematic case in the proof of part 1 where both derivations were decomposing the principal cut formula $\Uparrow p^+$.

2.5.2 Exponential optimization

The choice of adding $\Uparrow p^+$ as a special new connective instead of defining it as $!\uparrow p^+$ paves the way for us to modify its meaning further. For instance, there turns out to be no internal reason for the $\Uparrow_R$ rule to lose focus in its premise, even though it is critical that $!_R$ lose focus on its premise;

if we fail to do so propositions like $!(p^+ \otimes q^+) \multimap !(q^+ \otimes p^+)$ will have no proof. We can revise $\Uparrow_R$ accordingly.

$$\frac{\Gamma; \cdot \vdash [p^+]}{\Gamma; \cdot \vdash [\Uparrow p^+]}\ \Uparrow_R \qquad \frac{\Gamma, \langle p^+ \rangle; \Delta \vdash U}{\Gamma; \Delta, \Uparrow p^+ \vdash U}\ \Uparrow_L \qquad \frac{}{\Gamma; \langle A^+ \rangle \vdash [A^+]}\ id_1^+ \qquad \frac{}{\Gamma, \langle A^+ \rangle; \cdot \vdash [A^+]}\ id_2^+$$

This further optimization can be called the *exponential optimization*, as it, like the atom optimization, potentially reduces the number of focusing phases in a proof. Identity expansion is trivial to modify, and cut admissibility is significantly simpler.

The problematic case of cut is easy to handle in this modified system: we can conclude by case analysis that the first given derivation must prove p^+ in focus using the id_2^+ rule. This, in turn, means that $\langle p^+ \rangle$ must already appear in Γ, so $\Gamma = \Gamma', \langle p^+ \rangle$, and the cut reduces to an admissible instance of contraction.

$$\dfrac{\dfrac{\dfrac{}{\Gamma', \langle p^+ \rangle; \cdot \vdash [p^+]}\ id_2^+}{\Gamma', \langle p^+ \rangle; \cdot \vdash [\Uparrow p^+]}\ \Uparrow_R \qquad \dfrac{\Gamma', \langle p^+ \rangle, \langle p^+ \rangle; \Delta \vdash U}{\Gamma', \langle p^+ \rangle; \Delta, \Uparrow p^+ \vdash U}\ \Uparrow_L}{\Gamma', \langle p^+ \rangle; \Delta \vdash U}\ cut(1) \quad \Longrightarrow \quad \dfrac{\Gamma', \langle p^+ \rangle, \langle p^+ \rangle; \Delta \vdash U}{\Gamma', \langle p^+ \rangle; \Delta \vdash U}\ contract$$

Thus, we no longer need the complicated extra parts $6a$ - $6c$ of cut admissibility in order to prove cut admissibility for a focused system with the exponential optimization.

Because cut and identity hold, we can think of a focused logic with the exponential optimization as being internally sensible. The problem is that this logic is no longer *externally* sensible relative to normal linear logic, because we cannot erase $\Uparrow p^+$ into regular linear logic in a sensible way. Specifically, if we continue to define $(\Uparrow p^+)^\circ$ as $!p^+$, then $\Uparrow q^+ \multimap !(q^+ \multimap \uparrow p^+) \multimap \uparrow\Uparrow p^+$ has no proof in focused linear logic, whereas its erasure, $!q^+ \multimap !(q^+ \multimap p^+) \multimap !p^+$, does have an unfocused proof. In other words, the completeness of focusing (Theorem 2.6) no longer holds under the exponential optimization!

Our focused logic with the exponential optimization has some resemblance to tensor logic [MT10], as well the polarized logic that Girard presented in a note, "On the sex of angels," which first introduced the $\uparrow A^+$ and $\downarrow A^-$ notation to the discussion of polarity [Gir91]. Both of these presentations incorporate a general focus-preserving $\Uparrow A^+$ connective – a positive formula with a positive subformula – in lieu of the focus-interrupting $!A^-$ connective. Both presentations also have the prominent caveat that ! in the unfocused logic necessarily corresponds to $\Uparrow\downarrow$ in the focused logic: it is *not* possible to derive $\Uparrow(A \otimes B) \vdash \Uparrow(B \otimes A)$ in these systems, and no apology is made for this fact, because $\Uparrow\downarrow\uparrow(A \otimes B) \vdash \Uparrow\downarrow\uparrow(B \otimes A)$ holds as expected. We want avoid this route because it gives the shifts too much power: they influence the *existence* of proofs, not just the structure of proofs.[10] This interpretation of shifts therefore threatens our critical ability to intuitively understand and explain linear logic connectives as resources.

There is an easily identifiable class of sequents that obey *separation*, which is the property that positive atomic propositions can be separated into two classes p_l^+ and p_p^+. The *linear* positive

[10]Both the note of Girard and the paper of Melliès and Tabareau see the shifts as a form of negation; therefore, writing from an intuitionistic perspective, they are unconcerned that A^+ has a different meaning of $\downarrow\uparrow A^+$ in their constructive logic. There are many propositions where $\neg\neg A$ is provable even though A is not! This view of shifts as negations seems rather foreign to the erasure-based understanding of shifts we have been discussing, though Zeilberger has attempted to reconcile these viewpoints [Zei08b].

propositions p_l^+ are never suspended in the persistent context and never appear as $\Uparrow p_l^+$, whereas the *persistent* positive propositions p_p^+ are never suspended in the linear context and always appear as $\Uparrow p_p^+$ inside of other propositions. For sequents and formulas obeying separation, we can use the obvious erasure operation and obtain a proof of the completeness of focusing; this notion of separation was the basis of our completeness result in [PS09]. However, separation is a meta-logical property, something that we observe about a fragment of the logic and not an inherent property of the logic itself. There are many propositions A^+ and A^- that we cannot prove in focused linear logic with the exponential optimization even though $(A^+)^\circ$ and $(A^-)^\circ$ are provable in linear logic, and that makes the exponential optimization unsatisfactory.

The remaining two approaches, adjoint logic and the introduction of permeable atomic propositions, can both be seen as attempts to turn separation into a logical property instead of a meta-logical property.

2.5.3 Adjoint logic

We introduced $\Uparrow p^+$ as a connective defined as $!\uparrow p^+$ – that is, the regular $!A^-$ connective plus a little something extra, the shift. After our experience with modifying the rules of $\Uparrow$, we can motivate adjoint logic by trying to view $\Uparrow$ as a more primitive connective – that is, we will try to view $!$ as $\Uparrow$ plus a little something extra.

It is frequently observed that the exponential $!A$ of linear logic appears to have two or more parts; the general idea is that $\Uparrow$ represents just one of those pieces. Accounts of linear logic that follow the judgmental methodology of Martin-Löf [ML96], such as the analysis by Chang et al. [CCP03], emphasize that the regular hypothetical sequent $\Gamma; \Delta \longrightarrow A$ of linear logic is establishing the judgment that A is *true*: we can write $\Gamma; \Delta \longrightarrow A\ \mathit{true}$ to emphasize this. The judgment of *validity*, represented by the judgment $A\ \mathit{valid}$, is defined as truth using no ephemeral resources, and $!A$ is understood as the internalization of judgmental validity:

$$\frac{\Gamma; \cdot \longrightarrow A\ \mathit{true}}{\Gamma \longrightarrow A\ \mathit{valid}}\ \mathit{valid} \qquad \frac{\Delta = \cdot \qquad \Gamma \longrightarrow A\ \mathit{valid}}{\Gamma; \Delta \longrightarrow !A\ \mathit{true}}\ !'_R$$

The valid rule is invertible, so if we ever need to prove $\Gamma \longrightarrow A\ \mathit{valid}$, we may asynchronously transition to proving $\Gamma; \cdot \longrightarrow A\ \mathit{true}$. This observation is used to explain why we don't normally consider validity on the right in linear logic. Our more familiar rule for $!_R$ is derivable using these two rules:

$$\frac{\Delta = \cdot \qquad \dfrac{\Gamma; \cdot \vdash A\ \mathit{true}}{\Gamma \vdash A\ \mathit{valid}}\ \mathit{valid}}{\Gamma; \Delta \vdash !A\ \mathit{true}}\ !'_R$$

Note that the $!'_R$ rule is not invertible, because it forces the linear context to be empty, which means $!$ must be positive. The valid rule, on the other hand, is invertible and has an asynchronous or negative character, because it represents the invertible step of deciding to prove that A is *valid* (true without recourse to any ephemeral resources) by proving that it is *true* (in a context with no ephemeral resources). This combination of positive and negative actions explains why $!A^-$ is a positive proposition with a negative subformula, and similarly explains why we must break focus when we reach $!A$ on the right and why we must stop decomposing the proposition when

$$\frac{\Gamma; \cdot \longrightarrow A}{\Gamma \longrightarrow GA}\, G_R \qquad \frac{\Gamma, GA; \Delta, A \longrightarrow C}{\Gamma, GA; \Delta \longrightarrow C}\, G_L \qquad \frac{}{\Gamma, x \longrightarrow x}\, \mathit{init}_x$$

$$\frac{\Gamma, X \longrightarrow Y}{\Gamma \longrightarrow X \supset Y}\, \supset_R \qquad \frac{\Gamma, X \supset Y \longrightarrow X \quad \Gamma, X \supset Y, Y \longrightarrow Z}{\Gamma, X \supset Y \longrightarrow Z}\, \supset_L$$

$$\frac{\Gamma, X \supset Y \longrightarrow X \quad \Gamma, X \supset Y, Y; \Delta \longrightarrow C}{\Gamma, X \supset Y; \Delta \longrightarrow C}\, \supset'_L$$

$$\frac{\Gamma \longrightarrow X}{\Gamma; \cdot \longrightarrow FX}\, F_R \qquad \frac{\Gamma, X; \Delta \longrightarrow C}{\Gamma; \Delta, FX \longrightarrow C}\, F_L \qquad \frac{}{\Gamma; a \longrightarrow a}\, \mathit{init}_a$$

$$\frac{\Gamma; \Delta, A \longrightarrow B}{\Gamma; \Delta \longrightarrow A \multimap B}\, \multimap_R \qquad \frac{\Gamma; \Delta_A \longrightarrow A \quad \Gamma; \Delta, B \longrightarrow C}{\Gamma; \Delta_A, \Delta, A \multimap B \longrightarrow C}\, \multimap_L$$

Figure 2.13: Some relevant sequent calculus rules for adjoint logic

we reach $!A$ on the left. The salient feature of the exponential optimization's rules for $\Uparrow p^+$, of course, is that they do *not* break focus on the right and that they continue to decompose the proposition on the left (into a suspended proposition $\langle p^+ \rangle$ in the persistent context). This is the reason for arguing that $\Uparrow$ captures only the first, purely positive, component of the ! connective.

If the $\Uparrow$ connective is the first part of the ! connective, can we characterize the rest of the connective? Giving a reasonable answer necessarily requires a more general account of the $\Uparrow$ connective – an *unfocused* logic where it is generally applicable rather than restricted to positive atomic propositions. In other words, to account for the behavior of $\Uparrow$, we must give a more primitive logic into which linear logic can be faithfully encoded.

A candidate for a more primitive logic, and one that has tacitly formed the basis of much of my previous work on logic programming and logical specification in substructural logic [PS09, SP11b, SP11a], is *adjoint logic*. Adjoint logic was first characterized by Benton and Wadler as a natural deduction system [BW96] and was substantially generalized by Reed in a sequent calculus setting [Ree09b]. The logic generalizes both linear logic and Fairtlough and Mendler's lax logic [FM97] as sub-languages of a common logic, whose propositions come in two syntactically distinct categories that are connected by the adjoint operators F and G:

Persistent propositions	$X, Y, Z ::= GA \mid x \mid X \supset Y \mid X \times Y$
Linear propositions	$A, B, C ::= FX \mid a \mid A \multimap B \mid A \otimes B$

In adjoint logic, persistent propositions X appear in the persistent context Γ and as the succedents of sequents $\Gamma \longrightarrow X$, whereas linear propositions A appear in the linear context Δ and as the succedents of sequents $\Gamma; \Delta \longrightarrow A$. Going back to our previous discussion, this means that persistent propositions are only ever judged to be valid, and that linear propositions are only ever judged to be true. A fragment of the logic is shown in Figure 2.13. Note the similarity between the G_L rule and our unfocused *copy* rule, as well as the similarity between F_R and G_R

in Figure 2.13 and the rules $!_R$ and $valid$ in the previous discussion. Linear logic is recovered as a fragment of adjoint logic by removing all of the persistent propositions except for GA; the usual $!A$ is then definable as FGA.[11]

One drawback of this approach is simply the logistics of giving a fully focused presentation of adjoint logic. We end up with a proliferation of propositions, because the syntactic distinction between X and A is orthogonal to the syntactic distinction between positive and negative propositions. A polarized presentation of adjoint logic would have four syntactic categories: X^+, X^-, A^+, and A^-, with one pair of shifts mediating between X^+ and X^- and another pair of shifts mediating between A^+ and A^-.[12] Given a focused presentation of adjoint logic, however, the separation criteria discussed above can be in terms of the two forms of positive atomic proposition a and x. Positive atomic propositions that are always associated with $\Uparrow$ can be encoded as persistent positive atomic propositions x^+, whereas positive atomic propositions that are never associated with $\Uparrow$ can be encoded as linear positive atomic propositions a^+. The proposition $\Uparrow p^+$ can then be translated as Fx^+, where x^+ is the translation of p^+ as a persistent positive atomic proposition.

Adjoint logic gives one answer to why, in Andreoli-style presentations of linear logic, we can't easily substitute positive propositions for positive atomic propositions when those positive atomic propositions appear suspended in the persistent linear context: because these propositions are actually stand-ins for *persistent* propositions, not for linear propositions, and we are working in a fragment of the logic that has no interesting persistent propositions other than atomic propositions x and the negative inclusion GA back into linear propositions. This effectively captures the structure of the separation requirement (as defined at the end of Section 2.5.2 above) in a logical way, but it makes the structure of persistent atomic propositions rather barren and degenerate, and it places an extra logic, adjoint logic, between the focused system and our original presentation of intuitionistic linear logic.

2.5.4 Permeability

Let us review the problems with our previous attempts to motivate a satisfactory treatment of positive propositions in the persistent context. Andreoli's atom optimization interferes with the structure of cut admissibility. The exponential optimization lacks a good interpretation in unfocused linear logic. The adjoint formulation of linear logic introduces persistent positive propositions as members of a syntactic class X of persistent propositions, a syntactic class that usually lies hidden in between the two right-synchronous and right-asynchronous (that is, positive and negative) halves of the ! connective. This approach works but requires a lot of extra machinery.

Our final attempt to logically motivate a notion of a persistent positive proposition will be based on an analysis of *permeability*. Permeability in classical presentations of linear logic dates back to Girard's LU [Gir93]. In this section, we will motivate permeable atomic propositions in

[11] Lax logic, on the other hand, is recovered by removing all of the linear propositions except for FX; the distinguishing connective of lax logic, $\bigcirc X$, is then definable as GFX.

[12] To make matters worse, in Levy's Call-By-Push-Value language, the programming language formalism that corresponds to polarized logic, $\uparrow$ and $\downarrow$ are characterized as adjoints as well (F and U, respectively), so a fully polarized adjoint logic has *three* distinct pairs of unary connectives that can be characterized as adjoints!

intuitionistic linear logic by first considering a new identity expansion principle that only applies to permeable propositions, a syntactic refinement of the positive propositions.[13]

The admissible identity expansion rules, like the admissible identity rule present in most unfocused sequent calculus systems, help us write down compact proofs. If $F(n) = p_1^+ \otimes \ldots \otimes p_n^+$, then the number of steps in the smallest proof of $\Gamma; F(n) \vdash F(n)$ is in $\Omega(n)$. However, by using the admissible identity expansion rule η^+, we can represent the proof in a compact way:

$$\dfrac{\dfrac{\overline{\Gamma; \langle F(n) \rangle \vdash [F(n)]}\ id^+}{\Gamma; \langle F(n) \rangle \vdash F(n)}\ focus_R}{\Gamma; F(n) \vdash F(n)}\ \eta^+$$

Permeability as a property of identity expansion

The pattern we want to capture with our new version of identity expansion is the situation where we are trying to prove a sequent like $\Gamma; \Delta \vdash \mathbf{1}$ or $\Gamma; \Delta \vdash !A^-$ and we know, by the syntactic structure of Δ, that inversion will empty the linear context. One instance of this pattern is the sequent $\Gamma; G(n) \vdash !{\uparrow}G(n)$ where $G(n) = !p_1^+ \otimes \ldots \otimes !p_n^+$. Our goal will be to prove such a sequent succinctly by suspending the proposition $G(n)$ directly in the persistent context just as we did with the proof involving $F(n)$ above. To use these suspended propositions, we introduce a hypothesis rule for positive propositions suspended in the persistent context.

$$\overline{\Gamma, \langle A^+ \rangle; \cdot \vdash [A^+]}\ id_p^+$$

This rule is, of course, exactly the id_2^+ rule from our discussion of Andreoli's system. There is also a focal substitution principle, Theorem 2.8. This theorem was true in Andreoli's system, but we did not need or discuss it.

Theorem 2.8 (Focal substitution (positive, persistent))**.**
If $\Gamma; \cdot \vdash [A^+]$ *and* $\Gamma, \langle A^+ \rangle; \underline{\Delta'} \vdash \underline{U}$*, then* $\Gamma; \underline{\Delta'} \vdash \underline{U}$*.*

Proof. Once again, this is a straightforward induction over the second given derivation, as in a proof of regular substitution in a natural deduction system. If the second derivation is the axiom id_p^+ applied to the suspended proposition $\langle A^+ \rangle$ we are substituting for, then the result follows immediately using the first given derivation. □

Given this focal substitution principle, we can consider the class of *permeable* positive propositions. A permeable proposition is one where, when we use the admissible η^+ rule to suspend it in the linear context, we might just as well suspend it in the persistent context, as it decomposes entirely into persistent pieces. In other words, we want a class of propositions A_p^+ such that $\Gamma, \langle A_p^+ \rangle; \Delta \vdash U$ implies $\Gamma; \Delta, A_p^+ \vdash U$; this is the permeable identity expansion property.

[13]Permeable *negative* propositions are relevant to classical linear logic, but the asymmetry of intuitionistic linear logic means that, for now, it is reasonable to consider permeability exclusively as a property of positive propositions. We will consider a certain kind of right-permeable propositions in Chapter 3.

$$\dfrac{\overset{\mathcal{D}}{\Gamma, \langle !A\rangle; \Delta \vdash U}}{\Gamma; \Delta, !A \vdash U}\ \eta_p^+ \quad \Longrightarrow \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, A; [A] \vdash \langle A\rangle}\ id^-}{\Gamma, A; \cdot \vdash \langle A\rangle}\ copy}{\Gamma, A; \cdot \vdash A}\ \eta^-}{\Gamma, A; \cdot \vdash [!A]}\ !_R \qquad \dfrac{\overset{\mathcal{D}}{\Gamma, \langle !A\rangle; \Delta \vdash U}}{\Gamma, A, \langle !A\rangle; \Delta \vdash U}\ weaken}{\Gamma, A; \Delta \vdash U}\ subst_p^+}{\Gamma; \Delta, !A \vdash U}\ !_L$$

$$\dfrac{\overset{\mathcal{D}}{\Gamma, \langle \mathbf{1}\rangle; \Delta \vdash U}}{\Gamma; \Delta, \mathbf{1} \vdash U}\ \eta_p^+ \quad \Longrightarrow \quad \dfrac{\dfrac{\dfrac{}{\Gamma; \cdot \vdash [\mathbf{1}]}\ \mathbf{1}_R \qquad \overset{\mathcal{D}}{\Gamma; \Delta, \langle \mathbf{1}\rangle \vdash U}}{\Gamma; \Delta \vdash U}\ subst_p^+}{\Gamma; \Delta, \mathbf{1} \vdash U}\ \mathbf{1}_L$$

$$\dfrac{\overset{\mathcal{D}}{\Gamma, \langle A \otimes B\rangle; \Delta \vdash U}}{\Gamma; \Delta, A \otimes B \vdash U}\ \eta_p^+ \quad \Longrightarrow \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, \langle A\rangle, \langle B\rangle; \cdot \vdash [A]}\ id_p^+ \qquad \dfrac{}{\Gamma, \langle A\rangle, \langle B\rangle; \cdot \vdash [B]}\ id_p^+}{\Gamma, \langle A\rangle, \langle B\rangle; \cdot \vdash [A \otimes B]} \qquad \dfrac{\overset{\mathcal{D}}{\Gamma, \langle A\rangle, \langle B\rangle, \langle A \otimes B\rangle; \Delta \vdash U}}{\Gamma, \langle A \otimes B\rangle; \Delta \vdash U}\ weaken}{\Gamma, \langle A\rangle, \langle B\rangle; \Delta \vdash U}\ subst_p^+}{\Gamma, \langle A\rangle; \Delta, B \vdash U}\ \eta_p^+}{\Gamma; \Delta, A, B \vdash U}\ \eta_p^+}{\Gamma; \Delta, A \otimes B \vdash U}\ \otimes_L$$

Figure 2.14: Persistent identity expansion

It is possible to precisely characterize the MELL propositions that are permeable as a syntactic refinement of positive propositions:

$$A_p^+ ::= !A^- \mid \mathbf{1} \mid A_p^+ \otimes B_p^+$$

In full first-order linear logic, $\mathbf{0}$, $A_p^+ \oplus B_p^+$, and $\exists x.A_p^+$ would be included as well; essentially only p^+ and $\downarrow A^-$ are excluded from this fragment.

Theorem 2.9 (Permeable identity expansion)**.** *If* $\Gamma, \langle A_p^+\rangle; \Delta \vdash U$*, then* $\Gamma; \Delta, A_p^+ \vdash U$.

Proof. Induction over the structure of the proposition A_p^+ or A^-. The cases of this proof are represented in Figure 2.14. □

As admissible rules, Theorems 2.8 and 2.9 are written $subst_p^+$ and η_p^+:

$$\dfrac{\Gamma; \cdot \vdash [A^+] \quad \Gamma, \langle A^+\rangle; \underline{\Delta} \vdash \underline{U}}{\Gamma; \underline{\Delta} \vdash \underline{U}}\ subst_p^+ \qquad \dfrac{\Gamma, \langle A_p^+\rangle; \Delta \vdash U}{\Gamma; \Delta, A_p^+ \vdash U}\ \eta_p^+$$

We can use this persistent identity expansion property to give a compressed proof of our motivating example:

$$\dfrac{\dfrac{\dfrac{\overline{\Gamma; \langle G(n)\rangle \vdash [G(n)]}\ id^+}{\Gamma, \langle G(n)\rangle; \cdot \vdash [!G(n)]}\ !_R}{\Gamma, \langle G(n)\rangle; \cdot \vdash !G(n)}\ focus_R}{\Gamma; G(n) \vdash !G(n)}\ \eta_p^+$$

Permeable atomic propositions

It would have been possible, in the discussion of focused linear logic in Section 2.3, to present identity expansion as conceptually prior to atomic propositions. In such a retelling, the η^+ and η^- rules can be motivated as the necessary base cases of identity expansion when we have propositional variables that stand for unknown positive and negative propositions, respectively. Conversely, we can now present a new class of *permeable* atomic propositions p_p^+ that stand in for arbitrary permeable propositions A_p^+. These add a new base case to permeable identity expansion (Theorem 2.9) that can only be satisfied with an explicit η_p^+ rule:

$$\dfrac{\Gamma, \langle p_p^+\rangle; \Delta \vdash U}{\Gamma; \Delta, p_p^+ \vdash U}\ \eta_p^+$$

Because the permeable propositions are a syntactic refinement of the positive propositions, p_p^+ must be a valid positive atomic proposition as well. This is the revised grammar for intuitionistic MELL with permeable atomic propositions:

$$\begin{aligned} A^+ &::= p^+ \mid p_p^+ \mid \downarrow A^- \mid !A^- \mid \mathbf{1} \mid A^+ \otimes B^+ \\ A_p^+ &::= p_p^+ \mid !A^- \mid \mathbf{1} \mid A_p^+ \otimes B_p^+ \\ A^- &::= p^- \mid \uparrow A^+ \mid A^+ \multimap B^- \end{aligned}$$

This addition to the logic requires some additions to positive identity expansion, cut admissibility, and completeness, but none of the changes are too severe; we consider each in turn.

Identity expansion The new addition to the language of positive propositions requires us to extend identity expansion with one additional case:

$$\dfrac{\Gamma; \Delta, \langle p_p^+\rangle \vdash U}{\Gamma; \Delta, p_p^+ \vdash U}\ \eta_p^+ \quad \Longrightarrow \quad \dfrac{\dfrac{\overline{\Gamma, \langle p_p^+\rangle; \cdot \vdash [p_p^+]}\ id_p^+ \quad \dfrac{\Gamma; \Delta, \langle p_p^+\rangle \vdash U}{\Gamma, \langle p_p^+\rangle; \Delta, \langle p_p^+\rangle \vdash U}\ weaken}{\Gamma, \langle p_p^+\rangle; \Delta \vdash U}\ subst_p^+}{\Gamma; \Delta, p_p^+ \vdash U}\ \eta_p^+$$

Cut admissibility We must clarify the restriction on cut admissibility for our extended logic. In Theorem 2.4, we required that sequents contain only suspensions of atomic propositions, and in our generalization of cut admissibility, we need to further require that all suspensions in the persistent context Γ be permeable and atomic and that all suspensions in the linear context Δ be non-permeable and atomic. Under this restriction, the proof proceeds much as it did for the system with the exponential optimization.

Correctness of focusing There are two ways we can understand the soundness and completeness of focusing for linear logic extended with permeable atomic propositions. One option is to add a notion of permeable atomic propositions to our core linear logic from Figure 2.1, in which case soundness and completeness are straightforward. Alternatively, we can use our intuition that a permeable proposition A is interprovable with $!A$ and let $(p_p^+)^\circ = !p_p^+$.

The erasure of permeable propositions p_p^+ in the focused logic to $!p_p^+$ in the unfocused logic reveals that permeable propositions, which we motivated entirely from a discussion of identity expansion, are effectively a logical treatment of separation. Rather than $\Uparrow$, a separate proposition that we apply only to positive propositions, permeability is a property intrinsic to a given atomic proposition, much like the proposition's positivity or negativity.

2.6 Revisiting our notation

Andreoli, in his 2001 paper introducing the idea of synthetic inference rules [And01], observed that the atom optimization can lead to an exponential explosion in the number of synthetic rules associated with a proposition. For instance, if $a^+ \otimes b^+ \multimap \uparrow c^+$ appears in Γ, the atom optimization means that the following are all synthetic inference rules for that proposition:

$$\frac{\Gamma; \Delta, \langle c^+ \rangle \vdash U}{\Gamma; \Delta, \langle a^+ \rangle, \langle b^+ \rangle \vdash U} \quad \frac{\Gamma, \langle a^+ \rangle; \Delta, \langle c^+ \rangle \vdash U}{\Gamma, \langle a^+ \rangle; \Delta, \langle b^+ \rangle \vdash U}$$

$$\frac{\Gamma, \langle b^+ \rangle; \Delta, \langle c^+ \rangle \vdash U}{\Gamma, \langle b^+ \rangle; \Delta, \langle a^+ \rangle \vdash U} \quad \frac{\Gamma, \langle a^+ \rangle, \langle b^+ \rangle; \Delta, \langle c^+ \rangle \vdash U}{\Gamma, \langle a^+ \rangle, \langle b^+ \rangle; \Delta \vdash U}$$

Andreoli suggests coping with this problem by restricting the form of propositions so that positive atoms never appear in the persistent context. From our perspective, this is a rather unusual recommendation, since it just returns us to linear logic without the atom optimization! The focused system in Section 2.3, which we have argued is a more fundamental presentation (following Chaudhuri), effectively avoid this problem.

However, it's not necessary to view Andreoli's proliferation of rules as a problem with the logic; rather, it is possible to view it merely as a problem of notation. It is already the case that, in writing sequent calculus rules, we tacitly use of a fairly large number of notational conventions, at least relative to Gentzen's original formulation where all contexts were treated as sequences of propositions [Gen35]. For instance, the bottom-up reading of the $\mathbf{1}_R$ rule's conclusion, $\Gamma; \cdot \vdash [\mathbf{1}]$, indicates the presence of an additional premise checking that the linear context is empty, and the conclusion $\Gamma; \Delta_1, \Delta_2 \vdash [A \otimes B]$ of the $\otimes_R$ rule indicates the condition that the context can be split into two parts. In other words, both the conclusion of the $\mathbf{1}_R$ rule and $\otimes_R$ rule, as we normally write them, can be seen as having special *matching constructs* that constrain the shape of the context Δ.[14]

I propose to deal with the apparent proliferation of synthetic rules in a system with the atom optimization by adding a new matching construct for the conclusion of rules. We can say that

[14]More than anything else we have discussed so far, this is a view of inference rules that emphasizes *bottom-up* proof search and proof construction. A view of linear logic that is informed by the inverse method, or top-down proof construction, is bound to look very different (see, for example, [Cha06]).

$$\frac{}{\Gamma; \cdot / p \Longrightarrow p}\ \mathit{init}$$

$$\frac{\Gamma; \cdot \Longrightarrow A}{\Gamma; \cdot \Longrightarrow !A}\ !_R \quad \frac{\Gamma, A; \Delta \Longrightarrow C}{\Gamma; \Delta / !A \Longrightarrow C}\ !_L \quad \frac{}{\Gamma; \cdot \Longrightarrow \mathbf{1}}\ \mathbf{1}_R \quad \frac{\Gamma; \Delta \Longrightarrow C}{\Gamma; \Delta / \mathbf{1} \Longrightarrow C}\ \mathbf{1}_L$$

$$\frac{\Gamma; \Delta \Longrightarrow A \quad \Gamma; \Delta \Longrightarrow B}{\Gamma; \Delta_1, \Delta_2 \Longrightarrow A \otimes B}\ \otimes_R \quad \frac{\Gamma; \Delta, A, B \Longrightarrow C}{\Gamma; \Delta / A \otimes B \Longrightarrow C}\ \otimes_L$$

$$\frac{\Gamma; \Delta, A \Longrightarrow B}{\Gamma; \Delta \Longrightarrow A \multimap B}\ \multimap_R \quad \frac{\Gamma; \Delta_1 \Longrightarrow A \quad \Gamma; \Delta_2, B \Longrightarrow C}{\Gamma; \Delta_1, \Delta_2 / A \multimap B \Longrightarrow C}\ \multimap_L$$

Figure 2.15: Alternative presentation of intuitionstic linear logic

$\Gamma; \Delta$ matches $\Gamma; \Delta' / \langle p^+ \rangle$ either when $\langle p^+ \rangle \in \Gamma$ and $\Delta = \Delta'$ or when $\Delta = (\Delta', \langle p^+ \rangle)$. We can also iterate this construction, so that $\Gamma; \Delta$ matches $\Gamma; \Delta_n / \langle p_1^+ \rangle, \ldots, \langle p_n^+ \rangle$ if $\Gamma; \Delta$ matches $\Gamma; \Delta_1 / \langle p_1^+ \rangle$, $\Gamma; \Delta_1$ matches $\Gamma; \Delta_2 / \langle p_2^+ \rangle$, ... and $\Gamma; \Delta_{n-1}$ matches $\Gamma; \Delta_n / \langle p_n^+ \rangle$. Armed with this notation, we can create a concise synthetic connective that is equivalent to the four of the rules discussed previously:

$$\frac{\Gamma; \Delta, \langle c^+ \rangle \vdash U}{\Gamma; \Delta / \langle a^+ \rangle, \langle b^+ \rangle \vdash U}$$

This modified notation need not be reserved for synthetic connectives; we can also use it to combine the two positive identity rules id_1^+ and id_2^+ (in the exponential-optimized system) or, equivalently, id^+ and id_p^+ (in the system incorporating permeability). Furthermore, by giving $\Gamma; \Delta / A^-$ the obviously analogous meaning, we can fuse the focus_L rule and the copy rule into a single rule that is unconcerned with whether the proposition in question came from the persistent or linear contexts:

$$\frac{}{\Gamma; \cdot / \langle A^+ \rangle \vdash [A^+]}\ id^+ \quad \frac{\Gamma; \Delta, [A^-] \vdash U}{\Gamma; \Delta / A^- \vdash U}\ \mathit{focus}_L^*$$

Going yet one more step, we could use this notation to revise the original definition of linear logic in Figure 2.1. The *copy* rule in that presentation sticks out as the only rule that doesn't deal directly with a connective, but we can eliminate it by using the $\Gamma; \Delta / A$ matching construct. The resulting presentation, shown in Figure 2.15, is equivalent to the presentation in Figure 2.1.

Theorem 2.10. $\Gamma; \Delta \longrightarrow C$ *if and only if* $\Gamma; \Delta \Longrightarrow C$.

Proof. The reverse direction is a straightforward induction: each rule in Figure 2.15 can be translated as the related rule in Figure 2.1 along with (potentially) an instance of the copy rule.

The forward direction requires a lemma that the *copy* rule is admissible according to the rules of Figure 2.15; this lemma can be established by straightforward induction. Having established the lemma, the forward direction is a straightforward induction on derivations, applying the admissible rule whenever the *copy* rule is encountered. □

Chapter 3

Substructural logic

Linear logic is the most famous of the *substructural logics*. Traditional intuitionistic logic, which we call persistent to emphasize the treatment of truth as a persistent and reusable resource, admits the three so-called *structural rules* of weakening (premises need not be used), contraction (premises may be used multiple times) and exchange (the ordering of premises are irrelevant). Substructural logics, then, are logics that do not admit these structural rules – linear logic has only exchange, *affine* logic (which is frequently conflated with linear logic by programming language designers) has exchange and weakening, and *ordered* logic, first investigated as a proof theory by Lambek [Lam58], lacks all three.

Calling logics like linear, affine, and ordered logic substructural relative to persistent logic is greatly unfair to the substructural logics. Girard's linear logic can express persistent provability using the exponential connective $!A$, and this idea is generally applicable in substructural logics – for instance, it was applied by Polakow and Pfenning to Lambek's ordered logic [PP99]. It is certainly too late to advocate for these logics to be understood as superstructural logics, but that is undoubtedly what they are: generalizations of persistent logic that introduce more expressive power.

In this chapter, we will define a first-order ordered linear logic with a lax connective $\bigcirc A$ in both unfocused (Section 3.1) and focused (Section 3.3) flavors (this logic will henceforth be called OL_3, for ordered linear lax logic). Then, following the structural focalization methodology introduced in the previous chapter, we establish cut admissibility (Section 3.4), and identity expansion (Section 3.5) for focused OL_3; with these results, it is possible to prove the soundness and completeness of focusing (Section 3.6) for OL_3. A fragment of this system will form the basis of the logical framework in Chapter 4, and that framework will, in turn, underpin the rest of this dissertation.

Why present the rich logic OL_3 here if only the fragment detailed in Chapter 4 is needed? There are two main reasons. First, while we will use only a fragment of this logic in Chapter 4, other fragments of the logic may well be interesting and useful for other purposes. Second, the presentation in this chapter, and in particular the discussion of substructural contexts in Section 3.2, introduces a presentation style and infrastructure that I believe will generalize to focused presentations of richer logics, such as the logic of bunched implications [Pym02], non-associative ordered logic (or "rigid logic") [Sim09], subexponential logics [NM09], and so on.

Furthermore, the choice to present a full account of focusing in OL_3 is in keeping with as An-

dreoli's insistence that we should avoid ambiguity as to whether we are "defining a foundational paradigm or a [logic] programming language (two objectives that should clearly be kept separate)" [And01]. Both the full logic OL_3 and the general methodology followed in this chapter are general, foundational paradigms within which it is possible to instantiate families of logic programming languages and logical frameworks, even though we will focus on a particular logical framework starting in Chapter 4.

3.1 Ordered linear lax logic

Ordered linear logic was the subject of Polakow's dissertation [Pol01]. It extends linear logic with a notion of *ordered resources*. The multiplicative conjunction $A \otimes B$ of linear logic, which represents that we have both the resources to make an A and a B, is replaced in ordered logic by an ordered multiplicative conjunction $A \bullet B$, which represents that we have the resources to make an A, and they're to the left of the resources necessary to make a B. Linear implication $A \multimap B$, which represents a resource that, given the resources necessary to construct an A, can construct a B, splits into two propositions in ordered logic. The proposition $A \rightarrowtail B$ represents a resource that, given the resources necessary to construct an A *to the left*, can construct a B; the proposition $A \twoheadrightarrow B$ demands those resource to its right.

Ordered propositions were used by Lambek to model language [Lam58]. The word "clever" is a adjective that, given a noun to its right, constructs a noun phrase ("ideas" is a noun, and "clever ideas" is a noun phrase). Therefore, the world "clever" can be seen as an ordered resource $\mathsf{Phrase} \twoheadrightarrow \mathsf{NounPhrase}$. Similarly, the word "quietly" is an adverb that, given a verb to its left, constructs a verb phrase ("sleeps" is a verb, and "sleeps quietly" is a verb phrase). Therefore, the word "quietly" can be seen as an ordered resource $\mathsf{Verb} \rightarrowtail \mathsf{VerbPhrase}$. The key innovation made by Polakow and Pfenning was integrating both persistent and linear logic into Lambek's system with the persistent exponential $!A$ and the mobile exponential $¡A$. The latter proposition is pronounced "A mobile" or, whimsically, "gnab A" in reference to the pronunciation of $!A$ as "bang A."

The primary sequent of ordered logic is $\Gamma; \Delta; \Omega \implies A\,\mathit{true}$, which expresses that A is a resource derivable from the persistent resources in Γ, the ephemeral resources in Δ, and the ephemeral, ordered resources in Ω. The persistent context Γ and the linear context Δ are multisets as before (so we think of Δ_1, Δ_2 as being equal to Δ_2, Δ_1, for instance). The ordered context Ω is a sequence of propositions, as in Gentzen's original presentation of sequent calculi, and *not* a multiset. This means that the two ordered contexts Ω_1, Ω_2 and Ω_2, Ω_1 are, in general, not the same.

The presentation of ordered linear lax logic in Figure 3.1 uses an ordered logic adaptation of the matching constructs introduced in Section 2.6; all the left rules in that figure use the construct $\Gamma; \Delta; \Omega_L/A/\Omega_R \implies U$, which matches the sequent $\Gamma; \Delta'; \Omega' \implies U$

* if $\Omega' = (\Omega_L, A, \Omega_R)$ and $\Delta' = \Delta$;
* if $\Omega' = (\Omega_L, \Omega_R)$ and $\Delta' = (\Delta, A)$;
* or if $\Omega' = (\Omega_L, \Omega_R)$, $\Delta' = \Delta$, and $A \in \Gamma$.

As in the alternative presentation of linear logic where *copy* was admissible, both the *copy* rule

and a rule Polakow called $place$ are admissible in the logic described in Figure 3.1.

$$\frac{\Gamma, A; \Delta; \Omega_L, A, \Omega_R \Longrightarrow U}{\Gamma, A; \Delta; \Omega_L, \Omega_R \Longrightarrow U}\ copy \qquad \frac{\Gamma; \Delta; \Omega_L, A, \Omega_R \Longrightarrow U}{\Gamma; \Delta, A; \Omega_L, \Omega_R \Longrightarrow U}\ place$$

When this notation is used in the rule id, the meaning is that the atomic proposition p is either the only thing in the ordered context alongside an empty linear context, or else it is the only thing in the linear context alongside an empty ordered context, or else both the linear and ordered contexts are empty and p is present in the persistent context. This notion will be called *sole membership* in Section 3.2.1.

Ordered linear lax logic also encompasses Fairtlough and Mendler's lax logic [FM97] as reconstructed by Pfenning and Davies [PD01] and adapted to linear logic is the basis of the CLF logical framework [WCPW02]. The judgment $A\ lax$ is the foundation of ordered logic, and is usually interpreted as truth under some unspecified constraint or as a weaker version of truth: if we know $A\ true$ then we can conclude $A\ lax$:

$$\frac{\Gamma; \Delta; \Omega \Longrightarrow A\ true}{\Gamma; \Delta; \Omega \Longrightarrow A\ lax}\ lax$$

If we know $A\ lax$, on the other hand, we cannot prove $A\ true$, though we can prove $\bigcirc A\ true$, where $\bigcirc A$ is the propositional internalization of the lax judgment (rule $\bigcirc_R$ in Figure 3.1).

Lax truth is handled with the use of matching constructs, thereby making the rule lax rule above admissible just like $copy$ and $place$ are admissible. We write all the right rules in Figure 3.1 with a construct $\Gamma; \Delta; \Omega \Longrightarrow A\ lvl$ that matches both sequents of the form $\Gamma; \Delta; \Omega \Longrightarrow A\ true$ and sequents of the form $\Gamma; \Delta; \Omega \Longrightarrow A\ lax$ – in other words, we treat lvl as a metavariable ("level") that stands for judgments $true$ or lax on the right. The use of this construct gives us right rules for $A \oplus B$ that look like this:

$$\frac{\Gamma; \Delta; \Omega \Longrightarrow A\ true}{\Gamma; \Delta; \Omega \Longrightarrow A \oplus B\ lvl}\ \oplus_{R1} \qquad \frac{\Gamma; \Delta; \Omega \Longrightarrow B\ true}{\Gamma; \Delta; \Omega \Longrightarrow A \oplus B\ lvl}\ \oplus_{R2}$$

The metavariable U is even more generic, standing for an arbitrary succedent $A\ true$ or $A\ lax$. Putting the pieces of ordered linear lax logic together into one sequent calculus in Figure 3.1 is a relatively straightforward proof-theoretic exercise; the language of propositions is as follows:

$$\begin{array}{ll} \text{Propositions} & A, B, C ::= p \mid ¡A \mid !A \mid \bigcirc A \mid \\ & \qquad \mathbf{1} \mid A \bullet B \mid A \rightarrowtail B \mid A \twoheadrightarrow B \mid \mathbf{0} \mid A \oplus B \mid \top \mid A \,\&\, B \mid \\ & \qquad \exists a{:}\tau.A \mid \forall a{:}\tau \mid t \doteq_\tau s \end{array}$$

The connectives and $\mathbf{1}$, $\mathbf{0}$, $A \oplus B$, $\top$, and $A \,\&\, B$ were not mentioned above but are closely analogous to their counterparts in linear logic. The first-order connectives $\exists a{:}\tau.A$, $\forall a{:}\tau$, and $t \doteq_\tau s$ will be discussed presently.

Atomic propositions

$$\frac{}{\Gamma; \cdot; /p/ \Longrightarrow p\ lvl}\ id$$

Exponentials

$$\frac{\Gamma; \Delta; \cdot \Longrightarrow A\ true}{\Gamma; \Delta; \cdot \Longrightarrow ¡A\ lvl}\ ¡_R \quad \frac{\Gamma; \Delta, A; \Omega_L, \Omega_R \Longrightarrow U}{\Gamma; \Delta; \Omega_L / ¡A / \Omega_R \Longrightarrow U}\ ¡_L$$

$$\frac{\Gamma; \cdot; \cdot \Longrightarrow A\ true}{\Gamma; \cdot; \cdot \Longrightarrow !A\ lvl}\ !_R \quad \frac{\Gamma, A; \Delta; \Omega_L, \Omega_R \Longrightarrow U}{\Gamma; \Delta; \Omega_L / !A / \Omega_R \Longrightarrow U}\ !_L$$

$$\frac{\Gamma; \Delta; \Omega \Longrightarrow A\ lax}{\Gamma; \Delta; \Omega \Longrightarrow \bigcirc A\ lvl}\ \bigcirc_R \quad \frac{\Gamma; \Delta; \Omega_L, A, \Omega_R \Longrightarrow C\ lax}{\Gamma; \Delta; \Omega_L / \bigcirc A / \Omega_R \Longrightarrow C\ lax}\ \bigcirc_L$$

Multiplicative connectives

$$\frac{}{\Gamma; \cdot; \cdot \Longrightarrow \mathbf{1}\ lvl}\ \mathbf{1}_R \quad \frac{\Gamma; \Delta; \Omega_L, \Omega_R \Longrightarrow U}{\Gamma; \Delta; \Omega_L / \mathbf{1} / \Omega_R \Longrightarrow U}\ \mathbf{1}_L$$

$$\frac{\Gamma; \Delta; \Omega_L \Longrightarrow A\ true \quad \Gamma; \Delta; \Omega_R \Longrightarrow B\ true}{\Gamma; \Delta_1, \Delta_2; \Omega_L, \Omega_R \Longrightarrow A \bullet B\ lvl}\ \bullet_R \quad \frac{\Gamma; \Delta; \Omega_L, A, B, \Omega_R \Longrightarrow U}{\Gamma; \Delta; \Omega_L / A \bullet B / \Omega_R \Longrightarrow U}\ \bullet_L$$

$$\frac{\Gamma; \Delta; A, \Omega \Longrightarrow B\ true}{\Gamma; \Delta; \Omega \Longrightarrow A \rightarrowtail B\ lvl}\ \rightarrowtail_R \quad \frac{\Gamma; \Delta_A; \Omega_A \Longrightarrow A\ true \quad \Gamma; \Delta; \Omega_L, B, \Omega_R \Longrightarrow U}{\Gamma; \Delta_A, \Delta; \Omega_L, \Omega_A / A \rightarrowtail B / \Omega_R \Longrightarrow U}\ \rightarrowtail_L$$

$$\frac{\Gamma; \Delta; \Omega, A \Longrightarrow B\ true}{\Gamma; \Delta; \Omega \Longrightarrow A \twoheadrightarrow B\ lvl}\ \twoheadrightarrow_R \quad \frac{\Gamma; \Delta_A; \Omega_A \Longrightarrow A\ true \quad \Gamma; \Delta; \Omega_L, B, \Omega_R \Longrightarrow U}{\Gamma; \Delta_A, \Delta; \Omega_L / A \twoheadrightarrow B / \Omega_A, \Omega_R \Longrightarrow U}\ \twoheadrightarrow_L$$

Additive connectives

$$\frac{}{\Gamma; \Delta; \Omega_L / \mathbf{0} / \Omega_R \Longrightarrow U}\ \mathbf{0}_L \quad \frac{\Gamma; \Delta; \Omega \Longrightarrow A\ true}{\Gamma; \Delta; \Omega \Longrightarrow A \oplus B\ lvl}\ \oplus_{R1} \quad \frac{\Gamma; \Delta; \Omega \Longrightarrow B\ true}{\Gamma; \Delta; \Omega \Longrightarrow A \oplus B\ lvl}\ \oplus_{R2}$$

$$\frac{\Gamma; \Delta; \Omega_L, A, \Omega_R \Longrightarrow U \quad \Gamma; \Delta; \Omega_L, B, \Omega_R \Longrightarrow U}{\Gamma; \Delta; \Omega_L / A \oplus B / \Omega_R \Longrightarrow U}\ \oplus_L$$

$$\frac{}{\Gamma; \Delta; \Omega \Longrightarrow \top\ lvl}\ \top_R \quad \frac{\Gamma; \Delta; \Omega \Longrightarrow A\ true \quad \Gamma; \Delta; \Omega \Longrightarrow B\ true}{\Gamma; \Delta; \Omega \Longrightarrow A \,\&\, B\ lvl}\ \&_R$$

$$\frac{\Gamma; \Delta; \Omega_L, A, \Omega_R \Longrightarrow U}{\Gamma; \Delta; \Omega_L / A \,\&\, B / \Omega_R \Longrightarrow U}\ \&_{L1} \quad \frac{\Gamma; \Delta; \Omega_L, B, \Omega_R \Longrightarrow U}{\Gamma; \Delta; \Omega_L / A \,\&\, B / \Omega_R \Longrightarrow U}\ \&_{L2}$$

Figure 3.1: Propositional ordered linear lax logic

$$\frac{\Psi \vdash t : \tau \quad \Gamma; \Delta; \Omega \Longrightarrow [t/a]B\ true}{\Psi; \Gamma; \Delta; \Omega \Longrightarrow \exists a{:}\tau.B\ lvl}\ \exists_R \qquad \frac{\Psi, a{:}\tau; \Gamma; \Delta; \Omega_L, B, \Omega_R \Longrightarrow U}{\Psi; \Gamma; \Delta; \Omega_L/\exists a{:}\tau.B/\Omega_R \Longrightarrow U}\ \exists_L$$

$$\frac{\Psi, a{:}\tau; \Gamma; \Delta; \Omega \Longrightarrow B\ true}{\Psi; \Gamma; \Delta; \Omega \Longrightarrow \forall a{:}\tau.B\ lvl}\ \forall_R \qquad \frac{\Psi \vdash t : \tau \quad \Psi; \Gamma; \Delta; \Omega_L, [t/a]B, \Omega_R \Longrightarrow U}{\Psi; \Gamma; \Delta; \Omega_L/\forall a{:}\tau.B/\Omega_R \Longrightarrow U}\ \forall_L$$

$$\frac{}{\Psi; \Gamma; \Delta; \Omega \Longrightarrow t \doteq_\tau t\ lvl}\ \doteq_R$$

$$\frac{\forall(\Psi' \vdash \sigma : \Psi).\quad \sigma t = \sigma s \quad \longrightarrow \quad \Psi'; \sigma\Gamma; \sigma\Delta; \sigma\Omega_L, \sigma\Omega_R \Longrightarrow \sigma U}{\Psi; \Gamma; \Delta; \Omega_L/t \doteq_\tau s/\Omega_R \Longrightarrow U}\ \doteq_L$$

Figure 3.2: First-order ordered linear lax logic

3.1.1 First-order logic

The presentation in Figure 3.1 is propositional; by uniformly adding a first-order context Ψ to all sequents it can be treated as first-order. We define quantification (existential and universal), as well as first-order equality,[1] in Figure 3.2.

The equality proposition $t \doteq_\tau s$ is an interesting addition to our presentation of the logic, and will be present in the framework SLS defined in the next chapter, albeit in a highly restricted form. Equality in SLS will be used primarily in the logical transformations presented in Chapter 7 and in the program analysis methodology in Chapter 8. The left rule for equality $t \doteq_\tau s$ has a higher-order premise, in the sense that it reflects over the definition of simultaneous term substitutions $\Psi' \vdash \sigma : \Psi$ and over the syntactic equality judgment for first-order terms $t = s$. We used this exact style of presentation previously in [SP11b], but the approach is based on Schroeder-Heister's treatment of definitional reflection [SH93].

In one sense, the left rule $\doteq_L$ is actually a rule schema: there is one premise for each substitution σ that is a unifier for t and s (a unifier is any substitution σ that makes σt and σs syntactically identical). When we induct over the structure of proofs, there is correspondingly one smaller subderivation for each unifying substitution. By this reading, $\doteq_L$ is a rule that, in general, will have countably many premises; in the case of a trivially satisfiable equality problem like $x \doteq_\tau x$ it will have one premise for each well-formed substitution that substitutes a term of the appropriate type for x. However, as suggested by Zeilberger [Zei08a], it is more auspicious to take the higher-order formulation at face value: the premise is actually a (meta-level) mapping – a function – that takes a substitution σ, the codomain Ψ' of that substitution, and any evidence necessary to show that σ unifies t and s and returns a derivation of $\Psi'; \sigma\Gamma; \sigma\Delta; \sigma\Omega_L, \sigma\Omega_R \Longrightarrow \sigma U$. When we induct over the structure of proofs, the result of applying any unifying substitution to this function is a smaller subderivation for the purpose of invoking the induction hypothesis. This functional interpretation will be reflected in the proof terms we assign to focused OL_3 in Section 3.3.3.

There are two important special cases. First, an unsatisfiable equation on the left implies a contradiction, and makes the left rule for equality equivalent (at the level of provability) to one

[1] That is, equality of terms from the domain of first-order quantification.

with no premises. For instance, this means that

$$\frac{\textit{no unifier for } t \textit{ and } s}{\Psi; \Gamma; \Delta; \Omega_L/t \doteq_\tau s/\Omega_R \Longrightarrow U} \doteq_{no}$$

is derivable – a *unifier* is just a substitution σ such that σt and σs are syntactically identical. The other important special case is when t and s have a *most general unifier* σ_{mgu}, which just means that for all $\Psi' \vdash \sigma : \Psi$ such that $\sigma t = \sigma s$, it is the case that $\sigma = \sigma' \circ \sigma_{mgu}$ for some σ'.[2] In this case, the left rule for equality is equivalent (again, at the level of determining which sequents are provable) to the following rule:

$$\frac{\sigma = mgu(t, s) \quad \Psi' \vdash \sigma : \Psi \quad \Psi'; \sigma\Gamma; \sigma\Delta; \sigma\Omega_L, \sigma\Omega_R \Longrightarrow \sigma U}{\Psi; \Gamma; \Delta; \Omega_L/t \doteq_\tau s/\Omega_R \Longrightarrow U} \doteq_{yes}$$

Therefore, given a first-order domain in which any two terms are decidably either non-unifiable or unifiable with a most general unifier, we can choose to define the logic with two rules $\doteq_{no}$ and $\doteq_{yes}$; the resulting logic will be equivalent, at the level of derivable sequents, to the logic defined with the $\doteq_L$ rule.

We have not yet thoroughly specified the type and term structure of first-order individuals; in Section 4.1 we clarify that these types and terms will actually be types and terms of Spine Form LF. This does mean that we will have to pay attention, in the proofs of this chapter, to the fact that the types of first-order terms τ are *dependent types* that may include terms t. Particularly relevant in this chapter will be simultaneous substitutions σ: the judgment $\Psi' \vdash \sigma : \Psi$ expresses that σ can map terms and propositions defined in the context Ψ (the domain of the substitution) to terms and propositions defined in the context Ψ' (the range of the substitution). Simultaneous substitutions are defined more carefully in Section 4.1.2 and in [NPP08].

3.2 Substructural contexts

First-ordered linear lax logic has a lot of contexts – the persistent context Γ, the linear context Δ, the ordered context Ω, and the first-order context Ψ. In most presentations of substructural logics, the many contexts primarily serve to obscure the logic's presentation and ensure that the LaTeX code of figures and displays remains permanently unreadable. And there are yet more contexts we might want to add, such as the affine contexts present in the Celf implementation [SNS08].

In this section, we will consider a more compact way of dealing with the contexts that we interpret as containing resources (persistent, affine, linear, or ordered resources), though we choose to maintain the distinction between resource contexts and first-order variable contexts Ψ. The particular way we define substructural contexts can be generalized substantially: it would be possible to extend this presentation to the affine exponential $@A$, and we conjecture that the subexponentials discussed by Nigam and Miller [NM09] – as well as richer logics like the logic of bunched implications [Pym02] – could be given a straightforward treatment using this notation.

[2] Where $\circ$ is composition – $(\sigma' \circ \sigma_{mgu})t = \sigma'(\sigma_{mgu}t)$.

We write unified substructural contexts as either Δ or Ξ, preferring the latter when there is a chance of confusing them with linear contexts Δ. For the purposes of encoding OL_3, we can see these contexts as sequences of variable declarations, defined by the grammar

$$\Xi ::= \cdot \mid \Xi, x{:}T\ ord \mid \Xi, x{:}T\ eph \mid \Xi, x{:}T\ pers$$

where each of the *variables* x are distinct, so that the context also represents a finite map from variables x to *judgments* $T\ lvl$, where lvl is either ord, eph, or $pers$. By separating out a substructural context into three subsequences of persistent, linear, and ordered judgments, we can recover the presentations of contexts for OL_3 given in Figure 3.1. We will use this observation in an informal way throughout the chapter, writing $\Xi = \Gamma; \Delta; \Omega$.

The domain represented by the metavariable T is arbitrary: when discussing the unfocused logic given in Figure 3.1, T varies over unpolarized propositions A, but when discussing a focused logic in Section 3.3 it will vary over stable negative propositions A^-, positive suspended propositions $\langle A^+ \rangle$, focused negative propositions $[A^-]$, and inverting positive propositions A^+.

The key innovation in this presentation was already present in the unfocused logic shown in Figure 3.1: we need to differentiate *constructions*, which appear in the premises of rules, and *matching constructs*, which appear in the conclusions of rules. The notation $\Gamma; \Delta; \Omega_L / A \bullet B / \Omega_R$ that appears in the conclusion of $\bullet_L$ is a matching construct; as discussed in Section 3.1, there are multiple ways in which a context $\Gamma'; \Delta'; \Omega'$ could match this context, because $A \bullet B$ could come from any of the three contexts. However, $\Gamma; \Delta; \Omega_L, A, B, \Omega_R$ in the premise of $\bullet_L$ is a construction, and is unambiguously equal to only one context $\Gamma'; \Delta'; \Omega'$ – the one where $\Gamma' = \Gamma$, $\Delta' = \Delta$, and $\Omega' = \Omega_L, A, B, \Omega_R$.

3.2.1 Fundamental operations on contexts

The first fundamental idea we consider is *singleton* contexts. We construct a single-element context by writing $x{:}T\ lvl$. The corresponding matching construct on contexts is $x{:}T$. In unfocused OL_3, we say that Ξ matches $x{:}A$ if its decomposition into persistent, linear, and ordered contexts matches $\Gamma; \cdot; /A/$. Specifically,

Definition 3.1 (Sole membership)**.** *Ξ matches $x{:}T$ if*

- *Ξ contains no linear judgments and contains exactly one ordered judgment $x{:}T\ ord$ (corresponding to the situation where $\Xi = \Gamma; \cdot; T$),*
- *Ξ contains no ordered judgments and contains exactly one linear judgment $x{:}T\ eph$ (corresponding to the situation where $\Xi = \Gamma; T; \cdot$), or*
- *Ξ contains only persistent judgments, including $x{:}T\ pers$ (corresponding to the situation where $\Xi = \Gamma, T; \cdot; \cdot$).*

Sole membership is related to the initial sequents and the matching construct $\Gamma; \cdot; /A/$ for contexts that was used in Figure 3.1. We could rewrite the *id* rule from that figure as follows:

$$\frac{}{x{:}p \Longrightarrow p\ lvl}\ id$$

As with all rules involving matching constructs in the conclusion, it is fair to view the matching construct as an extra premise; thus, the id rule above is the same as the id rule below:

$$\frac{\Xi \textit{ matches } x{:}p}{\Xi \Longrightarrow p\ \mathit{lvl}}\ id$$

The second basic operation on contexts requires a new concept, *frames* Θ. Intuitively, we can view a frame as a set of persistent, linear, and ordered contexts where the ordered context is missing a particular piece. We can write this missing piece as a box: $\Gamma; \Delta; \Omega_L, \Box, \Omega_R$. Alternatively, we can think of a frame as a one-hole context or Huet-style zipper [Hue97] over the structure of substructural contexts. We will also think of them morally as linear functions $(\lambda\Xi.\ \Xi_L, \Xi, \Xi_R)$ as in [Sim09].

The construction associated with frames, $\Theta\{\Xi\}$, is just a straightforward operation of filling in the hole or β-reducing the linear function; doing this requires that the variables in Θ and Ξ be distinct. If we think of Θ informally as $\Gamma; \Delta; \Omega_L, \Box, \Omega_R$, then this is *almost* like the operation of filling in the hole, as $\Theta\{x{:}A\ \mathit{ord}\} = \Gamma; \Delta; \Omega_L, A, \Omega_R$. The main difference is that we can also use the operation to insert linear propositions ($\Theta\{x{:}A\ \mathit{eph}\} = \Gamma; \Delta, A; \Omega_L, \Omega_R$) and persistent propositions ($\Theta\{x{:}A\ \mathit{pers}\} = \Gamma, A; \Delta; \Omega_L, \Omega_R$).

The construction associated with frames is straightforward, but the matching construct associated with frames is a bit more complicated. Informally, if we treat linear contexts as multisets and say that $\Xi = \Gamma; \Delta, \Delta'; \Omega_L, \Omega', \Omega_R$, then we can say $\Xi = \Theta\{\!\{\Xi'\}\!\}$ in the case that $\Theta = \Gamma; \Delta; \Omega_L, \Box, \Omega_R$ and $\Xi' = \Gamma; \Delta'; \Omega'$. The sub-context Ξ', then, has been *framed off* from Ξ, its frame is Θ. If we only had ordered judgments $T\ \mathit{ord}$, then the framing-off matching construct $\Theta\{\!\{\Xi'\}\!\}$ would be essentially the same as the construction form $\Theta\{\Xi'\}$. However, persistent and linear judgments can be reordered in the process of matching, and persistent judgments always end up in both the frame and the framed-off context.

Definition 3.2 (Framing off)**.** *Ξ matches $\Theta\{\!\{\Xi'\}\!\}$ if the union of the variables in Θ and Ξ' is exactly the variables in Ξ and*

- *if $x{:}T\ \mathit{pers} \in \Xi$, then the same variable declaration appears in Θ and Ξ';*
- *if $x{:}T\ \mathit{eph} \in \Xi$ or $x{:}T\ \mathit{ord} \in \Xi$, then the same variable declaration appears in Θ or Ξ' (but not both);*
- *in both Θ and Ξ', the sequence of variable declarations $x{:}T\ \mathit{ord}$ is a subsequence of Ξ; and*
- *if $x{:}T\ \mathit{ord} \in \Theta$, then either*
  - *for all $y{:}T'\ \mathit{ord} \in \Xi'$, the variable declaration for x appeared before the variable declaration for y in Ξ, or*
  - *for all $y{:}T'\ \mathit{ord} \in \Xi'$, the variable declaration for x appeared after the variable declaration for y in Ξ.*

We can use the framing-off notation to describe one of the cut principles for ordered linear lax logic as follows:

$$\frac{\Xi \Longrightarrow A\ \mathit{true} \quad \Theta\{x{:}A\ \mathit{true}\} \Longrightarrow C\ \mathit{true}}{\Theta\{\!\{\Xi\}\!\} \Longrightarrow C\ \mathit{true}}\ cut$$

Especially for the eventual proof of this cut principle, it is important to consider that the admissible rule above is equivalent to the following admissible rule, which describes the matching as an explicit extra premise:

$$\frac{\Xi \Longrightarrow A\ \mathit{true} \quad \Theta\{x{:}A\ \mathit{true}\} \Longrightarrow C\ \mathit{true} \quad \Xi'\ \mathit{matches}\ \Theta\{\!\{\Xi\}\!\}}{\Xi' \Longrightarrow C\ \mathit{true}}\ \mathit{cut}$$

An important derived matching construct is $\Theta\{\!\{x{:}T\}\!\}$, which matches Ξ if Ξ matches $\Theta\{\!\{\Xi'\}\!\}$ for some Ξ' such that Ξ' matches $x{:}T$. This notation is equivalent to the matching construct $\Gamma; \Delta; \Omega_L/A/\Omega_R \Longrightarrow U$ from Figure 3.1, which is need to describe almost every left rule for OL_3. Here are three rules given with this matching construct:

$$\frac{\Theta\{y{:}A\ \mathit{ord}\} \Longrightarrow U \quad \Theta\{z{:}B\ \mathit{ord}\} \Longrightarrow U}{\Theta\{\!\{x{:}A \oplus B\}\!\} \Longrightarrow U} \qquad \frac{\Theta\{y{:}A\ \mathit{ord}\} \Longrightarrow U}{\Theta\{\!\{x{:}A \mathbin{\&} B\}\!\} \Longrightarrow U} \qquad \frac{\Theta\{y{:}B\ \mathit{ord}\} \Longrightarrow U}{\Theta\{\!\{x{:}A \mathbin{\&} B\}\!\} \Longrightarrow U}$$

To reemphasize, the reason we use the matching construct $\Theta\{\!\{x{:}A\}\!\}$ in the conclusions of rules is the same reason that we used the notation $\Gamma; \Delta; \Omega_L/A/\Omega_R$ in Figures 3.1 and 3.2: it allows us to generically talk about hypotheses associated with the judgments ord, eph, and pers. The following rules are all derivable using the last of the three rules above:

$$\frac{\Theta\{y{:}B\ \mathit{ord}\} \Longrightarrow U}{\Theta\{x{:}A \mathbin{\&} B\ \mathit{ord}\} \Longrightarrow U} \qquad \frac{\Theta\{y{:}B\ \mathit{ord}\} \Longrightarrow U}{\Theta\{x{:}A \mathbin{\&} B\ \mathit{eph}\} \Longrightarrow U} \qquad \frac{\Theta\{x{:}A \mathbin{\&} B\ \mathit{pers}, y{:}B\ \mathit{ord}\} \Longrightarrow U}{\Theta\{x{:}A \mathbin{\&} B\ \mathit{pers}\} \Longrightarrow U}$$

The consistent use of matching constructs like $\Theta\{\!\{\Delta\}\!\}$ in the conclusion of rules is also what gives us the space to informally treat syntactically distinct sequences of variable declarations as equivalent. As an example, we can think of $(x{:}A\ \mathit{eph}, y{:}B\ \mathit{eph})$ and $(y{:}B\ \mathit{eph}, x{:}A\ \mathit{eph})$ as equivalent by virtue of the fact that they satisfy the same set of matching constructs. Obviously, this means that none of the matching constructs presented in the remainder of this section will observe the ordering of ephemeral or persistent variable declarations.

3.2.2 Multiplicative operations

To describe the multiplicative connectives of OL_3, including the critical implication connectives, we need to have multiplicative operations on contexts. As a construction, Ξ_L, Ξ_R is just the syntactic concatenation of two contexts with distinct variable domains, and the unit $\cdot$ is just the empty sequence. The matching constructs are more complicated to define, but the intuition is, again, uncomplicated: if $\Xi = \Gamma; \Delta, \Delta'; \Omega_L, \Omega_R$, where linear contexts are multisets and ordered contexts are sequences, then $\Xi = \Xi_L, \Xi_R$ if $\Xi_L = \Gamma; \Delta; \Omega_L$ and $\Xi_R = \Gamma; \Delta'; \Omega_R$. Note that here we are using the same notation for constructions and matching constructs: Ξ_L, Ξ_R is a matching construct when it appears in the conclusion of a rule, Ξ_L, Ξ_R is a construction when it appears in the premise of a rule.

Definition 3.3 (Conjunction)**.**

Ξ matches $\cdot$ if Ξ contains only persistent judgments.

Ξ matches Ξ_L, Ξ_R if the union of the variables in Ξ_L and Ξ_R is exactly the variables in Ξ and

* *if $x{:}T\ pers \in \Xi$, then the same variable declaration appears in Ξ_L and Ξ_R;*
* *if $x{:}T\ eph \in \Xi$ or $x{:}T\ ord \in \Xi$, then the same variable declaration appears in Ξ_L or Ξ_R (but not both);*
* *in both Ξ_L and Ξ_R, the sequence of variable declarations $x{:}T\ ord$ is a subsequence of Ξ; and*
* *if $x{:}T\ ord \in \Xi_L$ and $y{:}T'\ ord \in \Xi_R$, then the variable declaration for x appeared before the variable declaration for y in Ξ.*

The constructs for context conjunction are put to obvious use in the description of multiplicative conjunction, which is essentially just the propositional internalization of context conjunction:

$$\frac{\Xi_L \Longrightarrow A\ true \quad \Xi_R \Longrightarrow B\ true}{\Xi_L, \Xi_R \Longrightarrow A \bullet B\ lvl} \qquad \frac{\Theta\{y{:}A, z{:}B\} \Longrightarrow U}{\Theta\{\!\{x{:}A \bullet B\}\!\} \Longrightarrow U} \qquad \frac{}{\cdot \Longrightarrow \mathbf{1}\ lvl} \qquad \frac{\Theta\{\cdot\} \Longrightarrow U}{\Theta\{\!\{x{:}\mathbf{1}\}\!\} \Longrightarrow U}$$

$$\frac{x{:}A\ ord, \Xi \Longrightarrow B\ true}{\Xi \Longrightarrow A \rightarrowtail B\ lvl} \qquad \frac{\Xi_A \Longrightarrow A\ true \quad \Theta\{y{:}B\ ord\} \Longrightarrow U}{\Theta\{\!\{\Xi_A, x{:}A \rightarrowtail B\}\!\} \Longrightarrow U}$$

$$\frac{\Xi, x{:}A\ ord \Longrightarrow B\ true}{\Xi \Longrightarrow A \twoheadrightarrow B\ lvl} \qquad \frac{\Xi_A \Longrightarrow A\ true \quad \Theta\{y{:}B\ ord\} \Longrightarrow U}{\Theta\{\!\{x{:}A \twoheadrightarrow B, \Xi_A\}\!\} \Longrightarrow U}$$

Implication makes deeper use of context conjunction: Ξ matches $\Theta\{\!\{\Xi_A, x{:}A \rightarrowtail B\}\!\}$ exactly when there exist Ξ' and Ξ'' such that Ξ matches $\Theta\{\!\{\Xi'\}\!\}$, Ξ' matches Ξ_A, Ξ'', and $x{:}A \rightarrowtail B$ matches Ξ''.

3.2.3 Exponential operations

The exponentials ! and ¡ do not have a construction form associated with them, unless we view the singleton construction forms $x{:}T\ pers$ and $x{:}T\ eph$ as being associated with these exponentials. The matching construct is quite simple: Ξ matches $\Xi\restriction_{pers}$ if Ξ contains no ephemeral or ordered judgments – in other words, it says that $\Xi = \Gamma; \cdot; \cdot$. This form can then be used to describe the right rule for $!A$ in unfocused OL_3:

$$\frac{\Xi \Longrightarrow A\ true}{\Xi\restriction_{pers} \Longrightarrow !A\ lvl}$$

Similarly, Ξ matches $\Xi\restriction_{eph}$ if Ξ contains no ordered judgments (that is, if $\Xi = \Gamma; \Delta; \cdot$). Ξ always matches $\Xi\restriction_{ord}$; we don't ever explicitly use this construct, but it allows us to generally refer to $\Xi\restriction_{lvl}$ in the statement of theorems like cut admissibility.

The exponential matching constructs don't actually modify contexts in the way other matching constructs do, but this is a consequence of the particular choice of logic we're considering. Given affine resources, for instance, the matching construct associated with the affine connective $@A$ would clear the context of affine facts: Ξ matches $\Xi'\restriction_{pers}$ if Ξ has only persistent and affine resources and Ξ' contains the same persistent resources as Ξ but none of the affine ones.

We can describe a mirror-image operation on succedents U. U matches $U\lfloor^{lax}$ only if it has the form $T\ lax$, and U always matches $U\lfloor^{true}$. The latter matching construct is another degenerate form that similarly allows us to refer to $U\lfloor^{lvl}$ as a generic matching construct. We write $\Delta\restriction_{lvl}$ as

	In the context Δ	As the succedent U
stable propositions	$x{:}A^-$ $ord, eph, pers$	A^+ $true, lax$
suspended propositions (also stable)	$x{:}\langle A^+\rangle$ $ord, eph, pers$	$\langle A^-\rangle$ $true, lax$
focused propositions	$x{:}[A^-]$ ord	$[A^+]$ $true$
inverting propositions	$x{:}A^+$ ord	A^- $true$

Figure 3.3: Summary of where propositions and judgments appear in OL_3 sequents

a judgment to mean that Δ matches $\Delta\restriction_{lvl}$, and write $U\lfloor^{lvl}$ as a judgment to mean that U matches $U\lfloor^{lvl}$.

The context constructions and context matching constructs that we have given are summarized as follows:

Constructions	$\Delta, \Xi ::= x{:}T\ lvl \mid \Theta\{\Delta\} \mid \cdot \mid \Delta, \Xi$
Matching constructs	$\Delta, \Xi ::= x{:}T \mid \Theta\{\!\{\Delta\}\!\} \mid \cdot \mid \Delta, \Xi \mid \Delta\restriction_{lvl}$

3.3 Focused sequent calculus

A sequent in the focused sequent calculus presentation of OL_3 has the form $\Psi; \Delta \vdash U$, where Ψ is the first-order variable context, Δ is a substructural context as described in the previous section, and U is a succedent. The domain T of the substructural context consists of stable negative propositions A^-, positive suspended propositions $\langle A^+\rangle$, focused negative propositions $[A^-]$, and inverting positive propositions A^+.

The form of the succedent U is $T\ lvl$, where lvl is either $true$ or lax; in this way, U is just a like a special substructural context with exactly one element – we don't need to care about the name of the variable, because there's only one. The domain of T for succedents is complementary to the domain of T for contexts: stable positive propositions A^+, negative suspended propositions $\langle A^-\rangle$, focused positive propositions $[A^+]$, and inverting negative propositions A^-.

Figure 3.3 summarizes the composition of contexts and succedents, taking into account the restrictions discussed below.

3.3.1 Restrictions on the form of sequents

A sequent $\Psi; \Delta \vdash U$ is *stable* when the context Δ and succedent U contain only stable propositions (A^- in the context, A^+ in the succedent) and suspended propositions ($\langle A^+\rangle$ in the context, $\langle A^-\rangle$ in the succedent). We adopt the focusing constraint discussed in Chapter 2: there is only ever at most one focused proposition in a sequent, and if there is focused proposition in the sequent, then the sequent is otherwise stable. A restriction on the rules $focus_L$ and $focus_R$ (presented below in Figure 3.5) is sufficient to enforce this restriction: reading rules from top down, we can only use a rule $focus_L$ or $focus_R$ to prove a stable sequent, and reading rules from bottom up, we can only apply $focus_L$ or $focus_R$ when we are searching for a proof of a stable sequent.

Because there is always a distinct focused proposition in a sequent, we do not need a variable name to reference the focused proposition in a context Δ any more than we need a variable name to reference the unique member of the context-like succedent U. Therefore, we can write $[B^-]\ ord$ instead of $x{:}[B^-]\ ord$. Furthermore, for presentation of focusing that we want to give it suffices to restrict focused propositions and inverting propositions so that they are always associated with the judgment ord (on the left) or $true$ (on the right). With this restriction, we can write $[A^-]$ and $x{:}A^+$ instead of $[A^-]\ ord$ and $x{:}A^+\ ord$ in Δ, and we can write $[A^+]$ and A^- instead of $[A^+]\ true$ and $A^-\ true$ for U.

In a confluent presentation of focused logic like the one given for linear logic in Chapter 2, that would be as far as we could take our simplifications. However, this presentation will use a fixed presentation of logic as described in Section 2.3.8. If there is more than one invertible proposition in a sequent, *only* the leftmost one will be eligible to have a rule or matching applied to it. All the propositions in Δ are treated as being to the left of the succedent U, so we always prioritize inversion on positive propositions in Δ. With this additional restriction, it is always unambiguous which proposition we are referring to in an invertible rule, and we write A^+ instead of $x{:}A^+$ or $x{:}A^+\ ord$.

We will maintain the notational convention (only) within this chapter that first-order variables are written as a, variables associated with stable negative propositions are written as x, and variables associated with suspended positive propositions are written as z.

In summary, the four forms of sequent in focused OL_3, which we define the rules for in Section 3.3.3 below, are:

* Right focused sequents $\Psi; \Delta \vdash [A^+]$ (where Δ is stable, containing only variable declarations $x{:}A^-\ lvl$ or $z{:}\langle A^+\rangle\ lvl$),
* Inversion sequents $\Psi; \Delta \vdash U$ (where Δ contains variable declarations $x{:}A^-\ lvl$, $z{:}\langle A^+\rangle\ lvl$ and inverting positive propositions A^+ and where U is either $A^+\ lvl$, $\langle A^-\rangle\ lvl$, or an inverting negative proposition A^-),
* Stable sequents, the special case of inversion sequents that contain no inverting positive propositions in Δ or inverting negative propositions in U.
* Left focused sequents $\Psi; \Theta\{[A^+]\} \vdash U$ (where Θ and U are stable – Θ contains only variable declarations $x{:}A^-\ lvl$ or $z{:}\langle A^+\rangle\ lvl$ and U is either $A^+\ lvl$ or $\langle A^-\rangle\ lvl$).

3.3.2 Polarized propositions

The propositions of ordered logic are fundamentally sorted into positive propositions A^+ and negative propositions A^-; both classes, and the inclusions between them, are shown in Figure 3.4. The positive propositions have a refinement, *permeable* propositions A^+_{pers}, that is analogous to the refinement discussed for linear logic in Section 2.5.4. There is also a more generous refinement, the *mobile* propositions, A^+_{eph}, for positive propositions that do not mention $\downarrow$ but that may mention ¡. We introduce atomic propositions p^+ that stand for arbitrary positive propositions, mobile atomic propositions p^+_{eph} that stand for arbitrary mobile propositions, and persistent p^+_{pers} that stand for arbitrary permeable propositions. We treat A^+_{ord} and p^+_{ord} as synonymous with A^+ and p^+, respectively, which allows us to generically refer to A^+_{lvl} and p^+_{lvl} in rules like η^+ and in

$$
\begin{array}{rl}
A^+ ::= & p^+ \mid p^+_{eph} \mid p^+_{pers} \mid \downarrow A^- \mid ¡A^- \mid !A^- \mid \mathbf{1} \mid A^+ \bullet B^+ \mid \mathbf{0} \mid A^+ \oplus B^+ \mid \exists a{:}\tau.A^+ \mid t \doteq_\tau s \\
A^+_{eph} ::= & p^+_{eph} \mid p^+_{pers} \mid ¡A^- \mid !A^- \mid \mathbf{1} \mid A^+_{eph} \bullet B^+_{eph} \mid \mathbf{0} \mid A^+_{eph} \oplus B^+_{eph} \mid \exists a{:}\tau.A^+_{eph} \mid t \doteq_\tau s \\
A^+_{pers} ::= & p^+_{pers} \mid !A^- \mid \mathbf{1} \mid A^+_{pers} \bullet B^+_{pers} \mid \mathbf{0} \mid A^+_{pers} \oplus B^+_{pers} \mid \exists a{:}\tau.A^+_{pers} \mid t \doteq_\tau s
\end{array}
$$

$$
\begin{array}{rl}
A^- ::= & p^- \mid p^-_{lax} \mid \uparrow A^+ \mid \bigcirc A^+ \mid A^+ \rightarrowtail B^- \mid A^+ \twoheadrightarrow B^- \mid \top \mid A^- \,\&\, B^- \mid \forall a{:}\tau.A^- \\
A^-_{lax} ::= & p^-_{lax} \mid \bigcirc A^+ \mid A^+ \rightarrowtail B^-_{lax} \mid A^+ \twoheadrightarrow B^-_{lax} \mid \top \mid A^-_{lax} \,\&\, B^-_{lax} \mid \forall a{:}\tau.A^-_{lax}
\end{array}
$$

Figure 3.4: Propositions of polarized OL_3

the statement of the identity expansion theorem.

Negative propositions also have a refinement, A^-_{lax}, for negative propositions that do not end in an upshift $\uparrow A^+$ or in a negative atomic proposition p^-. This is interesting as a formal artifact and there is very little overhead involved in putting it into our development, but the meaning of this syntactic class, as well as the meaning of right-permeable atomic propositions p^-_{lax}, is unclear. Certainly we do *not* want to include such propositions in our logical framework, as to do so would interfere with our development of traces as a syntax for partial proofs in Chapter 4.

The presentation of the exponentials, and the logic that we now present, emphasizes the degree to which the shifts $\uparrow$ and $\downarrow$ have much of the character of exponentials in a focused substructural logic. The upshift $\uparrow A^+$ is like an ordered variant of the lax truth $\bigcirc A^+$ that puts no constraints on the form of the succedent, and the downshift $\downarrow A^-$ is like an ordered variant of the persistent and linear exponentials $!A^-$ and $¡A^-$ that puts no constraints on the form of the context. This point is implicit in Laurent's dissertation [Lau02]. In that dissertation, Laurent defines the polarized LLP *without* the shifts $\uparrow$ and $\downarrow$, so that the only connection points between the polarities are the exponentials. Were it not for atomic propositions, the resulting logic would be more persistent than linear, a point we will return to in Section 3.7.

3.3.3 Derivations and proof terms

The multiplicative and exponential fragment of focused OL_3 is given in Figure 3.5, the additive fragment is given in Figure 3.6, and the first-order connectives are treated in Figure 3.7. We follow the convention of using matching constructs in the conclusions of rules and constructions in the premises with the exception of rules that are at the leaves, such as id^+ and $\doteq_R$, where we write out the matching condition as a premise.

These rules are all written with sequents of the form $\Psi; \Delta \vdash E : U$, where E is a *proof term* that corresponds to a derivation of that sequent. Just as sequent forms are divided into the right-focused, inverting, and left-focused sequents, we divide expressions into *values* V, derivations of right-focused sequents; *terms* N, derivations of inverting sequents; and *spines* Sp, derivations

Focus, identity, and atomic propositions

$$\frac{\Delta \vdash V : [A^+]}{\Delta \vdash V : A^+\ lvl}\ focus^*_R \qquad \frac{\Theta\{[A^-]\} \vdash Sp : U}{\Theta\{\!\{x{:}A^-\}\!\} \vdash x \cdot Sp : U}\ focus^*_L$$

$$\frac{\Theta\{z{:}\langle p^+_{lvl}\rangle\ lvl\} \vdash N : U}{\Theta\{\!\{p^+_{lvl}\}\!\} \vdash \langle z\rangle.N : U}\ \eta^+ \qquad \frac{\Delta\ matches\ z{:}\langle A^+\rangle}{\Delta \vdash z : [A^+]}\ id^+$$

$$\frac{\Delta \vdash N : \langle p^-_{lvl}\rangle\ lvl}{\Delta \vdash \langle N\rangle : p^-_{lvl}}\ \eta^- \qquad \frac{\Delta\ matches\ [A^-]}{\Delta \vdash \textsc{nil} : \langle A^-\rangle\ lvl}\ id^-$$

Shifts and modalities

$$\frac{\Delta \vdash N : A^-}{\Delta \vdash \downarrow N : [\downarrow A^-]}\ \downarrow_R \qquad \frac{\Theta\{x{:}A^-\ ord\} \vdash N : U}{\Theta\{\!\{\downarrow A^-\}\!\} \vdash \downarrow x.N : U}\ \downarrow_L$$

$$\frac{\Delta \vdash N : A^-}{\Delta\restriction_{eph} \vdash ¡N : [¡A^-]}\ ¡_R \qquad \frac{\Theta\{x{:}A^-\ eph\} \vdash N : U}{\Theta\{\!\{¡A^-\}\!\} \vdash ¡x.N : U}\ ¡_L$$

$$\frac{\Delta \vdash N : A^-}{\Delta\restriction_{pers} \vdash !N : [!A^-]}\ !_R \qquad \frac{\Theta\{x{:}A^-\ pers\} \vdash N : U}{\Theta\{\!\{!A^-\}\!\} \vdash !x.N : U}\ !_L$$

$$\frac{\Delta \vdash N : A^+\ true}{\Delta \vdash \uparrow N : \uparrow A^+}\ \uparrow_R \qquad \frac{\Theta\{A^+\} \vdash N : U}{\Theta\{\!\{[\uparrow A^+]\}\!\} \vdash \uparrow N : U}\ \uparrow_L$$

$$\frac{\Delta \vdash N : A^+\ lax}{\Delta \vdash \{N\} : \bigcirc A^+}\ \bigcirc_R \qquad \frac{\Theta\{A^+\} \vdash N : U}{\Theta\{\!\{[\bigcirc A^+]\}\!\} \vdash \{N\} : U\lfloor^{lax}}\ \bigcirc_L$$

Multiplicative connectives

$$\frac{\Delta\ matches\ \cdot}{\Delta \vdash () : [\mathbf{1}]}\ \mathbf{1}_R \qquad \frac{\Theta\{\cdot\} \vdash N : U}{\Theta\{\!\{\mathbf{1}\}\!\} \vdash ().N : U}\ \mathbf{1}_L$$

$$\frac{\Delta_1 \vdash V_1 : [A^+] \quad \Delta_2 \vdash V_2 : [B^+]}{\Delta_1, \Delta_2 \vdash V_1 \bullet V_2 : [A^+ \bullet B^+]}\ \bullet_R \qquad \frac{\Theta\{A^+, B^+\} \vdash N : U}{\Theta\{\!\{A^+ \bullet B^+\}\!\} \vdash \bullet N : U}\ \bullet_L$$

$$\frac{A^+, \Delta \vdash N : B^-}{\Delta \vdash \lambda^< N : A^+ \rightarrowtail B^-}\ \rightarrowtail_R \qquad \frac{\Delta_A \vdash V : [A^+] \quad \Theta\{[B]\} \vdash Sp : U}{\Theta\{\!\{\Delta_A, [A \rightarrowtail B]\}\!\} \vdash V^< Sp : U}\ \rightarrowtail_L$$

$$\frac{\Delta, A^+ \vdash N : B^-}{\Delta \vdash \lambda^> N : A^+ \twoheadrightarrow B^-}\ \twoheadrightarrow_R \qquad \frac{\Delta_A \vdash V : [A^+] \quad \Theta\{[B]\} \vdash Sp : U}{\Theta\{\!\{[A \twoheadrightarrow B], \Delta_A\}\!\} \vdash V^> Sp : U}\ \twoheadrightarrow_L$$

Figure 3.5: Multiplicative, exponential fragment of focused OL_3 (contexts Ψ suppressed)

$$\frac{}{\Theta\{\!\{\mathbf{0}\}\!\} \vdash \textsc{abort} : U}\ \mathbf{0}_L \qquad \frac{\Delta \vdash V : [A^+]}{\Delta \vdash \textsc{inl}(V) : [A^+ \oplus B^+]}\ \oplus_{R1} \qquad \frac{\Delta \vdash V : [B^+]}{\Delta \vdash \textsc{inr}(V) : [A^+ \oplus B^+]}\ \oplus_{R2}$$

$$\frac{\Theta\{A^+\} \vdash N_1 : U \quad \Theta\{B^+\} \vdash N_2 : U}{\Theta\{\!\{A^+ \oplus B^+\}\!\} \vdash [N_1, N_2] : U}\ \oplus_L$$

$$\frac{}{\Delta \vdash \top : \top}\ \top_R \qquad \frac{\Delta \vdash N_1 : A^- \quad \Delta \vdash N_2 : B^-}{\Delta \vdash N_1 \,\&\, N_2 : A^- \,\&\, B^-}\ \&_R$$

$$\frac{\Theta\{[A^-]\} \vdash Sp : U}{\Theta\{\!\{[A^- \,\&\, B^-]\}\!\} \vdash \pi_1; Sp : U}\ \&_{L1} \qquad \frac{\Theta\{[B^-]\} \vdash Sp : U}{\Theta\{\!\{[A^- \,\&\, B^-]\}\!\} \vdash \pi_2; Sp : U}\ \&_{L2}$$

Figure 3.6: Additive connectives of focused OL_3 (contexts Ψ suppressed)

$$\frac{\Psi \vdash t : \tau \quad \Psi; \Delta \vdash V : [[t/a]A^+]}{\Psi; \Delta \vdash t, V : [\exists a{:}\tau.A^+]}\ \exists_R \qquad \frac{\Psi, a{:}\tau; \Theta\{A^+\} \vdash N : U}{\Psi; \Theta\{\!\{\exists a{:}\tau.A^+\}\!\} \vdash a.N : U}\ \exists_L$$

$$\frac{\Psi, a{:}\tau; \Delta \vdash N : A^-}{\Psi; \Delta \vdash [a].N : \forall a{:}\tau.A^-}\ \forall_R \qquad \frac{\Psi \vdash t : \tau \quad \Psi; \Theta\{[[t/a]A^-]\} \vdash Sp : U}{\Psi; \Theta\{\!\{[\forall a{:}\tau.A^-]\}\!\} \vdash [t]; Sp : U}\ \forall_L$$

$$\frac{\Delta \textit{ matches } \cdot}{\Psi; \Delta \vdash \textsc{refl} : t \doteq_\tau t}\ \doteq_R \qquad \frac{\forall(\Psi' \vdash \sigma : \Psi).\quad \sigma t = \sigma s \quad \longrightarrow \quad \Psi'; \sigma\Theta\{\cdot\} \vdash \phi(\sigma) : \sigma U}{\Psi; \Theta\{\!\{t \doteq_\tau s\}\!\} \vdash \textsc{unif}\,(\mathsf{fn}\,\sigma \Rightarrow \phi(\sigma)) : U}\ \doteq_L$$

Figure 3.7: First-order connectives of focused OL_3

of left-focused sequents. The structure of values, terms, and spines is as follows:

$$\begin{array}{lll}
\text{Values} & V & ::= z \mid {\downarrow}N \mid {¡}N \mid !N \mid () \mid V_1 \bullet V_2 \mid \textsc{inl}(V) \mid \textsc{inr}(V) \mid t, V \mid \textsc{refl} \\
\text{Terms} & N & ::= V \mid x \cdot Sp \mid \langle z \rangle.N \mid \langle N \rangle \mid {\downarrow}x.N \mid {¡}x.N \mid !x.N \mid {\uparrow}N \mid \{N\} \\
& & \mid ().N \mid \bullet N \mid \lambda^< N \mid \lambda^> N \mid \textsc{abort} \mid [N_1, N_2] \mid \top \mid N_1 \,\&\, N_2 \mid a.N \mid [a].N \\
& & \mid \textsc{unif}\,(\mathsf{fn}\,\sigma \Rightarrow \phi(\sigma)) \\
\text{Spines} & Sp & ::= \textsc{nil} \mid {\uparrow}N \mid \{N\} \mid V^< Sp \mid V^> Sp \mid \pi_1; Sp \mid \pi_2; Sp \mid [t]; Sp
\end{array}$$

It is possible to take a "Curry-style" view of expressions as *extrinsically* typed, which means we consider both well-typed and ill-typed expressions; the well-typed expressions are then those for which the sequent $\Psi; \Delta \vdash E : U$ is derivable. However, we will take the "Church-style" view that expressions are intrinsically typed representatives of derivations: that is, $\Psi; \Delta \vdash E : U$ expresses that E is a derivation of the sequent $\Psi; \Delta \vdash U$. To justify this close correspondence, we require the inductive structure of expressions to be faithful to the inductive structure of proofs; this is one reason that we don't introduce the patterns that are common in other proof term assignments for focused logic [WCPW02, LZH08, Kri09]. (In Section 4.2.4, a limited syntax for patterns is introduced as part of the logical framework SLS.)

Proof terms for the left and right identity rules include angle brackets that reflect the notation for suspended propositions: $\langle N \rangle$ for η^- and $\langle z \rangle.N$ for η^+. We distinguish proof terms dealing

with existential quantifiers from those dealing with universal quantifiers in a nonstandard way by using square brackets for the latter: $[t]; Sp$ and $[a].N$ represent the left and right rules for universal quantification, whereas $a.N$ and t, V represent the left and right rules for existential quantification. Other than that, the main novelty in the proof term language and in Figures 3.5-3.7 is again the treatment of equality. We represent the proof term corresponding to the left rule for equality as $\textsc{unif}\,(\mathsf{fn}\,\sigma \Rightarrow \phi(\sigma))$, where $(\mathsf{fn}\,\sigma \Rightarrow \phi(\sigma))$ is intended to be a function from unifying substitutions σ to proof terms. This corresponds to the view of the $\doteq_L$ rule that takes the higher-order formulation seriously as a function, and we treat any proof term $\phi(\sigma))$ where σ is a unifying substitution as a subterm of $\textsc{unif}\,(\mathsf{fn}\,\sigma \Rightarrow \phi(\sigma))$.

There are two caveats to the idea that expressions are representatives of derivations. One caveat is that, in order for there to be an actual correspondence between expressions and terms, we need to annotate all variables with the judgment they are associated with, and we need to annotate the proof terms $\textsc{inr}(V)$, $\textsc{inl}(V)$, $\pi_1; Sp$, and $\pi_2; Sp$ with the type of the branch not taken. Pfenning writes these as superscripts [Pfe08], but we will follow Girard in leaving them implicit [GTL89]. The second caveat is that, because we do not explicitly represent the significant bookkeeping associated with matching constructs in proof terms, if $\Psi; \Delta \vdash E : U$, then $\Psi, a{:}\tau; \Delta, x{:}A^+\ \mathit{pers} \vdash E : U$ as well. Therefore, even given appropriate type annotations, when we say that some expression E is a derivation of $\Psi; \Delta \vdash U$, it is only *uniquely* a derivation of that sequent if we account for the implicit bookkeeping on contexts. It is likely that the first caveat can be largely dismissed by treating Figures 3.5-3.7 as bidirectional type system for proof terms. Addressing the second caveat will require a careful analysis of when the bookkeeping on contexts can be reconstructed, which we leave for future work.

The proof terms presented here mirror our formulation of a logical framework in the next chapter. Additionally, working on the level of proof terms allows for a greatly compressed presentation of cut admissibility and identity expansion that emphasizes the computational nature of these proofs: cut admissibility clearly generalizes the *hereditary substitution* operation in so-called *spine form* presentations of LF [CP02], and identity expansion is, computationally, a novel η-expansion property on proof terms. To be fair, much of this compression is due to neglecting the implicit bookkeeping associated with matching constructs, bookkeeping that must be made explicit in proofs like the cut admissibility theorem.

One theorem that takes place entirely at the level of this implicit bookkeeping is the admissible weakening lemma: if Δ' contains only persistent propositions and N is a derivation of $\Psi; \Delta \vdash U$, then N is also a derivation of $\Psi; \Delta, \Delta' \vdash U$. As usual, this proof can be established by straightforward induction on the structure of N.

3.3.4 Variable substitution

The first-order variables introduced by universal quantifiers (on the right) and existential quantifiers (on the left) are proper *variables* in the sense that the meaning of first-order variables is given by substitution [Har12, Chapter 1]. A sequent with free variables is thus a *generic* representative of all the sequents that can be obtained by plugging terms in for those free variables through the operation of substitution. This intuition is formalized by the variable substitution theorem, Theorem 3.4.

Theorem 3.4 (Variable substitution)**.** *If* $\Psi' \vdash \sigma : \Psi$ *and* $\Psi; \Delta \vdash U$*, then* $\Psi'; \sigma\Delta \vdash \sigma U$*.*

Proof. On the level of proof terms, we are given E, a expression corresponding to a derivation of $\Psi; \Delta \vdash U$; we are defining the operation σE, an expression corresponding to a derivation of $\Psi'; \sigma\Delta \vdash \sigma U$.

Propositional fragment For the exponential, multiplicative, and additive fragments, this operation is simple to define at the level of proof terms, and we will omit most of the cases: $\sigma(V_1 \bullet V_2) = \sigma V_1 \bullet \sigma V_2$, $\sigma(\downarrow x.N) = \downarrow x.\sigma N$, and so on. (Note that first-order variables a do not interact with variables x and z in the substructural context.) However, this compact notation does capture a great deal of complexity. In particular, it is important to emphasize that we need lemmas saying that variable substitution is compatible with all the context matching operations from Section 3.2. In full detail, these two simple cases would be:

- $\sigma(V_1 \bullet V_2) = \sigma V_1 \bullet \sigma V_2$

  We are given a proof of $\Psi; \Delta \vdash [A^+ \bullet B^+]$ that ends with the $\bullet_R$ rule; the subderivations are V_1, a derivation of $\Psi; \Delta_1 \vdash [A^+]$, and V_2, a derivation of $\Psi; \Delta_2 \vdash [B^+]$. Furthermore, we know that Δ matches Δ_1, Δ_2. We need a lemma that tells us that $\sigma\Delta$ matches $\sigma\Delta_1, \sigma\Delta_2$; then, by rule $\bullet_R$, it suffices to show that $\Psi'; \sigma\Delta_1 \vdash \sigma A^+$ (which we have by the induction hypothesis on σ and V_1) and that $\Psi'; \sigma\Delta_2 \vdash \sigma B^+$ (which we have by the induction hypothesis on σ and V_2).
- $\sigma(\downarrow x.N) = \downarrow x.\sigma N$

  We are given a proof of $\Psi; \Delta \vdash U$ that ends with $\downarrow_L$; the subderivation is N, a derivation of $\Psi; \Theta\{x{:}A^- \; ord\} \vdash U$. Furthermore, we know that Δ matches $\Theta\{\!\{\downarrow A^-\}\!\}$. We need a lemma that tells us that $\sigma\Delta$ matches $\sigma\Theta\{\!\{\downarrow \sigma A^-\}\!\}$; then, by rule $\downarrow_L$, it suffices to show $\Psi'; \sigma\Theta\{x{:}\sigma A^- \; ord\} \vdash \sigma U$ (which we have by the induction hypothesis on σ and N).

First-order fragment We will present variable substitution on the first-order fragment fully. Note the $\doteq_L$ rule in particular, which does *not* require an invocation of the induction hypothesis. The cases for the $\exists$ quantifier mimic the ones we give for the $\forall$ quantifier, and so the discussion of these cases is omitted.

- $\sigma(t, N) = (\sigma t, \sigma N)$
- $\sigma(a.Sp) = a.(\sigma, a/a)Sp$
- $\sigma(\text{REFL}) = \text{REFL}$
- $\sigma(\text{UNIF}\,(\mathsf{fn}\,\sigma'' \Rightarrow \phi(\sigma''))) = \text{UNIF}\,(\mathsf{fn}\,\sigma' \Rightarrow \phi(\sigma' \circ \sigma))$

  We are given a proof of $\Psi; \Delta \vdash U$ that ends with $\doteq_L$; we know that Δ matches $\Theta\{\!\{t \doteq s\}\!\}$, and the subderivation is ϕ, a function from substitutions $\Psi'' \vdash \sigma'' : \Psi$ that unify t and s to derivations of $\Psi''; \sigma''\Theta\{\cdot\} \vdash \sigma'' U$. We need a lemma that tells us that $\sigma\Delta$ matches $\sigma\Theta\{\!\{\sigma t \doteq \sigma s\}\!\}$; then, by rule $\doteq_L$, it suffices to show that for all $\Psi'' \vdash \sigma' : \Psi'$ that unify σt and σs, there exists a derivation of $\Psi''; \sigma'(\sigma\Theta)\{\cdot\} \vdash \sigma'(\sigma U)$, which is the same thing as a derivation of $\Psi''; (\sigma' \circ \sigma)\Theta\{\cdot\} \vdash (\sigma' \circ \sigma)U$. We have that $\Psi'' \vdash \sigma' \circ \sigma : \Psi$, and certainly $\sigma' \circ \sigma$ unifies t and s, so we can conclude by passing $\sigma' \circ \sigma$ to ϕ.

– $\sigma([a].N) = [a].(\sigma, a/a)N$

  We are given a proof of $\Psi; \Delta \vdash \forall a{:}\tau.A^-$ that ends with $\forall_R$; the subderivation is N, a derivation of $\Psi, a{:}\tau; \Delta \vdash A^-$. Because $\sigma(\forall a{:}\tau.A^-) = \forall a{:}\sigma\tau.(\sigma, a/a)A^-$, by rule $\forall_R$ it suffices to show $\Psi', a{:}\sigma\tau; \sigma\Delta \vdash (\sigma, a/a)A^-$.

  This is the same thing as $\Psi', a{:}\sigma\tau; (\sigma, a/a)\Delta \vdash (\sigma, a/a)A^-$; the result follows by the induction hypothesis on $(\sigma, a/a)$ and N.

– $\sigma([t]; Sp) = [\sigma t]; \sigma Sp$

  We are given a proof of $\Psi; \Delta \vdash U$ that ends with $\forall_L$; the subderivation is Sp, a derivation of $\Psi; \Theta\{[t/a]Sp\} \vdash U$. Furthermore, we know that Δ matches $\Theta\{\!\{[\forall a{:}\tau.A]\}\!\}$. We need a lemma that tells us that $\sigma\Delta$ matches $\sigma\Theta\{\!\{[\forall a{:}\tau.(\sigma, a/a)A^-]\}\!\}$; then, by rule $\forall_L$, it suffices to show $\Psi'; \sigma\Theta\{[[\sigma t/a]((\sigma, a/a)A^-)]\} \vdash \sigma U$.

  This is the same thing as $\Psi'; \sigma\Theta\{[\sigma([t/a]A^-)]\} \vdash \sigma U$; the result follows by the induction hypothesis on σ and Sp.

Note that, in the case for $\forall_R$, the substitution σ was applied to the first-order type τ as well as to the proposition A^-. This alludes to the fact that our first-order terms are dependently typed (Section 4.1). □

Given that we write the constructive content of the variable substitution theorem as σE, where E is an expression, we can also write Theorem 3.4 as an admissible rule in one of two ways, both with and without proof terms:

$$\frac{\Psi' \vdash \sigma : \Psi \quad \Psi; \Delta \vdash E : U}{\Psi'; \sigma\Delta \vdash \sigma E : \sigma U}\ \mathit{varsubst} \qquad \frac{\Psi' \vdash \sigma : \Psi \quad \Psi; \Delta \vdash U}{\Psi'; \sigma\Delta \vdash \sigma U}\ \mathit{varsubst}$$

We will tend towards the expression-annotated presentations, such as the one on the left, in this chapter.

3.3.5 Focal substitution

Both cut admissibility and identity expansion depend on the same focal substitution theorem that was considered for linear logic in Section 2.3.4. Both of these theorems use the compound matching construct $\Theta\{\!\{\Delta\restriction_{lvl}\}\!\}$, a pattern that will also be used in the proof of cut admissibility: Δ' matches $\Theta\{\!\{\Delta\restriction_{lvl}\}\!\}$ if $\Delta\restriction_{lvl}$ (which, again, is a shorthand way of saying Δ matches $\Delta\restriction_{lvl}$) and if Δ' matches $\Theta\{\!\{\Delta\}\!\}$.

Theorem 3.5 (Focal substitution)**.**

* *If* $\Psi; \Delta \vdash [A^+]$, $\Psi; \Theta\{z{:}\langle A^+\rangle\ lvl\} \vdash U$,
  and Ξ *matches* $\Theta\{\!\{\Delta\restriction_{lvl}\}\!\}$, *then* $\Psi; \Xi \vdash U$
* *If* $\Psi; \Delta \vdash \langle A^-\rangle\ lvl$, $\Psi; \Theta\{[A^-]\} \vdash U$,
  Ξ *matches* $\Theta\{\!\{\Delta\}\!\}$, *and* $U\lfloor^{lvl}$, *then* $\Psi; \Xi \vdash U$

Proof. The computational content of positive focal substitution is the substitution of a value V for a variable z in an expression E, written $[V/z]E$. As an admissible rule, positive focal substitution is represented as follows:

$$\frac{\Psi; \Delta \vdash V : [A^+] \quad \Psi; \Theta\{z{:}\langle A^+\rangle\ lvl\} \vdash E : U}{\Psi; \Theta\{\!\{\Delta\uparrow_{lvl}\}\!\} \vdash [V/z]E : U}\ subst^+$$

The proof of positive focal substitution proceeds by induction over the derivation E containing the suspended proposition. In the case where $E = z$, the derivation z concludes by right focusing on the proposition that we have a focused proof V of, so the result we are looking for is V.

The computational content of negative focal substitution is the substitution of a spine Sp out of an expression E. Sp represents a continuation, and the expression E is waiting on that continuation. As an admissible rule, negative focal substitution is represented as follows:

$$\frac{\Psi; \Delta \vdash E : \langle A^-\rangle\ lvl \quad \Psi; \Theta\{[A^-]\} \vdash Sp : U}{\Psi; \Theta\{\!\{\Delta\}\!\} \vdash [E]Sp : U\downharpoonleft^{lvl}}\ subst^-$$

The proof of negative focal substitution proceeds by induction over the derivation E containing the suspended proposition. In the case where $E = \text{NIL}$, the derivation NIL concludes by left focus on the proposition that we have a spine Sp for, so the result we are looking for is Sp. $\square$

Pay attention to the way compound matching constructs are being used. If we separate the substructural context out into its persistent, linear, and ordered constituents, the $subst^+$ rule can be seen as effectively expressing three admissible principles simultaneously:

* If $\Psi; \Gamma; \Delta; \Omega \vdash [A^+]$ and $\Psi; \Gamma; \Delta'; \Omega_L, \langle A^+\rangle, \Omega_R \vdash U$, then $\Psi; \Gamma; \Delta, \Delta'; \Omega_L, \Omega, \Omega_R \vdash U$.
* If $\Psi; \Gamma; \Delta; \cdot \vdash [A^+_{eph}]$ and $\Psi; \Gamma; \Delta', \langle A^+_{eph}\rangle; \Omega' \vdash U$, then $\Psi; \Gamma; \Delta, \Delta'; \Omega' \vdash U$.
* If $\Psi; \Gamma; \cdot; \cdot \vdash [A^+_{pers}]$ and $\Psi; \Gamma, \langle A^+_{pers}\rangle; \Delta'; \Omega' \vdash U$, then $\Psi; \Gamma; \Delta'; \Omega' \vdash U$.

In negative focal substitution, as in the leftist substitutions of cut admissibility, there is a corresponding use of $U\downharpoonleft^{lvl}$ to capture that we can use a proof of $\langle A^-\rangle\ true$ to discharge a hypothesis of $[A^-]$ in a proof of $C\ true$ or a proof of $C\ lax$, but that a proof of $\langle A^-_{lax}\rangle\ lax$ can only discharge a hypothesis of $[A^-_{lax}]$ in a proof of $C\ lax$.

3.4 Cut admissibility

It is a little wordy to say that, in a context or succedent, the only judgments involving suspensions are $(\langle p^+_{pers}\rangle\ pers)$, $(\langle p^+_{eph}\rangle\ eph)$, $(\langle p^+\rangle\ ord)$, $(\langle p^-\rangle\ true)$, and $(\langle p^-_{lax}\rangle\ lax)$, but this is a critical precondition of cut admissibility property for focused OL_3. We'll call contexts and succedents with this property *suspension-normal*.

Theorem 3.6 (Cut admissibility)**.** *For suspension-normal Ψ, A^+, A^-, Δ, Θ, Ξ, and U,*

1. *If $\Psi; \Delta \vdash [A^+]$, $\Psi; \Theta\{A^+\} \vdash U$,*
   and Ξ matches $\Theta\{\!\{\Delta\}\!\}$, then $\Psi; \Xi \vdash U$.
2. *If $\Psi; \Delta \vdash A^-$, $\Psi; \Theta\{[A^-]\} \vdash U$, Δ is stable,*
   and Ξ matches $\Theta\{\!\{\Delta\}\!\}$, then $\Psi; \Xi \vdash U$.

3. *If* $\Psi; \Delta \vdash A^+\ lvl$, $\Psi; \Theta\{A^+\} \vdash U$, Θ *and* U *are stable,* Ξ *matches* $\Theta\{\!\{\Delta\}\!\}$, *and* $U\lfloor^{lvl}$, *then* $\Psi; \Xi \vdash U$.
4. *If* $\Psi; \Delta \vdash A^-$, $\Psi; \Theta\{x{:}A^-\ lvl\} \vdash U$, Δ *is stable, and* Ξ *matches* $\Theta\{\!\{\Delta\lceil_{lvl}\}\!\}$, *then* $\Psi; \Xi \vdash U$

The four cases of cut admissibility (and their proof below) neatly codify an observation about the structure of cut admissibility proofs made by Pfenning in his work on structural cut elimination [Pfe00]. The first two parts of Theorem 3.6 are the home of the *principal cases* that decompose both derivations simultaneously – part 1 is for positive cut formulas and part 2 is for negative cut formulas. The third part contains all the *left commutative cases* that perform case analysis and induction only on the first given derivation, and the fourth part contains all the *right commutative cases* that perform case analysis and induction only on the second given derivation.

In Pfenning's work on structural cut elimination, this classification of cases was informal, but the structure of our cut admissibility proofs actually isolates the principal, left commutative, and right commutative cases into different parts of the theorem. This separation of cases is the reason why cut admissibility in a focused sequent calculus can use a more refined induction metric than cut admissibility in an unfocused sequent calculus. As noted previously in the proof of Theorem 2.4, the refined induction metric does away with the precondition, essential to Pfenning's proof of structural cut admissibility, that weakening and variable substitution preserve the size of derivations.

Before discussing the proof, it is worth noting that this theorem statement is already a sort of victory. It is an extremely simple statement of cut admissibility for a rather complex logic.

3.4.1 Optimizing the statement of cut admissibility

We will pick the cut admissibility proof from Chaudhuri's dissertation [Cha06] as a representative example of existing work on cut admissibility in focused logics. His statement of cut admissibility for linear logic has 10 parts, which are sorted into five groups. In order to extend his proof structure to handle the extra lax and mobile connectives in OL_3, we would need a dramatically larger number of cases. Furthermore, at a computational level, Chaudhuri's proof requires a lot of code duplication – that is, the proof of two different parts will frequently each require a case that looks essentially the same in both parts.

The structural focalization development in this chapter gives a compact proof of the completeness of focusing that is entirely free of code duplication. A great deal of simplification is due to the use of the matching constructs $\Theta\{\!\{\Delta\lceil_{lvl}\}\!\}$ and $U\lfloor^{lvl}$. Without that notation, part 3 would split into two parts for $true$ and lax and part 4 would split into three parts for ord, eph, and $pers$. The fifth part of the cut admissibility theorem in Section 2.3.6 (Theorem 2.4), which is computationally a near-duplicate of the fourth part of the same theorem, is due to the lack of this device.

Further simplification is due to defining right-focused, inverting, and left-focused sequents as refinements of general sequents $\Psi; \Delta \vdash U$. Without this approach, the statement of part 3 must be split into two parts (for substituting into terms and spines) and the statement of part 4 must be split into three parts (for substituting into values, terms, and spines). Without either of the

aforementioned simplifications, we would have 15 parts in the statement of Theorem 3.6 instead of four and twice as many cases that needed to be written down and checked.

Picking a fixed inversion strategy prevents us from having to prove the tedious, quadratically large confluence theorem discussed for linear logic in Section 2.3.8. This confluence theorem is certainly true, and we might want to prove it for any number of reasons, but it is interesting that we can avoid it altogether in our current development. A final improvement in our theorem statement is very subtle: insofar as our goal is to give a short proof of the completeness of focusing that avoids redundancy, the *particular* fixed inversion strategy we choose matters. The proof of Theorem 2.4 duplicates many right commutative cases in both part 1 and part 4 (which map directly onto parts 1 and 4 of Theorem 3.6 above). Our system prioritizes the inversion of positive formulas on the left over the inversion of negative formulas on the right. If we made the opposite choice, as Chaudhuri's system does, then this issue would remain, resulting in code duplication. We get a lot of mileage out of the fact that if $\Xi = \Theta\{A^+\}$ then A^+ unambiguously refers to the left-most proposition in Ξ, and this invariant would no longer be possible to maintain in the proof of cut admissibility if we prioritized inversion of negative propositions on the right.

3.4.2 Proof of cut admissibility, Theorem 3.6

The proof proceeds by lexicographic induction. In parts 1 and 2, the type gets smaller in every call to the induction hypothesis. In part 3, the induction hypothesis is only ever invoked on the same type A^+, and every invocation of the induction hypothesis is either to part 1 (smaller part number) or to part 3 (same part number, first derivation is smaller). Similarly, in part 4, the induction hypothesis is only invoked at the same type A^-, and every invocation of the induction hypothesis is either to part 2 (smaller part number) or to part 4 (same part number, second derivation is smaller).

The remainder of this section will cover each of the four parts of this proof in turn. Most of the theorem will be presented at the level of proof terms, but for representative cases we will discuss what the manipulation of proof terms means in terms of sequents and matching constructs. The computational content of parts 1 and 2 is *principal substitution*, written as $(V \circ N)^{A^+}$ and $(N \circ Sp)^{A^-}$ respectively, the computational content of part 3 is *leftist substitution*, written as $[\![E]\!]^{A^+} N$, and the computational content of part 4 is *rightist substitution*, written as $[\![M/x]\!]^{A^-} E$.

In many cases, we discuss the necessity of constructing certain contexts or frames; in general, we will state the necessary properties of these constructions without detailing the relatively straightforward process of constructing them.

Positive principal substitution

Positive principal substitution encompasses half the *principal cuts* from Pfenning's structural cut admissibility proof – the principal cuts where the principal cut formula is positive. The constructive content of this part is a function $(V \circ N)^{A^+}$ that normalizes a value against a term. Induction is on the structure of the positive type. The admissible rule associated with principal

positive substitution is cut^+.

$$\frac{\Psi;\Delta \vdash V : [A^+] \quad \Psi;\Theta\{A^+\} \vdash N : U}{\Psi;\Theta\{\!\{\Delta\}\!\} \vdash (V \circ N)^{A^+} : U}\ cut^+$$

We have to be careful, especially in the positive principal substitution associated with the type $A^+ \bullet B^+$, to maintain the invariant that, in an unstable context, we only ever consider the *leftmost* inverting positive proposition.

In most of these cases, one of the givens is that $\Theta\{A^+\}$ matches $\Theta'\{\!\{A^+\}\!\}$ for some Θ'. Because this implies that $\Theta = \Theta'$, we take the equality for granted rather than mentioning and reasoning explicitly about the premise every time.

- $(z \circ \langle z' \rangle.N_1)^{p^+_{lvl}} = [z/z']N_1$

  We must show $\Psi;\Xi \vdash U$, where

  - Δ matches $z{:}\langle p^+_{lvl}\rangle$,
  - N_1 is a derivation of $\Psi;\Theta\{z'{:}\langle p^+_{lvl}\rangle\ lvl\} \vdash U$,
  - and Ξ matches $\Theta\{\!\{\Delta\}\!\}$.

  Because Δ is suspension-normal, we can derive $\Psi;\Delta \vdash [p^+_{lvl}]$ by id^+, and Ξ matches $\Theta\{\!\{\Delta\restriction_{lvl}\}\!\}$. Therefore, the result follows by focal substitution on z and N_1.

- $(\downarrow M \circ \downarrow x.N_1)^{\downarrow A^-} = [\![M/x]\!]^{A^-} N_1$
- $(¡M \circ ¡x.N_1)^{¡A^-} = [\![M/x]\!]^{A^-} N_1$
- $(!M \circ !x.N_1)^{!A^-} = [\![M/x]\!]^{A^-} N_1$

  We must show $\Psi;\Xi \vdash U$, where

  - Δ matches $\Delta\restriction_{pers}$, M is derivation of $\Psi;\Delta \vdash A^-$,
  - N_1 is a derivation of $\Psi;\Theta\{x{:}A^-\ pers\} \vdash U$,
  - and Ξ matches $\Theta\{\!\{\Delta\}\!\}$.

  Ξ matches $\Theta\{\!\{\Delta\restriction_{pers}\}\!\}$ and Δ is stable (it was in a focused sequent $\Psi;\Delta \vdash !M : [!A^-]$), so the result follows by part 4 of cut admissibility on N_1 and M.

- $(() \circ ().N_1)^{\mathbf{1}} = N_1$
- $((V_1 \bullet V_2) \circ (\bullet N_1))^{A^+ \bullet B^+} = (V_2 \circ (V_1 \circ N_1)^{A^+})^{B^+}$

  We must show $\Psi;\Xi \vdash U$, where

  - Δ matches Δ_1, Δ_2,
    V_1 is a derivation of $\Psi;\Delta_1 \vdash [A^+]$, V_2 is a derivation of $\Psi;\Delta_2 \vdash [B^+]$,
  - N_1 is a derivation of $\Psi;\Theta\{A^+, B^+\} \vdash U$,
  - and Ξ matches $\Theta\{\!\{\Delta\}\!\}$.

  We can to construct a frame Θ_B such that $\Theta\{A^+, B^+\} = \Theta_B\{A^+\}$; we're just exchanging the part in the frame with the part not in the frame. We can also construct a second frame, Θ_A, such that 1) Ξ matches $\Theta_A\{\!\{\Delta_2\}\!\}$ and 2) $\Theta_A\{B^+\}$ matches $\Theta_B\{\!\{\Delta_1\}\!\}$.

  Because $\Theta_A\{B^+\}$ matches $\Theta_B\{\!\{\Delta_1\}\!\}$, by the induction hypothesis on V_1 and N_1 we have $(V_1 \circ N_1)^{A^+}$, a derivation of $\Psi;\Theta_A\{B^+\} \vdash U$.

Because Ξ matches $\Theta_A\{\!\{\Delta_2\}\!\}$, by the induction hypothesis on V_2 and $(V_1 \circ N_1)^{A^+}$, we have a derivation of $\Psi; \Xi \vdash U$ as required.

- $(\textsc{inl}(V_1) \circ [N_1, N_2])^{A^+ \oplus B^+} = (V_1 \circ N_1)^{A^+}$
- $(\textsc{inr}(V_2) \circ [N_1, N_2])^{A^+ \oplus B^+} = (V_2 \circ N_2)^{B^+}$
- $(t, V_1 \circ a.N_1)^{\exists a{:}\tau.A^+} = (V_1 \circ [t/a]N_1)^{[t/a]A^+}$

  We must show $\Psi; \Xi \vdash U$, where

  - $\Psi \vdash t : \tau$, V_1 is a derivation of $\Psi; \Delta \vdash [[t/a]A^+]$,
  - N_1 is a derivation of $\Psi, a{:}\tau; \Theta\{A^+\} \vdash U$,
  - and Ξ matches $\Theta\{\!\{\Delta\}\!\}$.

  By variable substitution on $[t/a]$ and N_1, we have a derivation $[t/a]N_1$ of $\Psi; \Theta\{[t/a]A^+\} \vdash U$. We count $[t/a]A^+$ as being a smaller formula than $\exists a{:}\tau.A^+$, so by the induction hypothesis on V_1 and $[t/a]N_1$, we get a derivation of $\Psi; \Xi \vdash U$ as required.

- $(\textsc{refl} \circ \textsc{unif}\,(\mathsf{fn}\,\sigma \Rightarrow \phi(\sigma)))^{t \doteq t} = \phi(\mathsf{id})$

  We must show $\Psi; \Xi \vdash U$, where

  - Δ matches $\cdot$,
  - ϕ is a function from substitutions $\Psi' \vdash \sigma : \Psi$ that unify t and t to derivations of $\Psi; \Theta\{\cdot\} \vdash U$,
  - and Ξ matches $\Theta\{\!\{\Delta\}\!\}$.

  We simply apply the identity substitution to ϕ to obtain a derivation of $\Psi; \Theta\{\cdot\} \vdash U$. Note that this is not quite the derivation of $\Psi; \Xi \vdash U$ that we need; we need an exchange-like lemma that, given a derivation of $\Psi; \Theta\{\cdot\} \vdash U$ and the fact that Ξ matches $\Theta\{\!\{\cdot\}\!\}$, we can get a proof of $\Psi; \Xi \vdash U$ as we require.

Negative principal substitution

Negative principal substitution encompass all the *principal cuts* from Pfenning's structural cut admissibility proof for which the principal formula is negative. The constructive content of this part is a function $(N \circ Sp)^{A^-}$ that normalizes a term against a spine; a similar function appears as *hereditary reduction* in presentations of hereditary substitution for LF [WCPW02]. Induction is on the structure of the negative type. The admissible rule associated with negative principal substitution is cut^-:

$$\frac{\Psi; \Delta \vdash N : A^- \quad \Psi; \Theta\{[A^-]\} \vdash Sp : U \quad \Delta\ stable_L}{\Psi; \Theta\{\!\{\Delta\}\!\} \vdash (N \circ Sp)^{A^-} : U}\ cut^-$$

- $(\langle N \rangle \circ \textsc{nil})^{p^-_{lvl}} = N$

  We must show $\Psi; \Xi \vdash U$, where

  - N is a derivation of $\Psi; \Delta \vdash \langle p^-_{lvl} \rangle\ lvl$
  - $\Theta\{[p^-_{lvl}]\}$ matches $[p^-_{lvl}]$, $U = \langle p^-_{lvl} \rangle\ lvl'$,

- and Ξ matches $\Theta\{\!\{\Delta\}\!\}$.

Because U is suspension-normal, $lvl = lvl'$. A derivation of $\Psi; \Delta \vdash \langle p^-_{lvl} \rangle\ lvl$ is not quite a proof of $\Psi; \Xi \vdash U$, so we need an exchange-like lemma that we can get one from the other.

– $(\uparrow N \circ \uparrow M)^{\uparrow A^+} = [\![N]\!]^{A^+} M$
– $(\{N\} \circ \{M\})^{\bigcirc A^+} = [\![N]\!]^{A^+} M$

We must show $\Psi; \Xi \vdash U$, where

- N is a derivation of $\Psi; \Delta \vdash A^+\ lax$,
- $\Theta\{\bigcirc A^+\}$ matches $\Theta'\{\!\{\bigcirc A^+\}\!\}$, $U\lfloor^{lax}$, Θ' and U are stable, M is a derivation of $\Psi; \Theta'\{A^+\} \vdash U$,
- and Ξ matches $\Theta\{\!\{\Delta\}\!\}$.

Ξ matches $\Theta'\{\!\{\Delta\}\!\}$, so the result follows by part 3 of cut admissibility on N and M.

– $((\lambda^< N) \circ (V^< Sp))^{A^+ \rightarrowtail B^-} = ((V \circ N)^{A^+} \circ Sp)^{B^-}$
– $((\lambda^> N) \circ (V^> Sp))^{A^+ \twoheadrightarrow B^-} = ((V \circ N)^{A^+} \circ Sp)^{B^-}$

We must show $\Psi; \Xi \vdash U$, where

- N is a derivation of $\Psi; \Delta, A^+ \vdash B^-$, Δ is stable (by the fixed inversion invariant – we only invert on the right when there is no further inversion to do on the left),
- $\Theta\{[A^+ \twoheadrightarrow B^-]\}$ matches $\Theta'\{\!\{[A^+ \twoheadrightarrow B^-], \Delta_A\}\!\}$, V is a derivation of $\Psi; \Delta_A \vdash [A^+]$, Sp is a derivation of $\Psi; \Theta'\{[B^-]\} \vdash U$,
- and Ξ matches $\Theta\{\!\{\Delta\}\!\}$.

We can simultaneously view the construction Δ, A^+ as a frame Θ_Δ such that $\Theta_\Delta\{A^+\} = \Delta, A^+$. Note that this is only possible to do because Δ is stable; if there were a non-stable proposition in Δ, the fixed inversion invariant would not permit us to frame off the right-most proposition A^+.

We next construct a context Δ'_A that matches $\Theta_\Delta\{\!\{\Delta_A\}\!\}$ (and also Δ, Δ_A viewed as a matching construct), while simultaneously Ξ matches $\Theta'\{\!\{\Delta'_A\}\!\}$.

By the part 1 of cut admissibility on V and N, we have $(V \circ N)^{A^+}$, a derivation of $\Psi; \Delta'_A \vdash B^-$, and the result then follows by the induction hypothesis on $(V \circ N)^{A^+}$ and Sp.

– $((N_1 \mathbin{\&} N_2) \circ (\pi_1; Sp))^{A^- \& B^-} = (N_1 \circ Sp)^{A^-}$
– $((N_1 \mathbin{\&} N_2) \circ (\pi_2; Sp))^{A^- \& B^-} = (N_2 \circ Sp)^{A^-}$
– $(([a].N) \circ ([t]; Sp))^{\forall a:\tau.A^-} = ([t/a]N \circ Sp)^{[t/a]A^-}$

Leftist substitution

In focal substitution, the positive case corresponds to our usual intuitions about substitution and the negative case is strange. In cut admissibility, the situation is reversed: rightist substitutions (considered in Section 3.4.2 below), associated with negative principal cut formals, look like normal substitutions, and the leftist substitutions, considered here, are strange, as they break

apart the expression that proves A^+ rather than the term where A^+ appears in the context.

Leftist substitutions encompass all the *left commutative cuts* from Pfenning's structural cut admissibility proof. The constructive content of leftist substitution is a function $[\![E]\!]M$; we say we are *substituting M out of E*. Induction is on the first subterm, as we crawl through E looking for places where focus takes place on the right. The admissible rule associated with leftist substitution is $lcut$:

$$\frac{\Psi;\Delta \vdash E : A^+\ lvl \quad \Psi;\Theta\{A^+\} \vdash M : U \quad \Theta\ stable_L \quad U\ stable_R}{\Psi;\Theta\{\!\{\Delta\}\!\} \vdash [\![E]\!]^{A^+} M : U\lfloor^{lvl}}\ lcut$$

Except for the case where the first given derivation ends in the rule $focus_R$, every case of this theorem involves a left rule. The general pattern for these cases is that Ξ matches $\Theta\{\!\{\Delta\}\!\}$ and Δ matches $\Theta_B\{\!\{x{:}T\ ord\}\!\}$. Θ and Θ_B have the same persistent variables but distinct ephemeral and ordered variables, and we must construct a frame $\Theta{\circ}\Theta_B$ that is effectively the composition of Θ and Θ_B. In cases that we discuss in detail, necessary properties of this composition frame are stated but not proven.

Substitution out of terms

- $[\![V]\!]^{A^+} M = (V \circ M)^{A^+}$

  We must show $\Psi;\Xi \vdash U$, where
  - V is a derivation of $\Psi;\Delta \vdash [A^+]$,
  - M is a derivation of $\Psi;\Theta\{A^+\} \vdash U$,
  - Ξ matches $\Theta\{\!\{\Delta\}\!\}$, and $U\lfloor^{lvl}$

  The result follows from part 1 of cut admissibility on V and M.
- $[\![x \cdot Sp]\!]^{A^+} M = x \cdot ([\![Sp]\!]^{A^+} M)$

  We must show $\Psi;\Xi \vdash U$, where
  - Δ matches $\Theta_B\{\!\{x{:}B^-\}\!\}$, Sp is a derivation of $\Psi;\Theta_B\{[B^-]\} \vdash A^+\ lvl$,
  - M is a derivation of $\Psi;\Theta\{A^+\} \vdash U$,
  - Ξ matches $\Theta\{\!\{\Delta\}\!\}$, and $U\lfloor^{lvl}$.

  Ξ matches $(\Theta{\circ}\Theta_B)\{\!\{x{:}B^-\}\!\}$ and $(\Theta{\circ}\Theta_B)\{[B^-]\}$ matches $\Theta'\{\!\{\Theta_B\{[B^-]\}\}\!\}$. By the induction hypothesis on Sp and M we have $\Psi;(\Theta{\circ}\Theta_B)\{[B^-]\} \vdash U$, and the required result then follows from rule $focus_L$.
- $[\![\langle z\rangle.N]\!]^{A^+} M = \langle z\rangle.([\![N]\!]^{A^+} M)$
- $[\![\downarrow x.N]\!]^{A^+} M = \downarrow x.([\![N]\!]^{A^+} M)$
- $[\![¡x.N]\!]^{A^+} M = ¡x.([\![N]\!]^{A^+} M)$
- $[\![!x.N]\!]^{A^+} M = !x.([\![N]\!]^{A^+} M)$

  We must show $\Psi;\Xi \vdash U$, where
  - Δ matches $\Theta_B\{\!\{!B^-\}\!\}$, N is a derivation of $\Psi;\Theta_B\{x{:}B^-\ pers\} \vdash A^+\ lvl$,
  - M is a derivation of $\Psi;\Theta\{A^+\} \vdash U$,

- Ξ matches $\Theta\{\!\{\Delta\}\!\}$, and $U\lfloor^{lvl}$.

We can construct a Θ' such that $\Theta'\{A^+\} = (\Theta\{A^+\}, x{:}B^-\ pers)$. By admissible weakening, M is a derivation of $\Psi; \Theta'\{A^+\} \vdash U$, too.

Ξ matches $(\Theta \circ \Theta_B)\{\!\{!B^-\}\!\}$ and $(\Theta \circ \Theta_B)\{x{:}B^-\ pers\}$ matches $\Theta'\{\!\{\Theta_B\{x{:}B^-\ pers\}\}\!\}$. By the induction hypothesis on N and M we have $\Psi; (\Theta \circ \Theta_B)\{x{:}B^-\ pers\} \vdash U$, and the required result then follows from rule $!_L$.

- $[\![\bullet N]\!]^{A^+} M = \bullet([\![N]\!]^{A^+} M)$
- $[\![\textsc{abort}]\!]^{A^+} M = \textsc{abort}$
- $[\![[N_1, N_2]]\!]^{A^+} M = [([\![N_1]\!]^{A^+} M), ([\![N_2]\!]^{A^+} M)]$
- $[\![a.N]\!]^{A^+} M = a.([\![N]\!]^{A^+} M)$

  We must show $\Psi; \Xi \vdash U$, where

  - Δ matches $\Theta_B\{\!\{\exists a{:}\tau.B^+\}\!\}$, N is a derivation of $\Psi, a{:}\tau; \Theta_B\{B^+\} \vdash A^+$,
  - M is a derivation of $\Psi; \Theta\{A^+\} \vdash U$,
  - Ξ matches $\Theta\{\!\{\Delta\}\!\}$, and $U\lfloor^{lvl}$.

  Ξ matches $(\Theta \circ \Theta_B)\{\!\{\exists a{:}\tau.B^+\}\!\}$ and $(\Theta \circ \Theta_B)\{B^+\}$ matches $\Theta\{\!\{\Theta_B\{B^+\}\}\!\}$. By variable weakening, M is also a derivation of $\Psi, a{:}\tau; \Theta\{A^+\} \vdash U$, so by the induction hypothesis on N and M we have $\Psi, a{:}\tau; (\Theta \circ \Theta_B)\{B^+\} \vdash U$, and the required result then follows from rule $\exists_L$.

- $[\![\textsc{unif}\,(\mathsf{fn}\,\sigma \Rightarrow \phi(\sigma))]\!]^{A^+} M = \textsc{unif}\,(\mathsf{fn}\,\sigma \Rightarrow [\![\phi(\sigma)]\!]^{\sigma A^+}(\sigma M))$

  We must show $\Psi; \Xi \vdash U$, where

  - Δ matches $\Theta_B\{\!\{t \doteq s\}\!\}$, ϕ is a function from substitutions $\Psi' \vdash \sigma : \Psi$ that unify t and s to derivations of $\Psi'; \sigma\Theta_B\{\cdot\} \vdash \sigma A^+$,
  - M is a derivation of $\Psi; \Theta\{A^+\} \vdash U$,
  - Ξ matches $\Theta\{\!\{\Delta\}\!\}$, and $U\lfloor^{lvl}$.

  Ξ matches $(\Theta \circ \Theta_B)\{\!\{t \doteq s\}\!\}$, and for any substitution σ, $\sigma U\lfloor^{lvl}$ and $\sigma(\Theta \circ \Theta_B)\{\cdot\}$ matches $\sigma\Theta\{\!\{\sigma\Theta_B\{\cdot\}\}\!\}$. By rule $\doteq_L$, it suffices to show that, given an arbitrary substitution $\Psi' \vdash \sigma : \Psi$, there is a derivation of $\Psi'; \sigma(\Theta \circ \Theta_B)\{\cdot\} \vdash \sigma U$.

  By applying σ to ϕ, we get $\phi(\sigma)$, a derivation of $\Psi'; \sigma\Theta_B\{\cdot\} \vdash \sigma A^+$. We treat σA^+ as having the same size as A^+, and the usual interpretation of higher-order derivations is that $\phi(\sigma)$ is a subderivation of ϕ, so $\phi(\sigma)$ can be used to invoke the induction hypothesis. From variable substitution, we get σM, a derivation of $\Psi'; \sigma\Theta\{\sigma A^+\} \vdash \sigma U$, and then the result follows by the induction hypothesis on $\phi(\sigma)$ and σM.

Substitution out of spines

- $[\![\uparrow N]\!]^{A^+} M = \uparrow([\![N]\!]^{A^+} M)$
- $[\![\{N\}]\!]^{A^+} M = \{[\![N]\!]^{A^+} M\}$

  We must show $\Psi; \Xi \vdash U$, where

- Δ matches $\Theta_B\{\!\{\bigcirc B^+\}\!\}$, $(A^+\ lvl)\downharpoonleft^{lax}$,
  N is a derivation of $\Psi; \Theta_B\{A^+\} \vdash U_A$
- M is a derivation of $\Psi; \Theta\{A^+\} \vdash U$,
- Ξ matches $\Theta\{\!\{\Delta\}\!\}$, and U' matches $U\downharpoonleft^{lvl}$.

Because $(A^+\ lvl)\downharpoonleft^{lax}$ and $U\downharpoonleft^{lvl}$, we can conclude that $U\downharpoonleft^{lax}$.

Ξ matches $(\Theta\circ\Theta_B)\{\!\{\bigcirc B^+\}\!\}$ and $(\Theta\circ\Theta_B)\{B^+\}$ matches $\Theta\{\!\{\Theta_B\{B^+\}\}\!\}$. By the induction hypothesis on N and M we have $\Psi; (\Theta\circ\Theta_B)\{B^+\} \vdash U$, and the result follows by rule $\bigcirc_L$.

– $[\![V^< Sp]\!]^{A^+} M = V^<([\![Sp]\!]^{A^+} M)$
– $[\![V^> Sp]\!]^{A^+} M = V^>([\![Sp]\!]^{A^+} M)$

  We must show $\Psi; \Xi \vdash U$, where

  - Δ matches $\Theta_B\{\!\{[B_1^+ \twoheadrightarrow B_2^-], \Delta_B\}\!\}$, V is a derivation of $\Psi; \Delta_B \vdash [B_1^+]$,
    Sp is a derivation of $\Psi; \Theta_B\{[B_2^-]\} \vdash A^+\ lvl$,
  - M is a derivation of $\Psi; \Theta\{A^+\} \vdash U$,
  - Ξ matches $\Theta\{\!\{\Delta\}\!\}$, and $U\downharpoonleft^{lvl}$.

  Ξ matches $(\Theta\circ\Theta_B)\{\!\{[B_1^+ \twoheadrightarrow B_2^-], \Delta_B\}\!\}$ and $(\Theta\circ\Theta_B)\{[B_2^-]\}$ matches $\Theta\{\!\{\Theta_B\{[B_2^-]\}\}\!\}$. By invoking the induction hypothesis to substitute M out of Sp, we have $[\![Sp]\!]^{A^+} M$, which is a derivation of $\Psi; (\Theta\circ\Theta_B)\{[B_2^-]\} \vdash U$. The required result follows by rule $\twoheadrightarrow_L$ on V and $[\![Sp]\!]^{A^+} M$.
– $[\![\pi_1; Sp]\!]^{A^+} M = \pi_1; ([\![Sp]\!]^{A^+} M)$
– $[\![\pi_2; Sp]\!]^{A^+} M = \pi_2; ([\![Sp]\!]^{A^+} M)$
– $[\![[t]; Sp]\!]^{A^+} M = [t]; ([\![Sp]\!]^{A^+} M)$

Rightist substitution

Rightist substitutions encompass all the *right commutative cuts* from Pfenning's structural cut admissibility proof. The constructive content of this part is a function $[\![M/x]\!]^{A^-} E$; we say we are *substituting M into E*. Induction is on the second subterm, as we crawl through E looking for places where x is mentioned. The admissible rule associated with rightist substitution is $rcut$:

$$\frac{\Psi; \Delta \vdash M : A^- \quad \Psi; \Theta\{x{:}A^-\ lvl\} \vdash E : U \quad \Delta\ stable_L}{\Psi; \Theta\{\!\{\Delta\restriction_{lvl}\}\!\} \vdash [\![M/x]\!]^{A^-} E : U}\ rcut$$

A unique aspect of the right commutative cuts is that the implicit bookkeeping on contexts matters to the computational behavior of the proof: when we deal with multiplicative connectives like $A^+ \bullet B^+$ and $A^+ \rightarrowtail B^+$ under focus, we actually must consider that the variable x that we're substituting for can end up in only one specific branch of the proof (if x is associated with a judgment $A^-\ ord$ or $A^-\ eph$) or in both branches of the proof (if x is associated with a judgment $x{:}A^-\ pers$). The computational representation of these cases looks nondeterministic, but it is actually determined by the annotations and bookkeeping that we don't write down as part of the proof term. This is a point that we return to in Section 3.8.

For cases involving left rules, the general pattern is that Ξ matches $\Theta\{\!\{\Delta\restriction_{lvl}\}\!\}$ and the action of the left rule, when we read it bottom-up, is observe that $\Theta\{x{:}A^- \; lvl\}$ matches $\Theta'\{\!\{y{:}T \; ord\}\!\}$ in its conclusion and constructs $\Theta'\{\!\{y{:}T' \; lvl'\}\!\}$ in its premise(s). Effectively, we need to abstract a *two*-hole function (call it Γ) from Ξ. One hole – the place where x is – is defined by the frame Θ: morally, $\Theta = \lambda\Delta_B.\Gamma(x{:}A^- \; lvl)(\Delta_B)$. The other hole – the place where y is – is defined by Θ': morally, $\Theta' = \lambda\Delta_A.\Gamma(\Delta_A)(y{:}T \; ord)$. However, we cannot directly represent these functions due to the need to operate around matching constructs. Instead, we construct Θ_Δ to represent the frame that is morally $\lambda\Delta_B.\Gamma(\Delta)(\Delta_B)$, and $\Theta_{T'}$ to represent the frame that is morally $\lambda\Delta_A.\Gamma(\Delta_A)(y{:}T' \; lvl')$. As before, in cases that we discuss in detail, necessary properties of these two frames are stated but not proven.

Substitution into values

- $[\![M/x]\!]^{A^-} z = z$
- $[\![M/x]\!]^{A^-} (\downarrow N) = \downarrow([\![M/x]\!]^{A^-} N)$
- $[\![M/x]\!]^{A^-} (¡N) = ¡([\![M/x]\!]^{A^-} N)$
- $[\![M/x]\!]^{A^-} (!N) = !([\![M/x]\!]^{A^-} N)$

  We must show $\Psi; \Xi \vdash [!B^-]$, where
  - M is a derivation of $\Psi; \Delta \vdash A^-$,
  - $\Theta\{x{:}A^- \; lvl\}$ matches $\Delta'\restriction_{pers}$, N is a derivation of $\Psi; \Delta' \vdash B^-$,
  - and Ξ matches $\Theta\{\!\{\Delta\restriction_{lvl}\}\!\}$.

  Because $\Theta\{x{:}A^- \; lvl\}$ matches $\Delta'\restriction_{pers}$ and Ξ matches $\Theta\{\!\{\Delta\restriction_{lvl}\}\!\}$, we can conclude that there exists a Θ' such that $\Delta' = \Theta'\{x{:}A^- \; lvl\}$ and also that Ξ matches $\Xi\restriction_{pers}$.

  By the induction hypothesis on M and N, we have a derivation of $\Psi; \Xi \vdash B^-$, and the result follows by rule $!_R$.

- $[\![M/x]\!]^{A^-} () = ()$

  We must show $\Psi; \Xi \vdash [\mathbf{1}]$, where
  - M is a derivation of $\Psi; \Delta \vdash A^-$,
  - $\Theta\{x{:}A^- \; lvl\}$ matches $\cdot$,
  - and Ξ matches $\Theta\{\!\{\Delta\restriction_{lvl}\}\!\}$.

  Because $\Theta\{x{:}A^- \; lvl\}$ matches $\cdot$, it must be the case that $lvl = pers$, and so Ξ matches $\cdot$ as well. The result follows by rule $\mathbf{1}_R$.

- $[\![M/x]\!]^{A^-} (V_1 \bullet V_2) =$

  $([\![M/x]\!]^{A^-} V_1) \bullet V_2$ *(if x is in V_1's context but not V_2's)*
  $V_1 \bullet ([\![M/x]\!]^{A^-} V_2)$ *(if x is in V_2's context but not V_1's)*
  $([\![M/x]\!]^{A^-} V_1) \bullet ([\![M/x]\!]^{A^-} V_2)$ *(if x is in both V_1 and V_2's contexts)*

  We must show $\Psi; \Xi \vdash [B_1^+ \bullet B_2^+]$, where
  - M is a derivation of $\Psi; \Delta \vdash A^-$,

- $\Theta\{x{:}A^- \; lvl\}$ matches Δ_1, Δ_2, V_1 is a derivation of $\Psi; \Delta_1 \vdash B_1^+$, V_2 is a derivation of $\Psi; \Delta_2 \vdash B_2^+$,
- and Ξ matches $\Theta\{\!\{\Delta\restriction_{lvl}\}\!\}$.

There are three possibilities: either x is a variable declaration in Δ_1 or Δ_2 but not both (if lvl is eph or ord) or x is a variable declaration in both Δ_1 and Δ_2 (if lvl is $pers$).

The first two cases are symmetric; assume without loss of generality that x is a variable declaration in Δ_1 but not Δ_2; we can construct a Θ_1 and Δ_1' such that $\Theta_1\{x{:}A^- \; lvl\} = \Delta_1$, Δ_1' matches $\Theta_1\{\!\{\Delta_1\restriction_{lvl}\}\!\}$, and Ξ matches Δ_1', Δ_2 By the induction hypothesis on M and V_1, we have $[\![M/x]\!]^{A^-} V_1$, a derivation of $\Psi; \Delta_1' \vdash [B_1^+]$, and the result follows by rule $\bullet_R$ on $[\![M/x]\!]^{A^-} V_1$ and V_2.

The third case is similar; we construct a Θ_1, Δ_1', Θ_2, and Δ_2' such that $\Theta_1\{x{:}A^- \; lvl\} = \Delta_1$, $\Theta_2\{x{:}A^- \; lvl\} = \Delta_2$, Δ_1' matches $\Theta_1\{\!\{\Delta_1\restriction_{lvl}\}\!\}$, Δ_2' matches $\Theta_1\{\!\{\Delta_2\restriction_{lvl}\}\!\}$, and Ξ matches Δ_1', Δ_2', which is only possible because $lvl = pers$; we then invoke the induction hypothesis twice.

- $[\![M/x]\!]^{A^-}(\textsc{inl}(V)) = \textsc{inl}([\![M/x]\!]^{A^-} V)$
- $[\![M/x]\!]^{A^-}(\textsc{inr}(V)) = \textsc{inr}([\![M/x]\!]^{A^-} V)$
- $[\![M/x]\!]^{A^-}(t, V) = t, ([\![M/x]\!]^{A^-} V)$
- $[\![M/x]\!]^{A^-}\textsc{refl} = \textsc{refl}$

Substitution into terms

- $[\![M/x]\!]^{A^-} V = [\![M/x]\!]^{A^-} V$
- $[\![M/x]\!]^{A^-}(y \cdot Sp) = y \cdot ([\![M/x]\!]^{A^-} Sp)$ *(x#y)*
- $[\![M/x]\!]^{A^-}(x \cdot Sp) =$
  $(M \circ Sp)^{A^-}$ *(if x is not in Sp's context)*
  $(M \circ ([\![M/x]\!]^{A^-} Sp))^{A^-}$ *(if x is in Sp's context)*

  We must show $\Psi; \Xi \vdash U$, where
  - M is a derivation of $\Psi; \Delta \vdash A^-$,
  - $\Theta\{x{:}A^- \; lvl\}$ matches $\Theta'\{\!\{x{:}A^-\}\!\}$, Sp is a derivation of $\Psi; \Theta'\{[A^-]\} \vdash U$,
  - and Ξ matches $\Theta\{\!\{\Delta\restriction_{lvl}\}\!\}$.

  If lvl is eph or ord, then Ξ matches $\Theta'\{\!\{\Delta\}\!\}$, and the result follows by part 1 of cut admissibility on M and Sp.

  If lvl is $pers$, Ξ doesn't match $\Theta'\{\!\{\Delta\}\!\}$, as Θ' has an extra variable declaration $x{:}A^- \; pers$. Instead, we know $\Theta\{[A^-]\}$ matches $\Theta_{[A^-]}\{\!\{\Delta\restriction_{pers}\}\!\}$ and $\Theta_{[A^-]}\{x{:}A^- \; pers\} = \Theta'\{[A^-]\}$, so Sp is also a derivation of $\Psi; \Theta_{[A^-]}\{x{:}A^- \; pers\} \vdash U$. By the induction hypothesis on M and Sp, we have $[\![M/x]\!]^{A^-} Sp$, a derivation of $\Psi; \Theta\{[A^-]\} \vdash U$. Then, because Ξ matches $\Theta\{\!\{\Delta\}\!\}$, the result follows from part 1 of cut admissibility on M and $[\![M/x]\!]^{A^-} Sp$.

- $[\![M/x]\!]^{A^-}(\langle z\rangle.N) = \langle z\rangle.([\![M/x]\!]^{A^-} N)$
- $[\![M/x]\!]^{A^-}\langle N\rangle = \langle [\![M/x]\!]^{A^-} N\rangle$

- $[\![M/x]\!]^{A^-}(\downarrow y.N) = \downarrow y.([\![M/x]\!]^{A^-} N)$
- $[\![M/x]\!]^{A^-}(¡y.N) = ¡y.([\![M/x]\!]^{A^-} N)$
- $[\![M/x]\!]^{A^-}(!y.N) = !y.([\![M/x]\!]^{A^-} N)$

  We must show $\Psi; \Xi \vdash U$, where

  - M is a derivation of $\Psi; \Delta \vdash A^-$,
  - $\Theta\{x{:}A^-\ lvl\}$ matches $\Theta'\{\!\{!B^-\}\!\}$, N is a derivation of $\Psi; \Theta'\{y{:}B^-\ pers\} \vdash U$,
  - and Ξ matches $\Theta\{\!\{\Delta\restriction_{lvl}\}\!\}$.

  Let $\Delta' = \Delta, y{:}B^-\ pers$. By admissible weakening, M is derivation of $\Psi; \Delta' \vdash A^-$ too.

  Ξ matches $\Theta_\Delta\{\!\{!B^-\}\!\}$, $\Theta_\Delta\{y{:}B^-\ pers\}$ matches $\Theta_{B^-}\{\!\{\Delta'\restriction_{lvl}\}\!\}$, and $\Theta_{B^-}\{x{:}A^-\ lvl\} = \Theta'\{y{:}B^-\ pers\}$. By the induction hypothesis on M and N we have $\Psi; \Theta_\Delta\{y{:}B^-\ pers\} \vdash U$, and the result follows by rule $!_L$.

- $[\![M/x]\!]^{A^-}(\uparrow N) = \uparrow([\![M/x]\!]^{A^-} N)$
- $[\![M/x]\!]^{A^-}\{N\} = \{[\![M/x]\!]^{A^-} N\}$
- $[\![M/x]\!]^{A^-}\bullet N = \bullet([\![M/x]\!]^{A^-} N)$
- $[\![M/x]\!]^{A^-}\lambda^< N = \lambda^<([\![M/x]\!]^{A^-} N)$
- $[\![M/x]\!]^{A^-}\lambda^> N = \lambda^>([\![M/x]\!]^{A^-} N)$
- $[\![M/x]\!]^{A^-}\textsc{abort} = \textsc{abort}$
- $[\![M/x]\!]^{A^-}[N_1, N_2] = [[\![M/x]\!]^{A^-} N_1, [\![M/x]\!]^{A^-} N_2]$

  We must show $\Psi; \Xi \vdash U$, where

  - M is a derivation of $\Psi; \Delta \vdash A^-$,
  - $\Theta\{x{:}A^-\ lvl\}$ matches $\Theta'\{\!\{B_1^+ \oplus B_2^+\}\!\}$, N_1 is a derivation of $\Psi; \Theta'\{B_1^+\} \vdash U$, N_2 is a derivation of $\Psi; \Theta'\{B_2^+\} \vdash U$,
  - and Ξ matches $\Theta\{\!\{\Delta\restriction_{lvl}\}\!\}$.

  Ξ matches $\Theta_\Delta\{\!\{B_1^+ \oplus B_2^+\}\!\}$, and for $i \in \{1, 2\}$, $\Theta_\Delta\{B_i^+\}$ matches $\Theta_{B_i^+}\{\!\{\Delta\restriction_{pers}\}\!\}$ and $\Theta_{B_i^+}\{x{:}A^-\ lvl\} = \Theta'\{B_i^+\}$.

  By the induction hypothesis on M and N_1, we have $\Psi; \Theta_\Delta\{B_1^+\} \vdash U$, by the induction hypothesis on M and N_2, we have $\Psi; \Theta_\Delta\{B_2^+\} \vdash U$, and the result follows by rule $\oplus_L$.

- $[\![M/x]\!]^{A^-}\top = \top$
- $[\![M/x]\!]^{A^-}(N_1 \,\&\, N_2) = ([\![M/x]\!]^{A^-} N_1) \,\&\, ([\![M/x]\!]^{A^-} N_2)$
- $[\![M/x]\!]^{A^-} a.N = a.([\![M/x]\!]^{A^-} N)$
- $[\![M/x]\!]^{A^-}[a].N = [a].([\![M/x]\!]^{A^-} N)$
- $[\![M/x]\!]^{A^-}\textsc{unif}\,(\mathsf{fn}\,\sigma \Rightarrow \phi(\sigma)) = \textsc{unif}\,(\mathsf{fn}\,\sigma \Rightarrow [\![\sigma M/x]\!]^{A^-}\phi(\sigma))$

  We must show $\Psi; \Xi \vdash U$, where

  - M is a derivation of $\Psi; \Delta \vdash A^-$,
  - $\Theta\{x{:}A^-\ lvl\}$ matches $\Theta'\{\!\{t \doteq s\}\!\}$, ϕ is a function from substitutions $\Psi' \vdash \sigma : \Psi$ that

unify t and s to derivations of $\Psi'; \sigma\Theta'\{\cdot\} \vdash \sigma U$.

- and Ξ matches $\Theta\{\!\{\Delta\restriction_{lvl}\}\!\}$.

Ξ matches $\Theta_\Delta\{\!\{t \doteq s\}\!\}$, and for any substitution σ, $\sigma\Theta_\Delta\{\cdot\}$ matches $\sigma\Theta\{\!\{\Delta\restriction_{lvl}\}\!\}$. By rule $\doteq_L$, it suffices to show that, given an arbitrary substitution $\Psi' \vdash \sigma : \Psi$, there is a derivation of $\Psi'; \sigma\Theta_\Delta\{\cdot\} \vdash \sigma U$.

By applying σ to ϕ, we get $\phi(\sigma)$, a derivation of $\Psi'; \sigma\Theta_B\{\cdot\} \vdash \sigma A^+$; the usual interpretation of higher-order derivations is that $\phi(\sigma)$ is a subderivation of ϕ, so $\phi(\sigma)$ can be used to invoke the induction hypothesis. From variable substitution, we get σM, a derivation of $\Psi'; \sigma\Delta \vdash \sigma A^- \; lvl$, and the result follows by the induction hypothesis on σM and $\phi(\sigma)$.

Substitution into spines

- $[\![M/x]\!]^{A^-}\text{NIL} = \text{NIL}$
- $[\![M/x]\!]^{A^-}(\uparrow N) = \uparrow([\![M/x]\!]^{A^-} N)$
- $[\![M/x]\!]^{A^-}\{N\} = \{[\![M/x]\!]^{A^-} N\}$
- $[\![M/x]\!]^{A^-} V^< Sp =$
  - $([\![M/x]\!]^{A^-} V)^< Sp$ *(if x is in V's context but not Sp's)*
  - $V^<([\![M/x]\!]^{A^-} Sp)$ *(if x is in Sp's context but not V's)*
  - $([\![M/x]\!]^{A^-} V)^<([\![M/x]\!]^{A^-} Sp)$ *(if x is in both V and Sp's contexts)*
- $[\![M/x]\!]^{A^-} V^> Sp =$
  - $([\![M/x]\!]^{A^-} V)^> Sp$ *(if x is in V's context but not Sp's)*
  - $V^>([\![M/x]\!]^{A^-} Sp)$ *(if x is in Sp's context but not V's)*
  - $([\![M/x]\!]^{A^-} V)^>([\![M/x]\!]^{A^-} Sp)$ *(if x is in both V and Sp's contexts)*

We must show $\Psi; \Xi \vdash \forall x{:}\tau.B^-$, where

- M is a derivation of $\Psi; \Delta \vdash A^-$,
- $\Theta\{x{:}A^-\; lvl\}$ matches $\Theta'\{\!\{[B_1^+ \twoheadrightarrow B_2^-], \Delta_A\}\!\}$, V is a derivation of $\Psi; \Delta_A \vdash [B_1^+]$, Sp is a derivation of $\Psi; \Theta'\{[B_2^-]\} \vdash U$,
- and Ξ matches $\Theta\{\!\{\Delta\restriction_{lvl}\}\!\}$.

There are three possibilities: either x is a variable declaration in Θ' or Δ_A but not both (if lvl is eph or ord) or x is a variable declaration in both Θ' and Δ_A (if lvl is $pers$).

In the first case (x is a variable declaration in Δ_A only), Ξ matches $\Theta'\{\!\{[B_1^+ \twoheadrightarrow B_2^-], \Delta'_A\}\!\}$, Δ'_A matches $\Theta_A\{\!\{\Delta\restriction_{lvl}\}\!\}$, and $\Delta_A = \Theta_A\{x{:}A^-\; lvl\}$. By the induction hypothesis on M and V we have $[\![M/x]\!]^{A^-} V$, a derivation of $\Psi; \Delta'_A \vdash [B_1^+]$, and the result follows by rule $\twoheadrightarrow_L$ on $[\![M/x]\!]^{A^-} V$ and Sp.

In the second case (x is in Θ' only), Ξ matches $\Theta_\Delta\{\!\{[B_1^+ \twoheadrightarrow B_2^-], \Delta_A\}\!\}$, $\Theta_\Delta\{[B_2^-]\}$ matches $\Theta_{[B_2^-]}\{\!\{\Delta\restriction_{lvl}\}\!\}$, and $\Theta_{[B_2^-]}\{x{:}lvl\} = \Theta'\{[B_2^-]\}$. By the induction hypothesis on M and Sp, we have $[\![M/x]\!]^{A^-} Sp$, a derivation of $\Psi; \Theta_\Delta\{[B_2^-]\} \vdash U$, and the result follows by rule $\twoheadrightarrow_L$ on V and $[\![M/x]\!]^{A^-} Sp$.

In the third case (x is in Θ' and Δ_A), Ξ matches $\Theta_\Delta\{\!\{[B_1^+ \twoheadrightarrow B_2^-], \Delta'_A\}\!\}$, where Θ_Δ and Δ'_A have the same properties as before, and we proceed invoking the induction hypothesis

twice.

- $[\![M/x]\!]^{A^-}\pi_1; Sp = \pi_1; ([\![M/x]\!]^{A^-} Sp)$
- $[\![M/x]\!]^{A^-}\pi_2; Sp = \pi_2; ([\![M/x]\!]^{A^-} Sp)$
- $[\![M/x]\!]^{A^-}[t]; Sp = [t]; ([\![M/x]\!]^{A^-} Sp)$

3.5 Identity expansion

The form of the identity expansion theorems is already available to us: the admissible rules $\eta_{A^+_{lvl}}$ and $\eta_{A^-_{lvl}}$ are straightforward generalizations of the explicit rules η^+ and η^- in Figure 3.5 from ordered atomic propositions p^+ and p^- to arbitrary propositions and from permeable atomic propositions p^+_{eph}, p^+_{pers}, and p^-_{lax} to arbitrary permeable propositions A^+_{eph}, A^+_{pers} and A^-_{lax}. The content of Theorem 3.7 below is captured by the two admissible rules $\eta_{A^+_{lvl}}$ and $\eta_{A^-_{lvl}}$ and also by the two functions and $\eta_{A^+_{lvl}}(z.N)$ and $\eta_{A^-_{lvl}}(N)$ that operate on proof terms.

$$\frac{\Psi; \Theta\{z{:}\langle A^+_{lvl}\rangle\ lvl\} \vdash N : U}{\Psi; \Theta\{\!\{A^+_{lvl}\}\!\} \vdash \eta_{A^+_{lvl}}(z.N) : U}\ \eta_{A^+_{lvl}} \qquad \frac{\Psi; \Delta \vdash N : \langle A^-_{lvl}\rangle\ lvl \quad \Delta\ stable_L}{\Psi; \Delta \vdash \eta_{A^-_{lvl}}(N) : A^-_{lvl}}\ \eta_{A^-_{lvl}}$$

Identity expansion is not the perhaps not the best name for the property; the name comes from the fact that the usual identity properties are a corollary of identity expansion. Specifically, $\eta_{A^+}(z.z)$ is a derivation of $\Psi; A^+ \vdash A^+\ true$ and $\eta_{A^-}(x \cdot \text{NIL})$ is a derivation of $\Psi; x{:}A^-\ ord \vdash A^-$.

In the proof of identity expansion, we do pay some price in return for including permeable propositions, as we perform slightly different bookkeeping depending on whether or not it is necessary to apply admissible weakening to the subderivation N. However, this cost is mostly borne by the part of the context we leave implicit.

Theorem 3.7 (Identity expansion)**.**

* *If* $\Psi; \Theta\{z{:}\langle A^+_{lvl}\rangle\ lvl\} \vdash U$ *and* Δ *matches* $\Theta\{\!\{A^+_{lvl}\}\!\}$*, then* $\Psi; \Delta \vdash U$*.*
* *If* $\Psi; \Delta \vdash \langle A^-_{lvl}\rangle\ lvl$ *and* Δ *is stable, then* $\Psi; \Delta \vdash A^-_{lvl}$*.*

Proof. By mutual induction over the structure of types. We provide the full definition at the level of proof terms and include an extra explanatory derivation for a few of the positive cases.

Positive cases

- $\eta_{p^+_{lvl}}(z.N) = \langle z\rangle.N$

  N is a derivation of $\Psi; \Theta\{z{:}p^+_{lvl}\ lvl\} \vdash U$; the result follows immediately by the rule η^+:

$$\frac{\Psi; \Theta\{z{:}p^+_{lvl}\ lvl\} \vdash N : U}{\Psi; \Theta\{\!\{p^+_{lvl}\}\!\} \vdash \langle z\rangle.N : U}\ \eta^+$$

– $\eta_{\downarrow A^-}(z.N) = \downarrow x.([(\downarrow(\eta_{A^-}(x \cdot \text{NIL})))/z]N)$

N is a derivation of $\Psi; \Theta\{z{:}\langle\downarrow A^-\rangle\ ord\} \vdash U$. We construct a context Ξ that contains only the persistent propositions from Δ. This means $\Theta\{\!\{\Xi, x{:}A^-\ ord\}\!\}$ matches $\Theta\{x{:}A^-\ ord\}$. We can then derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Psi; \Xi, [A^-] \vdash \text{NIL} : \langle A^-\rangle\ true}\ id^-}{\Psi; \Xi, x{:}A^-\ ord \vdash x \cdot \text{NIL} : \langle A^-\rangle\ true}\ focus_L}{\Psi; \Xi, x{:}A^-\ ord \vdash \eta_{A^-}(x \cdot \text{NIL}) : A^-\ true}\ \eta_{A^-}}{\Psi; \Xi, x{:}A^-\ ord \vdash \downarrow(\eta_{A^-}(x \cdot \text{NIL})) : [\downarrow A^-]}\ \downarrow_R \qquad \Psi; \Theta\{z{:}\langle\downarrow A^-\rangle\ ord\} \vdash N : U}{\dfrac{\Psi; \Theta\{x{:}A^-\ ord\} \vdash [(\downarrow(\eta_{A^-}(x \cdot \text{NIL})))/z]N : U}{\Psi; \Theta\{\!\{\downarrow A^-\}\!\} \vdash \downarrow x.([(\downarrow(\eta_{A^-}(x \cdot \text{NIL})))/z]N) : U}\ \downarrow_L}\ subst^+$$

– $\eta_{¡A^-}(z.N) = ¡x.([(¡(\eta_{A^-}(x \cdot \text{NIL})))/z]N)$

– $\eta_{!A^-}(z.N) = !x.([(!(\eta_{A^-}(x \cdot \text{NIL})))/z]N)$

N is a derivation of $\Psi; \Theta\{z{:}\langle !A^-\rangle\ lvl\} \vdash U$, where lvl can be anything (ord, eph, or $pers$). We construct a context Ξ that contains only the persistent propositions from Δ and a frame Θ^+ that is Θ plus an extra variable declaration $x{:}A^-\ pers$. This means that $\Theta\{x{:}A^-\ pers\}$ matches $\Theta^+\{\!\{(\Xi, x{:}A^-\ pers)\restriction_{lvl}\}\!\}$. We can then derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Psi; \Xi, x{:}A^-\ pers, [A^-] \vdash \text{NIL} : \langle A^-\rangle\ true}\ id^-}{\Psi; \Xi, x{:}A^-\ pers \vdash x \cdot \text{NIL} : \langle A^-\rangle\ true}\ focus_L}{\Psi; \Xi, x{:}A^-\ pers \vdash \eta_{A^-}(x \cdot \text{NIL}) : A^-\ true}\ \eta_{A^-}}{\Psi; \Xi, x{:}A^-\ pers \vdash !(\eta_{A^-}(x \cdot \text{NIL})) : [!A^-]}\ !_R \qquad \dfrac{\Psi; \Theta\{z{:}\langle\downarrow A^-\rangle\ lvl\} \vdash N : U}{\Psi; \Theta^+\{z{:}\langle\downarrow A^-\rangle\ lvl\} \vdash N : U}\ weaken}{\dfrac{\Psi; \Theta\{x{:}A^-\ pers\} \vdash [(!(\eta_{A^-}(x \cdot \text{NIL})))/z]N : U}{\Psi; \Theta\{\!\{!A^-\}\!\} \vdash !x.([(!(\eta_{A^-}(x \cdot \text{NIL})))/z]N) : U}\ !_L}\ subst^+$$

– $\eta_{\mathbf{1}}(z.N) = ().([()/z]N)$

– $\eta_{A^+_{lvl} \bullet B^+_{lvl}}(z.N) = \bullet\big(\eta_{A^+_{lvl}}(z_1.\ \eta_{B^+_{lvl}}(z_2.\ [(z_1 \bullet z_2)/z]N))\big)$

N is a derivation of $\Psi; \Theta\{z{:}\langle A^+_{lvl} \bullet B^+_{lvl}\rangle\ lvl\} \vdash U$, where lvl can be anything (ord, eph, or $pers$). We construct a context Ξ that contains only the persistent propositions from Δ and a frame Θ^+ that is either Θ (if lvl is ord or eph) or it is Θ plus additional variable declarations $z_1{:}\langle A^+_{lvl}\rangle\ lvl$ and $z_2{:}\langle B^+_{lvl}\rangle\ lvl$ (if lvl is $pers$). This means that $\Theta\{x_1{:}\langle A^+_{lvl}\rangle\ lvl, x_2{:}\langle B^+_{lvl}\rangle\ lvl\}$ matches $\Theta^+\{\!\{(\Xi, x_1{:}\langle A^+_{lvl}\rangle\ lvl, x_2{:}\langle B^+_{lvl}\rangle\ lvl)\restriction_{lvl}\}\!\}$. We can then derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Psi; \Xi_1 \vdash z_1 : [A^+_{lvl}]}\ id^+ \quad \dfrac{}{\Psi; \Xi_2 \vdash z_2 : [B^+_{lvl}]}\ id^+}{\Psi; \Xi, z_1{:}\langle A^+_{lvl}\rangle\ lvl, z_2{:}\langle B^+_{lvl}\rangle\ lvl \vdash z_1 \bullet z_2 : [A^+_{lvl} \bullet B^+_{lvl}]}\ \bullet_R \qquad \dfrac{\Psi; \Theta\{z{:}\langle A^+_{lvl} \bullet B^+_{lvl}\rangle\ lvl\} \vdash N : U}{\Psi; \Theta^+\{z{:}\langle A^+_{lvl} \bullet B^+_{lvl}\rangle\ lvl\} \vdash N : U}\ weaken}{\Psi; \Theta\{z_1{:}\langle A^+_{lvl}\rangle\ lvl, z_2{:}\langle B^+_{lvl}\rangle\ lvl\} \vdash [(z_1 \bullet z_2)/z]N : U}\ subst^+}{\Psi; \Theta\{z_1{:}\langle A^+_{lvl}\rangle\ lvl, B^+_{lvl}\} \vdash \eta_{B^+_{lvl}}(z_2.\ [(z_1 \bullet z_2)/z]N) : U}\ \eta_{B^+_{lvl}}}{\Psi; \Theta\{A^+_{lvl}, B^+_{lvl}\} \vdash \eta_{A^+_{lvl}}(z_1.\ \eta_{B^+_{lvl}}(z_2.\ [(z_1 \bullet z_2)/z]N)) : U}\ \eta_{A^+_{lvl}}}{\Psi; \Theta\{\!\{A^+_{lvl} \bullet B^+_{lvl}\}\!\} \vdash \bullet(\eta_{A^+_{lvl}}(z_1.\ \eta_{B^+_{lvl}}(z_2.\ [(z_1 \bullet z_2)/z]N))) : U}\ \bullet_L$$

Either Ξ_1 and Ξ_2 are both $\Xi, z_1{:}\langle A^+_{lvl}\rangle\ lvl, z_2{:}\langle B^+_{lvl}\rangle\ lvl$ (if lvl is $pers$), or Ξ_1 is $\Xi, z_1{:}\langle A^+_{lvl}\rangle\ lvl$ and Ξ_2 is $\Xi, z_2{:}\langle B^+_{lvl}\rangle\ lvl$ (if lvl is ord or eph).

- $\eta_{\mathbf{0}}(z.N) = \textsc{abort}$
- $\eta_{A^+_{lvl} \oplus B^+_{lvl}}(z.N) = [\eta_{A^+_{lvl}}(z_1.\ [\textsc{inl}(z_1)/z]N), \eta_{B^+_{lvl}}(z_2.\ [\textsc{inr}(z_2)/z]N)]$
- $\eta_{\exists a:\tau.A^+_{lvl}}(z.N) = a.\eta_{A^+}(z'.\ [(a, z')/z]N)$
- $\eta_{t \doteq_\tau s}(z.N) = \textsc{unif}\,(\mathsf{fn}\ \sigma \Rightarrow [\textsc{refl}/z](\sigma N))$

Negative cases

- $\eta_{p^-_{lvl}}(N) = \langle N \rangle$
- $\eta_{\uparrow A^+}(N) = \uparrow([N](\uparrow(\eta_{A^+}(z.z))))$
- $\eta_{\bigcirc A^+}(N) = \{[N](\{\eta_{A^+}(z.z)\})\}$
- $\eta_{A^+ \rightarrowtail B^-_{lvl}}(N) = \lambda^<(\eta_{A^+}(z.\ \eta_{B^-_{lvl}}([N](z^< \textsc{nil}))))$
- $\eta_{A^+ \twoheadrightarrow B^-_{lvl}}(N) = \lambda^>(\eta_{A^+}(z.\ \eta_{B^-_{lvl}}([N](z^> \textsc{nil}))))$
- $\eta_\top(N) = \top$
- $\eta_{A^-_{lvl} \& B^-_{lvl}}(N) = (\eta_{A^-_{lvl}}([N](\pi_1; \textsc{nil}))) \,\&\, (\eta_{B^-_{lvl}}([N](\pi_2; \textsc{nil})))$
- $\eta_{\forall a:\tau.A^-_{lvl}}(N) = [a].(\eta_{A^-_{lvl}}([N]([a]; \textsc{nil})))$

□

3.6 Correctness of focusing

Our proof of the correctness of focusing is based on erasure as described in Section 2.3.7. The argument follows the one from the structural focalization development, and the key component is the set of *unfocused admissibility lemmas*, lemmas that establish that each of the reasoning steps that can be made in unfocused OL_3 are admissible inferences made on stable sequents in focused OL_3.

3.6.1 Erasure

As in Section 2.3.7, we define erasure only on stable, suspension-normal sequents. Erasure for propositions is defined as in Figure 3.8. As discussed in Section 2.5.4, even though we have not incorporated a notion of permeable and mobile atomic propositions into the unfocused presentation of OL_3, it is possible to erase a permeable atomic proposition p^+_{pers} as $!p^+_{pers}$.[3] In this way, we can see the separation criteria from our previous work [SP08, PS09] arising as an emergent property of erasure.

We have to define erasure on non-stable sequents in order for the soundness of focusing to go through, though we will only define erasure on suspension-normal sequents. The erasure of sequents, U°, maps polarized succedents $A^+\ lvl$, $\langle p^-_{lvl} \rangle\ lvl$, $[A^+]$, and A^- in the obvious way to unpolarized succedents $(A^+)^\circ\ lvl$, $p^-_{lvl}\ lvl$, $(A^+)^\circ\ ord$, and $(A^-)^\circ\ ord$, respectively. To describe the erasure of contexts more simply, we will assume that we can give a presentation of unfocused

[3] The polarity and level annotations are meaningless in the unfocused logic. We keep them only to emphasize that p^+_{pers} and p^-_{lax} do *not* erase to the same unpolarized atomic proposition p but two distinct unpolarized atomic propositions.

$$
\begin{array}{rcl}
\boxed{(A^+)^\circ} & & \\
(p^+)^\circ & = & p^+ \\
(p^+_{eph})^\circ & = & ¡p^+_{eph} \\
(p^+_{pers})^\circ & = & !p^+_{pers} \\
(\downarrow A^-)^\circ & = & (A^-)^\circ \\
(¡A^-)^\circ & = & ¡(A^-)^\circ \\
(!A^-)^\circ & = & !(A^-)^\circ \\
(\mathbf{1})^\circ & = & \mathbf{1} \\
(A^+ \bullet B^+)^\circ & = & (A^+)^\circ \bullet (B^+)^\circ \\
(\mathbf{0})^\circ & = & \mathbf{0} \\
(A^+ \oplus B^+)^\circ & = & (A^+)^\circ \oplus (B^+)^\circ \\
(\exists a{:}\tau.A^+)^\circ & = & \exists a{:}\tau.(A^+)^\circ \\
(t \doteq s)^\circ & = & t \doteq s
\end{array}
\qquad
\begin{array}{rcl}
\boxed{(A^-)^\circ} & & \\
(p^-)^\circ & = & p^- \\
(p^-_{lax})^\circ & = & \bigcirc p^-_{lax} \\
 & & \\
(\uparrow A^+)^\circ & = & (A^+)^\circ \\
(\bigcirc A^+)^\circ & = & \bigcirc (A^+)^\circ \\
 & & \\
(A^+ \rightarrowtail B^-)^\circ & = & (A^+)^\circ \rightarrowtail (B^+)^\circ \\
(A^+ \twoheadrightarrow B^-)^\circ & = & (A^+)^\circ \twoheadrightarrow (B^-)^\circ \\
(\top)^\circ & = & \top \\
(A^- \,\&\, B^-)^\circ & = & (A^-)^\circ \,\&\, (B^-)^\circ \\
(\forall a{:}\tau.A^-)^\circ & = & \forall a{:}\tau.(A^-)^\circ
\end{array}
$$

Figure 3.8: Erasure in OL_3

OL_3 that uses unified substructural contexts, as we outlined in Section 3.2; the judgments of this presentation have the form $\Psi; \Delta \Longrightarrow U$. In this presentation, we can define Δ° that takes every variable declaration $x{:}A^-\ lvl$, $x{:}\langle p^+_{lvl}\rangle\ lvl$, $[A^-]$, or A^+ to a variable declaration $x{:}(A^-)^\circ\ lvl$, $x{:}p^+_{lvl}\ lvl$, $x{:}(A^-)^\circ\ ord$, or $x{:}(A^+)^\circ\ ord$ (and in the process, either comes up with or reveals the suppressed variable names associated with focused negative propositions and inverting positive propositions). Erasure of succedents U° is similar: $(A^+\ lvl)^\circ = (A^+)^\circ\ lvl$, $(\langle p^-_{lvl}\rangle\ lvl)^\circ = p^-_{lvl}\ lvl$, $([A^-])^\circ = (A^-)^\circ\ true$, and $(A^+)^\circ = (A^+)^\circ\ true$.

3.6.2 De-focalization

The act of taking a focused proof of a sequent and getting an unfocused proof of the corresponding erased sequent is *de-focalization*. If we run the constructive content of the proof of the soundness of focusing (the OL_3 analogue of Theorem 2.5 from Section 2.3.7), the proof performs de-focalization.

Theorem 3.8 (Soundness of focusing/de-focalization)**.** *If* $\Psi; \Delta \vdash U$*, then* $\Psi; \Delta^\circ \Longrightarrow U^\circ$.

Proof. By induction over the structure of focused proofs. Most rules ($\bullet_L$, $\twoheadrightarrow_R$, etc.) in the focused derivations have an obviously analogous rule in the unfocused logic, and for the four rules dealing with shifts, the necessary result follows directly from the induction hypothesis. The $focus_L$ rule potentially requires an instance of the admissible $copy$ or $place$ rules in unfocused OL_3, and the $focus_R$ rule potentially requires an instance of the admissible lax rule in unfocused OL_3. □

3.6.3 Unfocused admissibility

Unfocused admissibility has a structure that is unchanged from the previous discussion in the proof of the completeness of focusing for linear logic (Theorem 2.6 in Section 2.3.7). In this

Atomic propositions

$$\frac{}{\Psi; z{:}\langle p^+_{lvl}\rangle \vdash z : p^+_{lvl}\ lvl'} \qquad \frac{}{\Psi; x{:}\uparrow p^+_{lvl} \vdash x \cdot \uparrow\langle z\rangle.\ z : p^+_{lvl}\ lvl'}$$

$$\frac{}{\Psi; x{:}p^-_{lvl} \vdash x \cdot \textsc{nil} : \langle p^-_{lvl}\rangle\ lvl} \qquad \frac{}{\Psi; x{:}p^-_{lvl} \vdash \downarrow\langle x \cdot \textsc{nil}\rangle : \downarrow p^-_{lvl}\ lvl}$$

Exponentials

$$\frac{\Psi; \Delta \vdash N : A^+\ ord}{\Psi; \Delta \vdash \downarrow\uparrow N : \downarrow\uparrow A^+\ lvl} \qquad \frac{\Psi; \Theta\{x'{:}A^-\ ord\} \vdash N : U}{\Psi; \Theta\{\!\{x{:}\uparrow\downarrow A^-\}\!\} \vdash x \cdot \uparrow\downarrow x'.N : U}$$

$$\frac{\Psi; \Delta \vdash N : \downarrow A^-\ ord}{\Psi; \Delta\restriction_{eph} \vdash [\![\uparrow N/x]\!]^{\uparrow\downarrow A^-}\ ¡\eta_{A^-}(x \cdot \uparrow(\downarrow x'.\ (x' \cdot \textsc{nil}))) : ¡A^-\ lvl} \qquad \frac{\Psi; \Theta\{x'{:}A^-\ eph\} \vdash N : U}{\Psi; \Theta\{\!\{x{:}\uparrow ¡A^-\}\!\} \vdash x \cdot \uparrow ¡x'.N : U}$$

$$\frac{\Psi; \Delta \vdash N : \downarrow A^-\ ord}{\Psi; \Delta\restriction_{pers} \vdash [\![\uparrow N/x]\!]^{\uparrow\downarrow A^-}\ !\eta_{A^-}(x \cdot \uparrow(\downarrow x'.\ (x' \cdot \textsc{nil}))) : !A^-\ lvl} \qquad \frac{\Psi; \Theta\{x'{:}A^-\ pers\} \vdash N : U}{\Psi; \Theta\{\!\{x{:}\uparrow !A^-\}\!\} \vdash x \cdot \uparrow !x'.N : U}$$

$$\frac{\Psi; \Delta \vdash N : A^+\ lax}{\Psi; \Delta \vdash \downarrow\{N\} : \downarrow\bigcirc A^+\ lvl} \qquad \frac{\Psi; \Theta\{x'{:}\uparrow A^+\ ord\} \vdash N : U}{\Psi; \Theta\{\!\{x{:}\bigcirc A^+\}\!\} \vdash [\![x \cdot \eta_{A^+}(z.\ \downarrow\uparrow z)]\!]^{\downarrow\uparrow A^+}\downarrow x'.N : U\downharpoonleft^{lax}}$$

Multiplicative connectives ($\rightarrowtail$ and $\twoheadrightarrow$ are symmetric)

$$\frac{}{\Psi; \cdot \vdash () : \mathbf{1}\ lvl} \qquad \frac{\Psi; \Theta\{\cdot\} \vdash N : U}{\Psi; \Theta\{\!\{x{:}\uparrow\mathbf{1}\}\!\} \vdash x \cdot \uparrow().N : U}$$

$$\frac{\Psi; \Delta_1 \vdash N_1 : A^+\ ord \quad \Psi; \Delta_2 \vdash N_2 : B^+\ ord}{\Psi; \Delta_1, \Delta_2 \vdash [\![\uparrow N_1/x_1]\!]^{\uparrow A^+}([\![N_2]\!]^{B^+}\eta_{B^+}(z_2.\ x_1 \cdot \uparrow\eta_{A^+}(z_1.\ z_1 \bullet z_2))) : A^+ \bullet B^+\ lvl}$$

$$\frac{\Psi; \Theta\{x_1{:}\uparrow A^+, x_2{:}\uparrow B^+\} \vdash N : U}{\Psi; \Theta\{\!\{x{:}\uparrow(A^+ \bullet B^+)\}\!\} \vdash x \cdot \uparrow[\![\eta_{A^+}(z_1.\ \eta_{B^+}(z_2.\ \downarrow\uparrow z_1 \bullet \downarrow\uparrow z_2))]\!]^{\downarrow\uparrow A^+ \bullet \downarrow\uparrow B^+} \bullet (\downarrow x_1.\downarrow x_2.N) : U}$$

$$\frac{\Psi; x{:}\uparrow A^+\ ord, \Delta \vdash N : \downarrow B^-\ ord}{\Psi; \Delta \vdash [\![(\lambda^<(\downarrow x.\uparrow N))/x']\!]^{\downarrow\uparrow A^+ \rightarrowtail \uparrow\downarrow B^-}(\downarrow\lambda^<\eta_{A^+}(z.\ \eta_{B^-}(x' \cdot (\downarrow\uparrow z)^<(\uparrow\downarrow x''.x'' \cdot \textsc{nil})))) : \downarrow(A^+ \rightarrowtail B^-)\ ord}$$

$$\frac{\Psi; \Delta_A \vdash N_1 : A^+\ ord \quad \Psi; \Theta\{x'{:}B^-\ ord\} \vdash N_2 : U}{\Psi; \{\!\{\Delta_A, x{:}A^+ \rightarrowtail B^-\}\!\} \vdash [\![[\![N]\!]\eta_{A^+}(z.\ \downarrow\eta_{B^-}(x \cdot z^< \textsc{nil}))]\!]\downarrow x'.N_2 : U}$$

Figure 3.9: Unfocused admissibility for the multiplicative, exponential fragment of OL_3

$$\frac{}{\Psi;\Theta\{\!\{x{:}\uparrow\mathbf{0}\}\!\} \vdash x \cdot \uparrow\textsc{abort} : U} \qquad \frac{\Psi;\Delta \vdash N_1 : A^+\ ord}{\Psi;\Delta \vdash [\![N_1]\!]\eta_{A^+}(z.\textsc{inl}(z))A^+ : A^+ \oplus B^+\ lvl}$$

$$\frac{\Psi;\Theta\{x_1{:}\uparrow A^+\ ord\} \vdash N_1 : U \quad \Psi;\Theta\{x_1{:}\uparrow B^+\ ord\} \vdash N_2 : U}{\Psi;\Theta\{\!\{x{:}\uparrow(A^+ \oplus B^+)\}\!\} \vdash x \cdot \uparrow[\![[\eta_{A^+}(y.\downarrow\uparrow y), \eta_{B^+}(y.\downarrow\uparrow y)]]\!][\downarrow x_1.N_1, \downarrow x_2.N_2] : U}$$

$$\frac{}{\Psi;\Delta \vdash \downarrow\top : \downarrow\top\ lvl} \qquad \frac{\Psi;\Theta\{x_1{:}A^-\ ord\} \vdash N_1 : U}{\Psi;\Theta\{\!\{x{:}A^- \mathbin{\&} B^-\}\!\} \vdash [\![\eta_{A^-}(x \cdot \pi_1; \textsc{nil})/x_1]\!]N_1 : U}$$

$$\frac{\Psi;\Delta \vdash N_1 : \downarrow A^-\ ord \quad \Psi;\Delta \vdash N_2 : \downarrow B^-\ ord}{\Psi;\Delta \vdash [\![(\uparrow N_1 \mathbin{\&} \uparrow N_2)/x]\!]\downarrow(\eta_{A^-}(x \cdot \pi_1; \uparrow(\downarrow y.\, y \cdot \textsc{nil})) \mathbin{\&} \eta_{B^-}(\ldots)) : \downarrow(A^- \mathbin{\&} B^-)\ lvl}$$

Figure 3.10: Unfocused admissibility for the additive connectives of OL_3 (omits $\oplus_{R2}$, $\&_{L2}$)

$$\frac{\Psi \vdash t : \tau \quad \Psi;\Delta \vdash N : [t/a]A^+\ ord}{\Psi;\Delta \vdash [\![N]\!](\eta_{[t/a]A^+}(z.\,(t,z))) : \exists a{:}\tau.A^+\ lvl}$$

$$\frac{\Psi, a{:}\tau;\Theta\{x'{:}\uparrow A^+\ ord\} \vdash N : U}{\Psi;\Theta\{\!\{x{:}\uparrow(\exists a{:}\tau.A^+)\}\!\} \vdash x \cdot \uparrow[\![a.\eta_{A^+}(z.\,(a, \downarrow\uparrow z))]\!]^{\exists a:\tau.\downarrow\uparrow A^+}(a.\downarrow x'.N) : U}$$

$$\frac{\Psi, a{:}\tau;\Delta \vdash N : \downarrow A^-\ ord}{\Psi;\Delta \vdash [\![[a].\uparrow N/x]\!]^{\forall a:\tau.\uparrow\downarrow A^-}\uparrow([a].\eta_{A^-}(x \cdot [a]; \uparrow(\downarrow y.y \cdot \textsc{nil}))) : \downarrow(\forall a{:}\tau.A^-)\ lvl}$$

$$\frac{\Psi \vdash t : \tau \quad \Psi;\Theta\{x'{:}[t/a]A^-\ ord\} \vdash N : U}{\Psi;\Theta\{\!\{x{:}\forall a{:}\tau.A^-\}\!\} \vdash [\![\eta_{[t/a]A^-}(x \cdot [a]; \textsc{nil})/x']\!]N' : U} \qquad \frac{}{\Psi;\cdot \vdash \textsc{refl} : t \doteq t\ lvl}$$

$$\frac{\forall(\Psi' \vdash \sigma : \Psi).\quad \sigma t = \sigma s \quad \rightarrow \quad \Psi';\sigma\Theta\{\cdot\} \vdash \phi(\sigma) : \sigma U}{\Psi;\Theta\{\!\{x{:}\uparrow(t \doteq s)\}\!\} \vdash x \cdot \uparrow(\textsc{unif}\,(\mathsf{fn}\ \sigma \Rightarrow \phi(\sigma))) : U}$$

Figure 3.11: Unfocused admissibility for the first-order connectives of OL_3

presentation, we present unfocused admissibility primarily on the level of proof terms. The resulting presentation is quite dense; proofs of this variety really ought to be mechanized, though we leave that for future work.

For the most part, there is exactly one unfocused admissibility rule for each rule of unfocused OL_3. The justifications for the unfocused admissibility lemmas for the multiplicative, exponential fragment of OL_3 are given in Figure 3.9; the additive fragment is given in Figure 3.10, and the first-order connectives are treated in Figure 3.11. There are two additional rules that account for the fact that different polarized propositions, like $\downarrow\uparrow\downarrow\uparrow A^+$ and A^+ erase to the same unpolarized proposition $(A^+)^\circ$. For the same reason, Figure 3.9 contains four id-like rules, since atomic propositions can come in positive and negative varieties and can appear in the context either suspended or not.

We can view unfocused admissibility as creating an abstraction layer of admissible rules that can be used to build focused proofs of stable sequents. The proof of the completeness of focusing below constructs focused proofs entirely by working through the interface layer of unfocused admissibility.

3.6.4 Focalization

The act of taking an unfocused proof of an erased sequent and getting a focused proof of the unerased sequent is *focalization*. If we run the constructive content of the proof of the completeness of focusing (the OL_3 analogue of Theorem 2.6 from Section 2.3.7), which takes any stable, suspension-normal sequent as input, the proof performs focalization.

Theorem 3.9 (Complteness of focusing/focalization)**.**
If $\Psi; \Delta^\circ \Longrightarrow U^\circ$, where Δ and U are stable and suspension-normal, then $\Psi; \Delta \vdash U$.

Proof. By an outer induction on the structure of unfocused proofs and an inner induction over the structure of polarized formulas A^+ and A^- in order to remove series of shifts $\uparrow\downarrow \ldots \uparrow\downarrow A^-$ from formulas until an unfocused admissibility lemma can be applied. □

3.7 Properties of syntactic fragments

In the structural focalization methodology, once cut admissibility and identity expansion are established the only interesting part of the proof of the completeness of focusing is the definition of an erasure function and the presentation of a series of unfocused admissibility lemmas. The unfocused admissibility lemmas for non-invertible rules, like $\bullet_R$ and $\rightarrowtail_L$, look straightforward:

$$\frac{\Psi; \Delta_1 \vdash A^+ \; \mathit{true} \quad \Psi; \Delta_2 \vdash B^+ \; \mathit{true}}{\Psi; \Delta_1, \Delta_2 \vdash A^+ \bullet B^+ \; \mathit{lvl}} \qquad \frac{\Psi; \Delta_A \vdash A^+ \; \mathit{ord} \quad \Psi; \Theta\{x'{:}B^- \; \mathit{ord}\} \vdash U}{\Psi; \Theta\{\!\{\Delta_A, x{:}A^+ \rightarrowtail B^-\}\!\} \vdash U}$$

Because unfocused admissibility is defined only on stable sequents in our methodology, the invertible rules, like $\bullet_L$ and $\rightarrowtail_R$, require the presence of shifts:

$$\frac{\Psi; \Theta\{x_1{:}{\uparrow}A^+ \; \mathit{ord}, x_2{:}{\uparrow}B^+ \; \mathit{ord}\} \vdash U}{\Psi; \Theta\{\!\{x{:}{\uparrow}(A^+ \bullet B^+)\}\!\} \vdash U} \qquad \frac{\Psi; x{:}{\uparrow}A^+ \; \mathit{ord}, \Delta \vdash {\downarrow}B^- \; \mathit{true}}{\Psi; \Delta \vdash {\downarrow}(A^+ \rightarrowtail B^-) \; \mathit{lvl}}$$

The presence of shifts is curious, due to our observation in Section 3.3.2 that the shifts have much of the character of exponentials; they are exponentials that do not place any restrictions on the form of the context.

As a thought experiment, imagine the removal of shifts $\uparrow$ and $\downarrow$ from the language of propositions in OL_3. Were it not for the presence of atomic propositions p^+ and p^-, this change would make every proposition A^+ a mobile proposition A^+_{eph} and would make every proposition A^- a right-permeable proposition A^-_{lax}. But arbitrary atomic propositions are intended to be stand-ins for arbitrary propositions! If arbitrary propositions lack shifts, then non-mobile atomic propositions would appear to no longer stand for anything. Therefore, let's remove them too, leaving only the permeable, mobile, and right-permeable atomic propositions p^+_{pers}, p^+_{eph}, and p^-_{lax}. Having done so, every positive proposition is mobile, and every negative proposition is right-permeable.

Now we have a logical fragment where every positive proposition is mobile and every negative proposition is observed to be right-permeable. Consider a derivation $\Psi; \Delta \vdash A^+ \; \mathit{lax}$ where Δ is stable and includes only linear and persistent judgments (that is, $\Delta\restriction_{eph}$). It is simple to

observe that, for every subderivation $\Psi'; \Delta' \vdash U'$, if Δ' is stable then $\Delta' = \Delta'\restriction_{eph}$, and if U is stable then $U = U\lfloor^{lax}$. Given that this is the case, the restrictions that the focused $¡_R$ and $\bigcirc_L$ rules make are *always satisfiable*, the same property that we previously observed of focused shift rules $\downarrow_R$ and $\uparrow_L$. In our syntactic fragment, in other words, the exponentials ¡ and $\bigcirc$ have become effective replacements for $\downarrow$ and $\uparrow$.

The cut and identity theorems survive our restriction of the logic entirely intact: these theorems handle each of the connectives separately and are stable to the addition or removal of individual connectives. That is not true for the unfocused admissibility lemmas, which critically and heavily use shifts. However, while we no longer have our original shifts, we have replacement shifts in the form of ¡ and $\bigcirc$, and can replay the logic of the unfocused admissibility lemmas in order to gain new ones that look like this:

$$\frac{\Psi; \Delta_1 \vdash A^+ \; lax \quad \Psi; \Delta_2 \vdash B^+ \; lax}{\Psi; \Delta_1, \Delta_2 \vdash A^+ \bullet B^+ \; lax} \qquad \frac{\Psi; \Delta_A \vdash A^+ \; lax \quad \Psi; \Theta\{x'{:}B^- \; eph\} \vdash U}{\Psi; \Theta\{\!\{\Delta_A, x{:}A^+ \rightarrowtail B^-\}\!\} \vdash U}$$

$$\frac{\Psi; \Theta\{x_1{:}¡A^+ \; eph, x_2{:}¡B^+ \; eph\} \vdash U}{\Psi; \Theta\{\!\{x{:}¡(A^+ \bullet B^+)\}\!\} \vdash U} \qquad \frac{\Psi; x{:}¡A^+ \; eph, \Delta \vdash \bigcirc B^- \; lax}{\Psi; \Delta \vdash \bigcirc(A^+ \rightarrowtail B^-) \; lax}$$

(To be clear, just as all the unfocused admissibility lemmas only applied to stable sequents, the unfocused admissibility lemmas above only apply when contexts and succedents are both stable and free of judgments $T \; ord$ and $T \; true$.)

The point of this exercise is that, given the definition and metatheory of OL_3, there is a reasonably large family of related systems, including ordered linear logic, lax logic, linear lax logic, and linear logic, that can be given erasure-based focalization proofs relative to OL_3; at most, the erasure function and the unfocused admissibility lemmas need to be adapted. The fragment we have defined here corresponds to regular linear logic. In the erasure of polarized OL_3 propositions to linear logic propositions, the "pseudo-shifts" $\bigcirc$ and ¡ are wiped away: $(\bigcirc A^+)^\circ = (A^+)^\circ$ and $(¡A^-)^\circ = (A^-)^\circ$. Additionally, the two implications are conflated: $(A^+ \rightarrowtail B^-)^\circ = (A^+ \twoheadrightarrow B^-)^\circ = (A^+)^\circ \multimap (B^-)^\circ$. Beyond that, and the renaming of fuse to tensor – $(A^+ \bullet B^+)^\circ = (A^+)^\circ \otimes (B^+)^\circ$ – the structure of erasure remains intact, and we can meaningfully focalize unfocused linear logic derivations into focused OL_3 derivations.

3.8 The design space of proof terms

In the design space of logical frameworks, our decision to view proof terms E as being fully intrinsically typed representatives of focused derivations is somewhat unusual. This is because, in a dependently typed logical framework, the variable substitution theorem (which we had to establish very early on) and the cut admissibility theorem (which we established much later) are effectively the same theorem; handling everything at once is difficult at best, and dependent types seem to force everything to be handled at once in an intrinsically typed presentation.

Since the advent of Watkins' observations about the existence of hereditary substitution and its application to logical frameworks [WCPW02], the dominant approach to the metatheory of logical frameworks has to define proof terms E that have little, if any, implicit type structure: just enough so that it is possible to define the hereditary substitution function $[\![M/x]\!]E$. The work

by Martens and Crary goes further, treating hereditary substitution as a relation, not a function, so that absolutely no intrinsic type system is necessary, and the proof terms are merely untyped abstract binding trees [MC12].

If we were to take such an approach, we would need to treat the judgment $\Psi; \Delta \vdash E : U$ as a genuine four-place relation, rather than the three-place relation $\Psi; \Delta \vdash U$ annotated with a derivation E of that sequent. Then, the analogue of cut admissibility (part 4) would show that if $\Psi; \Delta \vdash M : A^-$ and $\Psi; \Theta\{x{:}A^- \; lvl\} \vdash E : U$, and Ξ matches $\Theta\{\!\{\Delta\restriction_{lvl}\}\!\}$, then $\Psi; \Xi \vdash [\![M/x]\!]E : U$, where $[\![M/x]\!]E$ *is some function on proof terms that has already been defined*, rather than just an expression of the computational content of the theorem. Being able to comfortably conflate the computational content of a theorem with its operation on proof terms is the primary advantage of the approach taken in this chapter; it avoids a great deal of duplicated effort. The cost to this approach is that we cannot apply the modern Canonical LF methodology in which we define a proof term language that is intrinsically only simply well-typed and then overlay a dependent type system on top of it (this is discussed in Section 4.1.2 in the context of LF). As we discuss further in Section 4.7.3, this turns out not to be a severe limitation given the way we want to use OL_3.

It is not immediately obvious how the substitution $[\![M/x]\!]E$ could be defined without accounting for the full structure of derivations. The rightist substitution function, in particular, is computationally dependent on the implicit bookkeeping associated with the matching constructs, and that bookkeeping is more of an obstacle in our setting than the implicit type annotations. The problem, if we wish to see it as a problem, is that we cannot substitute a derivation M of $\Psi; \Delta \vdash A^-$ into a derivation E of $\Psi; \Theta\{x{:}A^- \; ord\} \vdash U$ unless x is *actually free* in E. Therefore, when we try to substitute the same M into $V_1 \bullet V_2$, we are forced to determine what judgment x is associated with; if x is associated with a linear or ephemeral judgment, we must track which subderivation x is assigned to in order to determine what is to be done next.

Fine-grained tracking of variables during substitution is both very inefficient when type theories are implemented as logical frameworks and unnatural to represent for proof assistants like Twelf that implement a persistent notion of bound variables. Therefore other developments have addressed this problem (see, for example, Cervesato et al. [CdPR99], Schack-Nielsen and Schürmann [SNS10], and Crary [Cra10]). It might be possible to bring our development more in line with these other developments by introducing a new matching construct of substitution into contexts, the substitution construct $[\Delta/(x{:}A^- \; lvl)]\Xi$. If $\Xi = \Theta\{x{:}A^- \; lvl\}$, then this would be the same as $\Theta\{\!\{\Delta\}\!\}$, but if x is not in the variable domain of Ξ, then Ξ matches $[\Delta/(x{:}A^- \; lvl)]\Xi$.

$$\frac{\Delta\restriction_{lvl} \quad \Psi; \Delta \vdash M : A^- \quad \Psi; \Xi \vdash E : U \quad \Delta \; stable_L \quad [\![M/x]\!]E = E'}{\Psi; [\Delta/(x{:}A^- \; lvl)]\Xi \vdash E' : U} \; rcut$$

Using this formulation of *rcut*, it becomes unproblematic to define $[\![M/x]\!](V_1 \bullet V_2)$ as substituting M into both V_1 and V_2, as we are allowed to substitute for x even in terms where the variable cannot appear. Using this strategy, it should be possible to describe and formalize the development in this chapter with proof terms that do nothing more than capture the binding structure of derivations.

The above argument suggests that the framing-off operation is *inconvenient* to use for specifying the *rcut* part of cut admissibility, because it forces us to track where the variable ends up and

direct the computational content of cut admissibility accordingly. However, the development in this chapter shows that it is clearly possible to define cut admissibility in terms of the framing-off operation $\Theta\{\!\{\Delta\}\!\}$. That is not necessarily the case for every logic. For instance, to give a focused presentation of Reed's queue logic [Ree09c], we would need a matching construct $[\Delta/x]\Xi$ that is quite different from the framing-off operation $\Delta\{\!\{x{:}A^-\}\!\}$ used to describe the logic's left rules. I conjecture that logics where the framing-off operation is adequate for the presentation of cut admissibility are the same as those logics which can be treated in Belnap's display logic [Bel82].

Chapter 4

Substructural logical specifications

In this chapter, we design a logical framework of substructural logical specifications (SLS), a framework heavily inspired by the Concurrent Logical Framework (CLF) [WCPW02]. The framework is justified as a fragment of the logic OL_3 from Chapter 3. There are a number of reasons why we do not just use the already-specified OL_3 outright as a logical framework.

* *Formality.* The specifics of the domain of first-order quantification in OL_3 were omitted in Chapter 3, so in Section 4.1 we give a careful presentation of the term language for SLS, Spine Form LF.

* *Clarity.* The syntax constructions that we presented for OL_3 proof terms had a 1-to-1 correspondence with the sequent calculus rules; the drawback of this presentation is that large proof terms are notationally heavy and difficult to read. The proof terms we present for SLS will leave implicit some of the information present in the diacritical marks of OL_3 proof terms.

  An implementation based on these proof terms would need to consider type reconstruction and/or bidirectional typechecking to recover the omitted information, but we will not consider those issues in this dissertation.

* *Separating concurrent and deductive reasoning.* Comparing CLF to OL_3 leads us to conclude that the single most critical design feature of CLF is its omission of the proposition $\uparrow A^+$. This single omission[1] means that stable sequents in CLF or SLS are effectively restricted to have the succedent $\langle p^- \rangle\ true$ or the succedent $A^+\ lax$.

  Furthermore, any left focus when the succedent is $\langle p^- \rangle\ true$ must conclude with the rule id^-, and any left focus when the succedent is $A^+\ lax$ must conclude with $\bigcirc_L$ – without the elimination of $\uparrow A^+$, left focus in both cases could additionally conclude with the rule $\uparrow_L$. This allows derivations that prove $\langle p^- \rangle\ true$ – the *deductive fragment* of CLF or SLS – to adequately represent deductive systems, conservatively extending deductive logical frameworks like LF and LLF. Derivations that prove $A^+\ lax$, on the other hand, fall into the *concurrent fragment* of CLF and SLS and can encode evolving systems. These fragments have interesting logic programming interpretations, which we explore in Section 4.6.

[1] In our development, the omission of right-permeable propositions p^-_{lax} from OL_3 is equally important, but permeable propositions as we have presented them in Section 2.5.4 were not a relevant consideration in the design of CLF.

* *Partial proofs.* The design of CLF makes it difficult to reason about and manipulate the proof terms corresponding to partial evaluations of evolving systems in the concurrent fragment: the proof terms in CLF correspond to complete proofs and partial evaluations naturally correspond to partial proofs.

  The syntax of SLS is designed to support the explicit representation of partial OL_3 proofs. The omission of the propositions $\mathbf{0}$, $A^+ \oplus B^+$, and the restrictions we place on $t \doteq_\tau s$ are made in the service of presenting a convenient and simple syntax for partial proofs. The three syntactic objects representing partial proofs, *patterns* (Section 4.2.4), *steps*, and *traces* (Section 4.2.6), allow us to treat proof terms for evolving systems as first-class members of SLS.

  The removal of $\mathbf{0}$ and $A^+ \oplus B^+$, and the restrictions we place on $t \doteq_\tau s$, also assist in imposing an equivalence relation, *concurrent equality*, on SLS terms in Section 4.3. Concurrent equality is a coarser equivalence relation than the α-equivalence of OL_3 terms.

* *Removal of* $\top$. The presence of $\top$ causes pervasive problems in the design of substructural logical frameworks. Many of these problems arise at the level of implementation and type reconstruction, which motivated Schack-Nielsen to remove $\top$ from the Celf implementation of CLF [SN11]. Even though those considerations are outside the scope of this dissertation, the presence of $\top$ causes other pervasive difficulties: for instance, the presence of $\top$ complicates the discussion of concurrent equality in CLF. We therefore follow Schack-Nielsen in removing $\top$ from SLS.

In summary, with SLS we *simplify* the presentation of OL_3 for convenience and readability, *restrict* the propositions of OL_3 to separate concurrent and deductive reasoning and to make the syntax for partial proofs feasible, and *extend* OL_3 with a syntax for partial proofs and a coarser equivalence relation.

In Section 4.1 we review the term language for SLS, Spine Form LF. In Section 4.2 we present SLS as a fragment of OL_3, and in Section 4.3 we discuss concurrent equality. In Section 4.4 we adopt the methodology of adequate encoding from LF to SLS, in the process introducing *generative signatures*, which play a starring role in Chapter 9. In Section 4.5 we cover the SLS prototype implementation, and in Section 4.6 we review some intuitions about logic programming in SLS. Finally, in Section 4.7, we discuss some of the decisions reflected in the design of SLS and how some decisions could have been potentially been made differently.

4.1 Spine Form LF as a term language

Other substructural logical frameworks, like Cervesato and Pfenning's LLF [CP02], Polakow's OLF [Pol01], and Watkins et al.'s CLF [WCPW02] are *fully-dependent type theories*: the language of terms (that is, the domain of first-order quantification) is the same as the language of proof terms, the representatives of logical derivations. The logical framework SLS presented in this chapter breaks from this tradition – a choice we discuss further in Section 4.7.3. The domain of first-order quantification, which was left unspecified in Chapter 3, will be presently described as Spine Form LF, a well-understood logical framework derived from the normal forms of the purely persistent type theory LF [HHP93].

All the information in this section is standard and adapted from various sources, especially Harper, Honsell, and Plotkin's original presentation of LF [HHP93], Cervesato and Pfenning's discussion of spine form terms [CP02], Watkins et al.'s presentation of the canonical forms of CLF [WCPW02], Nanevski et al.'s dependent contextual modal type theory [NPP08], Harper and Licata's discussion of Canonical LF [HL07], and Reed's spine form presentation of HLF [Ree09a].

It would be entirely consistent for us to appropriate Harper and Licata's Canonical LF presentation instead of presenting Spine Form LF. Nevertheless, a spine-form presentation of canonical LF serves to make our presentation more uniform, as spines are used in the proof term language of SLS. Canonical term languages like Canonical LF correspond to normal natural deduction presentations of logic, whereas spine form term languages correspond to focused sequent calculus presentations like the ones we have considered thus far.

4.1.1 Core syntax

The syntax of Spine Form LF is extended in two places to handle SLS: rules $\mathsf{r} : A^-$ in the signature contain negative SLS types A^- (though it would be possible to separate out the LF portion of signatures from the SLS rules), and several new base kinds are introduced for the sake of SLS – prop, prop ord, prop lin, and prop pers.

Signatures	$\Sigma ::= \cdot \mid \Sigma, \mathsf{c} : \tau \mid \Sigma, \mathsf{a} : \kappa \mid \Sigma, \mathsf{r} : A^-$
Variables	$a, b ::= \ldots$
Variable contexts	$\Psi ::= \cdot \mid \Psi, a{:}\tau$
Kinds	$\kappa ::= \Pi a{:}\tau.\kappa \mid \mathsf{type} \mid \mathsf{prop} \mid \mathsf{prop\ ord} \mid \mathsf{prop\ lin} \mid \mathsf{prop\ pers}$
Types	$\tau ::= \Pi a{:}\tau.\tau' \mid \mathsf{a} \cdot sp$
Heads	$h ::= a \mid \mathsf{c}$
Normal terms	$t, s ::= \lambda a.t \mid h \cdot sp$
Spines	$sp ::= t; sp \mid ()$
Substitutions	$\sigma ::= \cdot \mid t/a, \sigma \mid b/\!\!/a, \sigma$

Types τ and kinds κ overlap, and will be referred to generically as *classifiers* ν when it is convenient to do so; types and kinds can be seen as refinements of classifiers. Another important refinement are *atomic classifiers* $\mathsf{a} \cdot sp$, which we abbreviate as p.

LF spines sp are just sequences of terms $(t_1; (\ldots; (t_n; ()) \ldots))$; we follow common convention and write $h\, t_1 \ldots t_n$ as a convenient shorthand for the atomic term $h \cdot (t_1; \ldots; (t_n; ()) \ldots)$; similarly, we will write $\mathsf{a}\, t_1 \ldots t_n$ as a shorthand for atomic classifiers $\mathsf{a} \cdot (t_1; (\ldots; (t_n; ()) \ldots))$. This shorthand is given a formal justification in [CP02]; we will use the same shorthand for SLS proof terms in Section 4.2.5.

4.1.2 Simple types and hereditary substitution

In addition to LF types like $\Pi a{:}(\Pi z{:}(\mathsf{a1} \cdot sp_1).\, (\mathsf{a2} \cdot sp_2)).\, \Pi y{:}(\mathsf{a3} \cdot sp_3).\, (\mathsf{a4} \cdot sp_4)$, both Canonical LF and Spine Form LF take *simple types* into consideration. The simple type corresponding

$$\boxed{t \circ sp}$$

$$(\lambda a.t') \circ (t; sp) = [\![t/a]\!]t' \circ sp$$
$$h \cdot sp \circ () = h \cdot sp$$

$$\boxed{[\![t/a]\!]\,sp}$$

$$[\![t/a]\!](t'; sp) = [\![t/a]\!]t'; [\![t/a]\!]\,sp$$
$$[\![t/a]\!]() = ()$$

$$\boxed{[\![t/a]\!]t'}$$

$$[\![t/a]\!](\lambda y.t') = \lambda b.\,[\![t/a]\!]t' \qquad (a \neq b)$$
$$[\![t/a]\!](a \cdot sp) = t \circ [\![t/a]\!]\,sp$$
$$[\![t/a]\!](h \cdot sp) = h \cdot [\![t/a]\!]\,sp \qquad (\mathit{if}\ h \neq a)$$

Figure 4.1: Hereditary substitution on terms, spines, and classifiers

to the type above is $(\mathsf{a1} \to \mathsf{a2}) \to \mathsf{a3} \to \mathsf{a4}$, where $\to$ associates to the right. The simple type associated with the LF type τ is given by the function $|\tau|^- = \tau_s$, where $|\mathsf{a} \cdot sp|^- = \mathsf{a}$ and $|\Pi a{:}\tau.\tau'|^- = |\tau|^- \to |\tau'|^-$.

Variables and constants are treated as having an intrinsic simple type; these intrinsic simple types are sometimes written explicitly as annotations a^{τ_s} or c^{τ_s} (see [Pfe08] for an example), but we will leave them implicit. An atomic term $h\,t_1 \ldots t_n$ must have a simple atomic type a. This means that the head h must have simple type $\tau_{s1} \to \ldots \to \tau_{sn} \to \mathsf{a}$ and each t_i much have simple type τ_{si}. Similarly, a lambda term $\lambda a.t$ must have simple type $\tau_s \to \tau'_s$ where a is a variable with simple type τ_s and t has simple type τ'_s.

Simple types, which are treated in full detail elsewhere [HL07, Ree09a], are critical because they allow us to define hereditary substitution and hereditary reduction as total functions in Figure 4.1. Intrinsically-typed Spine Form LF terms correspond to the proof terms for a focused presentation of (non-dependent) minimal logic. Hereditary reduction $t \circ sp$ and hereditary substitution $[\![t/a]\!]t'$, which are both implicitly indexed by the simple type τ_s of t, capture the computational content of structural cut admissibility on these proof terms. Informally, the action of hereditary substitution is to perform a substitution into a term and then continue to reduce any β-redexes that would introduced by a traditional substitution operation. Therefore, $[\![\lambda x.x/f]\!](\mathsf{a}\,(f\,\mathsf{b})\,(f\,\mathsf{c}))$ is not $\mathsf{a}\,((\lambda x.x)\,\mathsf{b})\,((\lambda x.x)\,\mathsf{c})$ – that's not even a syntactically well-formed term according to the grammar for Spine Form LF. Rather, the result of that hereditary substitution is $\mathsf{a}\,\mathsf{b}\,\mathsf{c}$.

4.1.3 Judgments

Hereditary substitution is necessary to define simultaneous substitution into types and terms in Figure 4.2. We will treat simultaneous substitutions in a mostly informal way, relying on the more careful treatment by Nanevski et al. [NPP08]. A substitution takes every variable in the context and either substitutes a term for it (the form $\sigma, t/a$) or substitutes another variable for it (the form $\sigma, b/\!\!/a$). The latter form is helpful for defining identity substitutions, which we write

$$\boxed{\sigma(sp)}$$

$$\begin{aligned}
\sigma(t'; sp) &= \sigma(t'); \sigma(sp) \\
\sigma() &= ()
\end{aligned}$$

$$\boxed{\sigma(t')}$$

$$\begin{aligned}
\sigma(\lambda a.t') &= \lambda a.\,(\sigma, a/\!\!/a)(t') && (a\#\sigma) \\
\sigma(a \cdot sp) &= t \circ \sigma(sp) && t/a \in \sigma \\
\sigma(a \cdot sp) &= b \cdot \sigma(sp) && b/\!\!/a \in \sigma \\
\sigma(\mathsf{c} \cdot sp) &= \mathsf{c} \cdot \sigma(sp)
\end{aligned}$$

$$\boxed{\sigma\nu}$$

$$\begin{aligned}
\sigma(\Pi b{:}\nu.\nu') &= \Pi b{:}\sigma\nu.\,(\sigma, b/\!\!/b)\nu' && (a \neq b) \\
\sigma(\mathsf{type}) &= \mathsf{type} \\
\sigma(\mathsf{prop}) &= \mathsf{prop} \\
\sigma(\mathsf{prop\ ord}) &= \mathsf{prop\ ord} \\
\sigma(\mathsf{prop\ lin}) &= \mathsf{prop\ lin} \\
\sigma(\mathsf{prop\ pers}) &= \mathsf{prop\ pers} \\
\sigma(\mathsf{a} \cdot sp) &= \mathsf{a} \cdot \sigma sp
\end{aligned}$$

Figure 4.2: Simultaneous substitution on terms, spines, and classifiers

as id or id_Ψ, as well as generic substitutions $[t/a]$ that act like the identity on all variables except for a; the latter notation is used in the definition of LF typing in in Figure 4.3, which is adapted to Spine Form LF from Harper and Licata's Canonical LF presentation [HL07]. The judgments $a\#\sigma$, $a\#\Psi$, $\mathsf{c}\#\Sigma$, $\mathsf{a}\#\Sigma$, and $\mathsf{r}\#\Sigma$ assert that the relevant variable or constant does not already appear in the context Ψ (as a binding $a{:}\tau$), the signature Σ (as a declaration $\mathsf{c} : \tau$, $\mathsf{a} : \nu$, or $\mathsf{r} : A^-$), or the substitution σ (as a binding t/a or $b/\!\!/a$).

All the judgments in Figure 4.3 are indexed by a transitive *subordination relation* $\mathcal{R}$, similar to the one introduced by Virga in [Vir99]. The subordination relation is used to determine if a term or variable of type τ_1 can be a (proper) subterm of a term of type τ_2. Uses of subordination appear in the definition of well-formed equality propositions $t \doteq_\tau s$ in Section 4.2, in the preservation proofs in Section 9.5, and in adequacy arguments (as discussed in [HL07]). We treat $\mathcal{R}$ as a binary relation on type family constants. Let $\mathsf{head}(\tau) = \mathsf{a}$ if $\tau = \Pi a_1{:}\tau_1 \ldots \Pi a_m{:}\tau_m.\,\mathsf{a} \cdot sp$. The signature formation operations depend on three judgments. The index subordination judgment, $\kappa \sqsubset_\mathcal{R} \mathsf{a}$, relates type family constants to types. It is always the case that $\kappa = \Pi a_1{:}\tau_1 \ldots \Pi a_n{:}\tau_n.\mathsf{type}$, and the judgment $\kappa \sqsubset_\mathcal{R} \mathsf{a}$ holds if $(\mathsf{head}(\tau_i), \mathsf{a}) \in \mathcal{R}$ for $1 \leq i \leq n$. The type subordination judgment $\tau \prec_\mathcal{R} \tau'$ holds if $(\mathsf{head}(\tau), \mathsf{head}(\tau')) \in \mathcal{R}$, and the judgment $\tau \preceq_\mathcal{R} \tau'$ is the symmetric extension of this relation.

In Figure 4.3, we define the formation judgments for LF. The first formation judgment is $\vdash_\mathcal{R} \Sigma\,\mathsf{sig}$, which takes a context Σ and determines whether it is well-formed. The premise $\tau \prec_\mathcal{R} \tau$ is used in the definition of term constants to enforce that only self-subordinate types can have constructors. This, conversely, means that types that are not self-subordinate can only

$\boxed{\vdash_{\mathcal{R}} \Sigma\,\mathsf{sig}}$

$$\frac{}{\vdash_{\mathcal{R}} \cdot\,\mathsf{sig}} \qquad \frac{\vdash_{\mathcal{R}} \Sigma\,\mathsf{sig} \quad \cdot \vdash_{\Sigma,\mathcal{R}} \tau\,\mathsf{type} \quad \tau \prec_{\mathcal{R}} \tau \quad \mathsf{c}\#\Sigma}{\vdash_{\mathcal{R}} (\Sigma, \mathsf{c} : \tau)\,\mathsf{sig}}$$

$$\frac{\vdash_{\mathcal{R}} \Sigma\,\mathsf{sig} \quad \cdot \vdash_{\Sigma,\mathcal{R}} \kappa\,\mathsf{kind} \quad \kappa \sqsubset_{\mathcal{R}} \mathsf{a} \quad \mathsf{a}\#\Sigma}{\vdash_{\mathcal{R}} (\Sigma, \mathsf{a} : \kappa)\,\mathsf{sig}} \qquad \frac{\vdash_{\mathcal{R}} \Sigma\,\mathsf{sig} \quad \cdot;\cdot \vdash_{\Sigma,\mathcal{R}} A^{-}\,\mathsf{prop}^{-} \quad \mathsf{r}\#\Sigma}{\vdash_{\mathcal{R}} (\Sigma, \mathsf{r} : A^{-})\,\mathsf{sig}}$$

$\boxed{\vdash_{\Sigma,\mathcal{R}} \Psi\,\mathsf{ctx}}$ – presumes $\vdash_{\mathcal{R}} \Sigma\,\mathsf{sig}$

$$\frac{}{\vdash_{\Sigma,\mathcal{R}} \cdot\,\mathsf{ctx}} \qquad \frac{\vdash_{\Sigma,\mathcal{R}} \Psi\,\mathsf{ctx} \quad \Psi \vdash_{\Sigma,\mathcal{R}} \tau\,\mathsf{type} \quad a\#\Psi}{\vdash_{\Sigma,\mathcal{R}} (\Psi, a{:}\tau)\,\mathsf{ctx}}$$

$\boxed{\Psi \vdash_{\Sigma,\mathcal{R}} \kappa\,\mathsf{kind}}$ – presumes $\vdash_{\Sigma,\mathcal{R}} \Psi\,\mathsf{ctx}$

$$\frac{\Psi \vdash_{\Sigma,\mathcal{R}} \tau\,\mathsf{type} \quad \Psi, a{:}\tau \vdash_{\Sigma,\mathcal{R}} \kappa\,\mathsf{kind}}{\Psi \vdash_{\Sigma,\mathcal{R}} (\Pi a{:}\tau.\kappa)\,\mathsf{kind}} \qquad \frac{}{\Psi \vdash_{\Sigma,\mathcal{R}} \mathsf{type}\,\mathsf{kind}} \qquad \frac{}{\Psi \vdash_{\Sigma,\mathcal{R}} \mathsf{prop}\,\mathsf{kind}}$$

$$\frac{}{\Psi \vdash_{\Sigma,\mathcal{R}} (\mathsf{prop}\,\mathsf{ord})\,\mathsf{kind}} \qquad \frac{}{\Psi \vdash_{\Sigma,\mathcal{R}} (\mathsf{prop}\,\mathsf{lin})\,\mathsf{kind}} \qquad \frac{}{\Psi \vdash_{\Sigma,\mathcal{R}} (\mathsf{prop}\,\mathsf{pers})\,\mathsf{kind}}$$

$\boxed{\Psi \vdash_{\Sigma,\mathcal{R}} \tau\,\mathsf{type}}$ – presumes $\vdash_{\Sigma,\mathcal{R}} \Psi\,\mathsf{ctx}$

$$\frac{\Psi \vdash_{\Sigma,\mathcal{R}} \tau\,\mathsf{type} \quad \Psi, a{:}\tau \vdash_{\Sigma,\mathcal{R}} \tau'\,\mathsf{type} \quad \tau \preceq_{\mathcal{R}} \tau'}{\Psi \vdash_{\Sigma,\mathcal{R}} (\Pi a{:}\tau.\tau')\,\mathsf{type}} \qquad \frac{a{:}\kappa \in \Sigma \quad \Psi, [\kappa] \vdash_{\Sigma,\mathcal{R}} sp : \mathsf{type}}{\Psi \vdash_{\Sigma,\mathcal{R}} (\mathsf{a} \cdot sp)\,\mathsf{type}}$$

$\boxed{\Psi \vdash_{\Sigma,\mathcal{R}} t : \tau}$ – presumes $\Psi \vdash_{\Sigma,\mathcal{R}} \tau\,\mathsf{type}$

$$\frac{\Psi, a{:}\tau \vdash_{\Sigma,\mathcal{R}} t : \tau'}{\Psi \vdash_{\Sigma,\mathcal{R}} \lambda a.t : \Pi x{:}\tau.\tau'} \qquad \frac{\mathsf{c} : \tau \in \Sigma \quad \Psi, [\tau] \vdash_{\Sigma,\mathcal{R}} sp : \tau' \quad \tau' = p}{\Psi \vdash_{\Sigma,\mathcal{R}} \mathsf{c} \cdot sp : p}$$

$$\frac{a{:}\tau \in \Psi \quad \Psi, [\tau] \vdash_{\Sigma,\mathcal{R}} sp : \tau' \quad \tau' = p}{\Psi \vdash_{\Sigma,\mathcal{R}} a \cdot sp : p}$$

$\boxed{\Psi, [\nu] \vdash_{\Sigma,\mathcal{R}} sp : \nu_0}$ – presumes that either $\Psi \vdash_{\Sigma,\mathcal{R}} \nu\,\mathsf{type}$ or that $\Psi \vdash_{\Sigma,\mathcal{R}} \nu\,\mathsf{kind}$

$$\frac{}{\Psi, [\nu] \vdash_{\Sigma,\mathcal{R}} () : \nu} \qquad \frac{\Psi \vdash_{\Sigma,\mathcal{R}} t : \tau \quad [t/a]\nu = \nu' \quad \Psi, [\nu'] \vdash_{\Sigma,\mathcal{R}} sp : \nu_0}{\Psi, [\Pi a{:}\tau.\nu] \vdash_{\Sigma,\mathcal{R}} t; sp : \nu_0}$$

$\boxed{\Psi \vdash \sigma : \Psi'}$ – presumes $\vdash_{\Sigma,\mathcal{R}} \Psi\,\mathsf{ctx}$ and $\vdash_{\Sigma,\mathcal{R}} \Psi'\,\mathsf{ctx}$

$$\frac{}{\Psi \vdash_{\Sigma,\mathcal{R}} \cdot : \cdot} \qquad \frac{\Psi \vdash_{\Sigma,\mathcal{R}} \sigma : \Psi' \quad \Psi \vdash_{\Sigma,\mathcal{R}} t : \sigma\tau}{\Psi \vdash_{\Sigma,\mathcal{R}} (\sigma, t/a) : \Psi', a{:}\tau} \qquad \frac{\Psi \vdash_{\Sigma,\mathcal{R}} \sigma : \Psi' \quad b{:}\sigma\tau \in \Psi}{\Psi \vdash_{\Sigma,\mathcal{R}} (\sigma, b/\!\!/a) : \Psi', a{:}\tau}$$

Figure 4.3: LF formation judgments ($\tau' = p$ refers to α-equivalence)

be inhabited by variables a, which is important for one of the two types of equality $t \doteq_\tau s$ that SLS supports. The judgments $\vdash_{\Sigma,\mathcal{R}} \Psi\,\mathsf{ctx}$, $\Psi \vdash_{\Sigma,\mathcal{R}} \kappa\,\mathsf{kind}$, and $\Psi \vdash_{\Sigma,\mathcal{R}} \tau\,\mathsf{type}$ take contexts Ψ, kinds κ, and types τ and ensure that they are well-formed in the current signature or (if applicable) context. The judgment $\Psi \vdash_{\Sigma,\mathcal{R}} t{:}\tau$ takes a term and a type and typechecks the term against the type, and the judgment $\Psi \vdash_{\Sigma,\mathcal{R}} \sigma : \Psi'$ checks that a substitution σ can transport objects (terms, types, etc.) defined in the context Ψ' to objects defined in Ψ.

The judgment $\Psi, [\nu] \vdash_{\Sigma,\mathcal{R}} sp : \nu_0$ is read a bit differently than these other judgments. The notation, first of all, is meant to evoke the (exactly analogous) left-focus judgments from Chapters 2 and 3. In most other sources (for example, in [CP02]) this judgment is instead written as $\Psi \vdash_{\Sigma,\mathcal{R}} sp : \nu > \nu_0$. In either case, we read this judgment as checking a spine sp against a classifier ν (actually either a type τ or a kind κ) and *synthesizing* a return classifier ν_0. In other words, ν_0 is an output of the judgment $\Psi, [\nu] \vdash_{\Sigma,\mathcal{R}} sp : \nu_0$, and given that this judgment presumes that either $\Psi \vdash_{\Sigma,\mathcal{R}} \nu\,\mathsf{type}$ or $\Psi \vdash_{\Sigma,\mathcal{R}} \nu\,\mathsf{kind}$, it *ensures* that either $\Psi \vdash_{\Sigma,\mathcal{R}} \nu_0\,\mathsf{type}$ or $\Psi \vdash_{\Sigma,\mathcal{R}} \nu_0\,\mathsf{kind}$, where the classifiers of ν and ν_0 (type or kind) always match. It is because ν_0 is an output that we add an explicit premise to check that $\tau' = p$ in the typechecking rule for $\mathsf{c} \cdot sp$; this equality refers to the α-equality of Spine Form LF terms.

There are a number of well-formedness theorems that we need to consider, such as the fact that substitutions compose in a well-behaved way and that hereditary substitution is always well-typed. However, as these theorems are adequately covered in the aforementioned literature on LF, we will proceed with using LF as a term language and will treat term-level operations like substitution somewhat informally.

We will include annotations for the signature Σ and the subordination relation $\mathcal{R}$ in the definitions of this section and the next one. In the following sections and chapters, however, we will often leave the signature Σ implicit when it is unambiguous or unimportant. We will almost always leave the subordination relation implicit; we can assume where applicable that we are working with the *strongest* (that is, the smallest) subordination relation for the given signature [HL07].

4.1.4 Adequacy

Adequacy was the name given by Harper, Honsell, and Plotkin to the methodology of connecting inductive definitions to the canonical forms of a particular type family in LF. Consider, as a standard example, the untyped lambda calculus, which is generally specified by a BNF grammar such as the following:

$$e ::= x \mid \lambda x.e \mid e_1\, e_2$$

We can adequately encode this language of terms into LF (with a subordination relation $\mathcal{R}$ such that $(\mathsf{exp}, \mathsf{exp}) \in \mathcal{R}$) by giving the following signature:

$$\begin{aligned}
\Sigma = \cdot,\ & \\
& \mathsf{exp} : \mathsf{type}, \\
& \mathsf{app} : \Pi a{:}\mathsf{exp}.\ \Pi b{:}\mathsf{exp}.\ \mathsf{exp}, \\
& \mathsf{lam} : \Pi a{:}(\Pi b{:}\mathsf{exp}.\ \mathsf{exp}).\ \mathsf{exp}
\end{aligned}$$

Note that the variables a and b are bound by Π-quantifiers in the declaration of app and lam but never used. The usual convention is to abbreviate $\Pi a{:}\tau.\tau'$ as $\tau \to \tau'$ when a is not free in τ', which would give app type $\mathsf{exp} \to \mathsf{exp} \to \mathsf{exp}$ and lam type $(\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp}$.

Theorem 4.1 (Adequacy for terms)**.** *Up to standard α-equivalence, there is a bijection between expressions e (with free variables in the set $\{x_1, \ldots, x_n\}$) and Spine Form LF terms t such that* $x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash t : \mathsf{exp}$.

Proof. By induction on the structure of the inductive definition of e in the forward direction and by induction on the structure of terms t with type exp in the reverse direction. □

We express the constructive content of this theorem as an invertible function $\ulcorner e \urcorner = t$ from object language terms e to representations LF terms t of type exp:

* $\ulcorner x \urcorner = x$,
* $\ulcorner e_1\, e_2 \urcorner = \mathsf{app} \ulcorner e_1 \urcorner \ulcorner e_2 \urcorner$, and
* $\ulcorner \lambda x.e \urcorner = \mathsf{lam}\, \lambda x. \ulcorner e \urcorner$.

If we had also defined substitution $[e/x]e'$ on terms, it would be necessary to show that the bijection is compositional: that is, that $[\ulcorner e \urcorner / x] \ulcorner e' \urcorner = \ulcorner [e/x]e' \urcorner$. Note that adequacy critically depends on the context having the form $x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp}$. If we had a context with a variable $y{:}(\mathsf{exp} \to \mathsf{exp})$, then we could form an LF term $y\,(\mathsf{lam}\, \lambda x.x)$ with type exp that does *not* adequately encode any term e in the untyped lambda calculus.

One of the reasons subordination is important in practice is that it allows us to consider the adequate encoding of expressions in contexts Ψ that have other variables $x{:}\tau$ as long as $(\mathsf{head}(\tau), \mathsf{exp}) \notin \mathcal{R}$. If $\Psi, x{:}\tau \vdash_{\Sigma,\mathcal{R}} t : \mathsf{exp}$ and $\tau \not\preceq_{\mathcal{R}} \mathsf{exp}$, then x cannot be free in t, so $\Psi \vdash_{\Sigma,\mathcal{R}} t : \mathsf{exp}$ holds as well. By iterating this procedure, it may be possible to strengthen a context Ψ into one of the form $x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp}$, in which case we can conclude that $t = \ulcorner e \urcorner$ for some untyped lambda calculus term e.

4.2 The logical framework SLS

In this section, we will describe the restricted set of polarized OL_3 propositions and focused OL_3 proof terms that make up the logical framework SLS. For the remainder of the dissertation, we will work exclusively with the following positive and negative SLS propositions, which are a syntactic refinement of the positive and negative propositions of polarized OL_3:

$$A^+, B^+, C^+ ::= p^+ \mid p^+_{eph} \mid p^+_{pers} \mid {\downarrow}A^- \mid {¡}A^- \mid !A^- \mid \mathbf{1} \mid A^+ \bullet B^+ \mid \exists a{:}\tau.A^+ \mid t \doteq_\tau s$$
$$A^-, B^-, C^- ::= p^- \mid \bigcirc A^+ \mid A^+ \rightarrowtail B^- \mid A^+ \twoheadrightarrow B^- \mid A^- \,\&\, B^- \mid \forall a{:}\tau.A^-$$

We now have to deal with a point of notational dissonance: all existing work on CLF, all existing implementations of CLF, and the prototype implementation of SLS (Section 4.5) use the notation $\{A^+\}$ for the connective internalizing the judgment $A^+\ lax$, which we have written as $\bigcirc A^+$, following Fairtlough and Mendler [FM97]. The traditional notation overloads curly braces, which

$\boxed{\Psi \vdash_{\Sigma,\mathcal{R}} \mathcal{C}\ \mathsf{satisfiable}}$ – presumes $\vdash_{\Sigma,\mathcal{R}} \Psi$ ctx, that the terms in

$$\frac{}{\Psi \vdash_{\Sigma,\mathcal{R}} \cdot\ \mathsf{satisfiable}} \qquad \frac{\Psi \vdash_{\Sigma,\mathcal{R}} t{:}\tau \quad \Psi \vdash_{\Sigma,\mathcal{R}} \mathcal{C}\ \mathsf{satisfiable}}{\Psi \vdash_{\Sigma,\mathcal{R}} (\mathcal{C}, t \doteq_\tau t)\ \mathsf{satisfiable}}$$

$$\frac{a{:}p \in \Psi \quad \Psi \vdash_{\Sigma,\mathcal{R}} t : p \quad \Psi' \vdash_{\Sigma,\mathcal{R}} [t/a] : \Psi \quad \Psi' \vdash_{\Sigma,\mathcal{R}} [t/a]\mathcal{C}\ \mathsf{satisfiable}}{\Psi \vdash_{\Sigma,\mathcal{R}} (\mathcal{C}, a \doteq_p t)\ \mathsf{satisfiable}}$$

Figure 4.4: Equality constraints (used to support notational definitions)

we also use for the context-framing notation $\Theta\{\Delta\}$ introduced in Section 3.2. We will treat $\bigcirc A^+$ and $\{A^+\}$ as synonyms in SLS, preferring the former in this chapter and the latter afterwards.

Positive ordered atomic propositions p^+ are atomic classifiers $\mathsf{a}\, t_1 \ldots t_n$ with kind $\mathsf{prop\ ord}$, positive linear and persistent atomic propositions p^+_{eph} and p^+_{pers} are (respectively) atomic classifiers with kind $\mathsf{prop\ lin}$ and $\mathsf{prop\ pers}$, and negative ordered atomic propositions p^- are atomic classifiers with kind prop. From this point on, we will unambiguously refer to atomic propositions p^- as negative atomic propositions, omitting "ordered." Similarly, we will refer to atomic propositions p^+, p^+_{eph}, and p^+_{pers} collectively as positive atomic propositions but individually as ordered, linear, and persistent propositions, respectively, omitting "positive." ("Mobile" and "ephemeral" will continue to be used as synonyms for "linear.")

4.2.1 Propositions

The formation judgments for SLS types are given in Figure 4.5. As discussed in the introduction to this chapter, the removal of $\uparrow A^+$ and p^-_{lax} is fundamental to the separation of the deductive and concurrent fragments of SLS; most of the other restrictions made to the language are for the purpose of giving partial proofs a list-like structure. In particular, all positive propositions whose left rules have more or less than one premise are restricted. The propositions $\mathbf{0}$ and $A^+ \oplus B^+$ are excluded from SLS to this end, and we must place rather draconian restrictions on the use of equality in order to ensure that $\doteq_L$ can always be treated as having exactly one premise.

The formation rules for propositions are given in Figure 4.5. Much of the complexity of this presentation, such as the existence of an additional constraint context $\mathcal{C}$, described in Figure 4.4, is aimed at allowing the inclusion of equality in SLS in a sufficiently restricted form. The intent of these restrictions is to ensure that, whenever we decompose a positive proposition $s \doteq t$ on the left, we have that s is some variable a in the context and that a is not free in t. When this is the case, $[t/a]$ is always a most general unifier of $s = a$ and t, which in turn means that the left rule for equality in OL_3

$$\frac{\forall(\Psi' \vdash \sigma : \Psi).\quad \sigma t = \sigma s \quad \longrightarrow \quad \Psi'; \sigma\Theta\{\cdot\} \vdash \sigma U}{\Psi; \Theta\{\!\{t \doteq_\tau s\}\!\} \vdash U}\ \doteq_L$$

is equivalent to a much simpler rule:

$$\frac{\Psi, [t/a]\Psi'; [t/a]\Theta\{\cdot\} \vdash [t/a]U}{\Psi, a{:}\tau, \Psi'; \Theta\{\!\{a \doteq_\tau t\}\!\} \vdash U}\ \doteq_{yes}$$

$\boxed{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} A^+ \ \mathsf{prop}^+}$ – presumes $\vdash_{\Sigma,\mathcal{R}} \Psi$ ctx and that $\Psi \vdash \mathcal{C}$ satisfiable.

$$\frac{\mathsf{a}{:}\kappa \in \Sigma \quad \Psi, [\kappa] \vdash_{\Sigma,\mathcal{R}} sp : \mathsf{prop\ ord}}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} \mathsf{a} \cdot sp \ \mathsf{prop}^+} \qquad \frac{\mathsf{a}{:}\kappa \in \Sigma \quad \Psi, [\kappa] \vdash_{\Sigma,\mathcal{R}} sp : \mathsf{prop\ lin}}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} \mathsf{a} \cdot sp \ \mathsf{prop}^+}$$

$$\frac{\mathsf{a}{:}\kappa \in \Sigma \quad \Psi, [\kappa] \vdash_{\Sigma,\mathcal{R}} sp : \mathsf{prop\ pers}}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} \mathsf{a} \cdot sp \ \mathsf{prop}^+}$$

$$\frac{\Psi; \cdot \vdash_{\Sigma,\mathcal{R}} A^- \ \mathsf{prop}^-}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} {\downarrow}A^- \ \mathsf{prop}^+} \qquad \frac{\Psi; \cdot \vdash_{\Sigma,\mathcal{R}} A^- \ \mathsf{prop}^-}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} {\text{¡}}A^- \ \mathsf{prop}^+} \qquad \frac{\Psi; \cdot \vdash_{\Sigma,\mathcal{R}} A^- \ \mathsf{prop}^-}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} !A^- \ \mathsf{prop}^+} \qquad \frac{}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} \mathbf{1} \ \mathsf{prop}^+}$$

$$\frac{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} A^+ \ \mathsf{prop}^+ \quad \Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} B^+ \ \mathsf{prop}^+}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} A^+ \bullet B^+ \ \mathsf{prop}^+}$$

$$\frac{\Psi \vdash_{\Sigma,\mathcal{R}} \tau \ \mathsf{type} \quad (t = a \text{ or } \Psi \vdash_{\Sigma,\mathcal{R}} t : \tau) \quad \Psi, a{:}\tau; \mathcal{C}, a \doteq_\tau t \vdash_{\Sigma,\mathcal{R}} A^+ \ \mathsf{prop}^+}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} \exists a{:}\tau.A^+ \ \mathsf{prop}^+}$$

$$\frac{\Psi \vdash_{\Sigma,\mathcal{R}} p \ \mathsf{type} \quad a{:}p \in \Psi \quad b{:}p \in \Psi \quad p \not\leftarrow_{\mathcal{R}} p}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} a \doteq_p b \ \mathsf{prop}^+} \qquad \frac{t \doteq_p s \in \mathcal{C} \quad \Psi \vdash_{\Sigma,\mathcal{R}} s : p}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} t \doteq_p s \ \mathsf{prop}^+}$$

$\boxed{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} A^- \ \mathsf{prop}^-}$ – presumes $\vdash_{\Sigma,\mathcal{R}} \Psi$ ctx and that $\Psi \vdash \mathcal{C}$ satisfiable.

$$\frac{\mathsf{a}{:}\kappa \in \Sigma \quad \Psi, [\kappa] \vdash_{\Sigma,\mathcal{R}} sp : \mathsf{prop}}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} \mathsf{a} \cdot sp \ \mathsf{prop}^-} \qquad \frac{\Psi; \cdot \vdash_{\Sigma,\mathcal{R}} A^+ : \mathsf{prop}^+}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} \bigcirc A^+ \ \mathsf{prop}^-}$$

$$\frac{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} A^+ \ \mathsf{prop}^+ \quad \Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} B^- \ \mathsf{prop}^-}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} A^+ \rightarrowtail B^- \ \mathsf{prop}^-} \qquad \frac{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} A^+ \ \mathsf{prop}^+ \quad \Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} B^- \ \mathsf{prop}^-}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} A^+ \twoheadrightarrow B^- \ \mathsf{prop}^-}$$

$$\frac{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} A^- \ \mathsf{prop}^- \quad \Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} B^- \ \mathsf{prop}^-}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} A^- \ \& \ B^- \ \mathsf{prop}^-}$$

$$\frac{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} \tau \ \mathsf{type} \quad (t = a \text{ or } \Psi \vdash_{\Sigma,\mathcal{R}} t : \tau) \quad \Psi, a{:}\tau; \mathcal{C}, a \doteq_\tau t \vdash_{\Sigma,\mathcal{R}} A^- \ \mathsf{prop}^-}{\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} \forall a{:}\tau.A^- \ \mathsf{prop}^-}$$

Figure 4.5: SLS propositions

Usually, when we require the "existence of most general unifiers," that signals that a most general unifier must exist if any unifier exists. The condition we are requiring is much stronger: for the unification problems we will encounter due to the $\doteq_L$ rule, a most general unifier *must* exist. Allowing unification problems that could fail would require us to consider positive inversion rules with zero premises, and the proposition $\mathbf{0}$ was excluded from SLS precisely to prevent us from needing to deal with positive inversion rules with zero premises.[2]

There are two distinct conditions under which we can be sure that unification problems always have a most general solution – when equality is performed over *pure variables* and when equality is used as a *notational definition* [PS99a]. Equality of pure variable types is used in the destination-adding transformation in Chapter 7, and notational definitions are used extensively in Chapter 8.

Pure variables Equality at an atomic type p that is *not subordinate to itself* ($p \not\prec_{\mathcal{R}} p$) is always allowed. This is reflected in the first formation rule for $t \doteq s$ in Figure 4.5.

Types that are not self-subordinate can only be inhabited by variables: that is, if $p \not\prec_{\mathcal{R}} p$ and $\Psi \vdash_{\Sigma,\mathcal{R}} t : p$, then $t = a$ where $a{:}p \in \Psi$. For any unification problem $a \doteq b$, both $[a/b]$ and $[b/a]$ are most general unifiers.

Notational definitions Using equality as a notational definition allows us manipulate propositions in ways that have no effect on the structure of synthetic inference rules. When we check a universal quantifier $\forall a{:}p.A^-$ or existential quantifier $\exists a{:}p.A^+$, we are allowed to introduce one term t that will be forced, by the use of equality, to be unified with this newly-introduced variable. By adding the $a \doteq_p t$ to the set of constraints $\mathcal{C}$, we allow ourselves to mention $a \doteq_p t$ later on in the proposition by using the second formation rule for equality in Figure 4.5.

The notational definition $a \doteq_p t$ must be reachable from its associated quantifier without crossing a shift or an exponential – in Andreoli's terms, it must be in the same *monopole* (Section 2.4). This condition, which it might be possible to relax at the cost of further complexity elsewhere in the presentation, is enforced by the formation rules for shifts and exponentials, which clear the context $\mathcal{C}$ in their premise. The proposition $\forall a. {\downarrow}(\mathsf{p}\, a) \rightarrowtail a \doteq t \rightarrowtail \mathsf{p}\, t$ satisfies this condition but $\forall a. \bigcirc(a \doteq t)$ does not ($\bigcirc$ breaks focus), and the proposition $\bigcirc(\exists a. a \doteq_p t)$ satisfies this condition but $\bigcirc(\exists a. {\uparrow}(a \doteq t \rightarrowtail \mathsf{p}\, a))$ does not ($\uparrow$ breaks focus). The rule $\bigcirc(\exists a{:}p.\, a \doteq_p \mathsf{s}\, a)$ doesn't pass muster because the term t must be well-formed in a context that does not include a – this is related to the *occurs check* in unification. The rule $\bigcirc(\exists a.\, a \doteq t \bullet a \doteq s)$ is not well-formed if t and s are syntactically distinct. Each variable can only be notationally defined to be one term; otherwise we could encode an arbitrary unification problem $t \doteq s$.

4.2.2 Substructural contexts

Figure 4.6 describes the well-formed substructural contexts in SLS. The judgment $\Psi \vdash_{\Sigma,\mathcal{R}} T\ \mathsf{left}$ is used to check stable bindings $x{:}A^-\ lvl$ and $z{:}\langle p^+_{lvl}\rangle\ lvl$ that can appear as a part of stable, inverting, or left-focused sequents; the judgment $\Psi \vdash_{\Sigma,\mathcal{R}} \Delta\ \mathsf{stable}$ just maps this judgment over

[2]The other side of this observation is that, if we allow the proposition $\mathbf{0}$ and adapt the logical framework accordingly, it might be possible to relax the restrictions we have placed on equality.

$\boxed{\Psi \vdash_{\Sigma,\mathcal{R}} T\ \mathsf{left}}$ – presumes $\vdash_{\Sigma,\mathcal{R}} \Psi\ \mathsf{ctx}$

$$\frac{\Psi; \cdot \vdash_{\Sigma,\mathcal{R}} A^-\ \mathsf{prop}^-}{\Psi \vdash_{\Sigma,\mathcal{R}} (A^-\ lvl)\ \mathsf{left}} \qquad \frac{\mathsf{a} : \kappa \in \Sigma \quad \Psi, [\kappa] \vdash_{\Sigma,\mathcal{R}} sp : \mathsf{prop\ ord}}{\Psi \vdash_{\Sigma,\mathcal{R}} (\langle \mathsf{a} \cdot sp \rangle\ ord)\ \mathsf{left}}$$

$$\frac{\mathsf{a} : \kappa \in \Sigma \quad \Psi, [\kappa] \vdash_{\Sigma,\mathcal{R}} sp : \mathsf{prop\ lin}}{\Psi \vdash_{\Sigma,\mathcal{R}} (\langle \mathsf{a} \cdot sp \rangle\ eph)\ \mathsf{left}} \qquad \frac{\mathsf{a} : \kappa \in \Sigma \quad \Psi, [\kappa] \vdash_{\Sigma,\mathcal{R}} sp : \mathsf{prop\ pers}}{\Psi \vdash_{\Sigma,\mathcal{R}} (\langle \mathsf{a} \cdot sp \rangle\ pers)\ \mathsf{left}}$$

$\boxed{\Psi \vdash_{\Sigma,\mathcal{R}} \Delta\ \mathsf{stable}}$ – presumes $\vdash_{\Sigma,\mathcal{R}} \Psi\ \mathsf{ctx}$

$$\frac{}{\Psi \vdash_{\Sigma,\mathcal{R}} \cdot\ \mathsf{stable}} \qquad \frac{\Psi \vdash_{\Sigma,\mathcal{R}} \Delta\ \mathsf{stable} \quad \Psi \vdash_{\Sigma,\mathcal{R}} T\ \mathsf{left}}{\Psi \vdash_{\Sigma,\mathcal{R}} (\Delta, x{:}T)\ \mathsf{stable}}$$

$\boxed{\Psi \vdash_{\Sigma,\mathcal{R}} \Delta\ \mathsf{inv}}$ – presumes $\vdash_{\Sigma,\mathcal{R}} \Psi\ \mathsf{ctx}$

$$\frac{}{\Psi \vdash_{\Sigma,\mathcal{R}} \cdot\ \mathsf{inv}} \qquad \frac{\Psi \vdash_{\Sigma,\mathcal{R}} \Delta\ \mathsf{inv} \quad \Psi \vdash_{\Sigma,\mathcal{R}} T\ \mathsf{left}}{\Psi \vdash_{\Sigma,\mathcal{R}} (\Delta, x{:}T)\ \mathsf{inv}} \qquad \frac{\Psi \vdash_{\Sigma,\mathcal{R}} \Delta\ \mathsf{inv} \quad \Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} A^+\ \mathsf{prop}^+}{\Psi \vdash_{\Sigma,\mathcal{R}} (\Delta, x{:}A^+\ ord)\ \mathsf{inv}}$$

$\boxed{\Psi \vdash_{\Sigma,\mathcal{R}} \Delta\ \mathsf{infoc}}$ – presumes $\vdash_{\Sigma,\mathcal{R}} \Psi\ \mathsf{ctx}$

$$\frac{\Psi \vdash_{\Sigma,\mathcal{R}} \Delta\ \mathsf{infoc} \quad \Psi \vdash_{\Sigma,\mathcal{R}} T\ \mathsf{left}}{\Psi \vdash_{\Sigma,\mathcal{R}} (\Delta, x{:}T)\ \mathsf{infoc}} \qquad \frac{\Psi \vdash_{\Sigma,\mathcal{R}} \Delta\ \mathsf{stable} \quad \Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} A^-\ \mathsf{prop}^-}{\Psi \vdash_{\Sigma,\mathcal{R}} (\Delta, x{:}[A^-]\ ord)\ \mathsf{infoc}}$$

Figure 4.6: SLS contexts

the context. The judgment $\Psi \vdash_{\Sigma,\mathcal{R}} \Delta\ \mathsf{inv}$ describes contexts during the inversion phase, which can also contain inverting positive propositions A^+. The judgment $\Psi \vdash_{\Sigma,\mathcal{R}} \Delta\ \mathsf{infoc}$ describes a context that is stable aside from the one negative proposition in focus.

The rules for inverting and focusing on propositions in Figure 4.6 use non-empty constraint context. This is necessary because the property of being a well-formed proposition is not stable under arbitrary substitutions. Even though $\forall a{:}p.\,(a \doteq_p \mathsf{c}) \rightarrowtail A^-$ is a well-formed negative proposition according to Figure 4.5, $(a \doteq_p \mathsf{c}) \rightarrowtail A^-$ is only a well-formed proposition if we add $a \doteq_p \mathsf{c}$ to the set of constraints, and $(\mathsf{c} \doteq_p \mathsf{c}) \rightarrowtail [\mathsf{c}/a]A^-$ is only a well-formed proposition if we add $\mathsf{c} \doteq_p \mathsf{c}$ to the set of constraints.

The restrictions we make to contexts justify our continued practice of omitting the ord annotation when talking about inverting positive propositions A^+ or focused negative propositions $[A^-]$ in the context, since these context constituents only appear in conjunction with the ord judgment.

This discussion of well-formed propositions and contexts takes care of any issues dealing with variables that were swept under the rug in Chapter 3. We could stop here and use the refinement of OL_3 proof terms that corresponds to our refinement of propositions as the language

of SLS proof terms. This is not desirable for two main reasons. First, the proof terms of focused OL_3 make it inconvenient (though not impossible) to talk about concurrent equality (Section 4.3). Second, one of our primary uses of SLS in this dissertation will be to talk about *traces*, which correspond roughly to partial proofs

$$\begin{array}{c}\Psi'; \Delta' \vdash A^+\ lax \\ \vdots \\ \Psi; \Delta \vdash A^+\ lax\end{array}$$

in OL_3, where both the top and bottom sequents are stable and where A^+ is some unspecified, parametric positive proposition. Using OL_3-derived proof terms makes it difficult to talk about about and manipulate proofs of this form.

In the remainder of this section, we will present a proof term assignment for SLS that facilitates discussing concurrent equality and partial proofs. SLS proof terms are in bijective correspondence with a refinement of OL_3 proof terms when we consider complete (deductive) proofs, but the introduction of patterns and traces reconfigures the structure of derivations and proof terms.

4.2.3 Process states

A *process state* is a disembodied left-hand side of a sequent that we use to describe the intermediate states of concurrent systems. Traces, introduced in Section 4.2.6, are intended to capture the structure of partial proofs:

$$\begin{array}{c}\Psi'; \Delta' \vdash A^+\ lax \\ \vdots \\ \Psi; \Delta \vdash A^+\ lax\end{array}$$

The type of a trace will be presented as a relation between two process states. As a first cut, we can represent the initial state as $(\Psi; \Delta)$ and the final state as $(\Psi'; \Delta')$, and we can omit Ψ and just write Δ when that is sufficiently clear.

Representing a process state as merely an LF context Ψ and a substructural context Δ is insufficient because of the way equality – pure variable equality in particular – can unify distinct variables. Consider the following partial proof:

$$\begin{array}{c} b{:}p;\ \ z{:}\langle \mathsf{foo}\, b\, b\rangle\ eph \vdash (\mathsf{foo}\, b\, b)\ lax \\ \vdots \\ a{:}p, b{:}p;\ \ x{:}\bigcirc(a \doteq_\tau b)\ eph,\ \ z{:}\langle \mathsf{foo}\, a\, a\rangle\ eph \vdash (\mathsf{foo}\, a\, b)\ lax \end{array}$$

This partial proof can be constructed in one focusing stage by a left focus on x. It is insufficient to capture the first process state as $(a{:}p, b{:}p;\ \ x{:}\bigcirc(a \doteq_\tau b),\ \ z{:}\langle \mathsf{foo}\, a\, a\rangle\ eph)$ and the second process state as $(b{:}p;\ \ z{:}\langle \mathsf{foo}\, b\, b\rangle\ eph)$, as this would fail to capture that the succedent $(\mathsf{foo}\, b\, b)\ lax$ is a substitution instance of the succedent $(\mathsf{foo}\, a\, b)\ lax$. In general, if the derivation above proved some arbitrary succedent $A^+\ lax$ instead of the specific succedent $(\mathsf{foo}\, a\, b)\ lax$, then the missing subproof would have the succedent $[b/a]A^+\ lax$.

A process state is therefore written as $(\Psi; \Delta)_\sigma$ and is well-formed under the signature Σ and the subordination relation $\mathcal{R}$ if $\Psi \vdash_{\Sigma,\mathcal{R}} \Delta\ \mathsf{inv}$ (which presumes that $\vdash_{\Sigma,\mathcal{R}} \Psi\ \mathsf{ctx}$, as defined in

Figure 4.3) and if $\Psi \vdash \sigma : \Psi_0$, where Ψ_0 is some other context that represents the starting point, the context in which the disconnected succedent $A^+\ lax$ is well-formed.

$$\frac{\Psi \vdash_{\Sigma,\mathcal{R}} \Delta\ \mathsf{inv} \quad \vdash_{\Sigma,\mathcal{R}} \Psi_0 : \mathsf{ctx} \quad \Psi \vdash \sigma : \Psi_0}{\vdash_{\Sigma,\mathcal{R}} (\Psi; \Delta)_\sigma\ \mathsf{state}}$$

Taking $\Psi_0 = a{:}p, b{:}p$, the partial proof above can thus be represented as a step (Section 4.2.6) between these two process states:

$$(a{:}p, b{:}p;\ \ x{:}\bigcirc(a \doteq_\tau b),\ \ z{:}\langle \mathsf{foo}\ a\ a\rangle\ eph)_{(a/a,\,b/b)} \rightsquigarrow_{\Sigma,\mathcal{R}} (b{:}p;\ \ z{:}\langle \mathsf{foo}\ b\ b\rangle\ eph)_{(b/a,\,b/b)}$$

Substitutions are just one of several ways that we could cope with free variables in succedents; another option, discussed in Section 4.3, is to track the set of constraints $a = b$ that have been encountered by unification. When we consider traces in isolation, we will generally let $\Psi_0 = \cdot$ and $\sigma = \cdot$, which corresponds to the case where the parametric conclusion A^+ is a closed proposition. When the substitution is not mentioned, it can therefore be presumed to be empty. Additionally, when the LF context Ψ is empty or clear from the context, we will omit it as well. One further simplification is that we will occasionally omit the judgment lvl associated with a suspended positive atomic proposition $\langle p^+_{lvl}\rangle\ lvl$, but only when it is unambiguous from the current signature that p^+_{lvl} is an ordered, linear, or persistent positive atomic proposition. In the examples above, we tacitly assumed that foo was given kind $p \to p \to \mathsf{prop}\ \mathsf{lin}$ in the signature Σ when we tagged the suspended atomic propositions with the judgment eph. If it had been clear that foo was linear, then this judgment could have been omitted.

4.2.4 Patterns

A *pattern* is a syntactic entity that captures the list-like structure of left inversion on positive propositions. The OL_3 proof term for the proposition $(\exists a.\ \mathsf{p}\ a \bullet {\text{¡}}A^- \bullet {\downarrow}B^-) \rightarrowtail C^-$, is somewhat inscrutable: $\lambda^< a.\bullet\bullet\langle x\rangle.{\text{¡}}y.{\downarrow}z.N$. The SLS proof of this proposition, which uses patterns, is $(\lambda a, x, y, z.\ N)$. The pattern $P = a, x, y, z$ captures the structure of left inversion on the positive proposition $\exists a.\ \mathsf{p}\ a \bullet {\text{¡}}A^- \bullet {\downarrow}B^-$.

The grammar of patterns is straightforward. Inversion on positive propositions can only have the effect of introducing new bindings (either LF variables a or SLS variables x) or handling a unification $a \doteq_p t$, which by our discussion above can always be resolved by the most general unifier $[t/a]$, so the pattern associated with a proposition $a \doteq_p t$ is t/a.

$$P ::= () \mid x, P \mid a, P \mid t/a, P$$

For sequences with one or more elements, we omit the trailing comma and $()$, writing $x, \ldots, z$ instead of $x, \ldots, z, ()$.

SLS patterns have a list-like structure (the comma is right associative) because they capture the sequential structure of proofs. The associated decomposition judgment $P :: (\Psi; \Delta)_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}$ takes two process states. It operates a bit like the spine typing judgment from Figure 4.3 in that the process state $(\Psi; \Delta)_\sigma$ (and the pattern P) are treated as an input and the process state $(\Psi'; \Delta')_{\sigma'}$ is treated as an output. The typing rules for SLS patterns are given in

$\boxed{P :: (\Psi; \Delta)_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}$ – presumes $\vdash_{\Sigma,\mathcal{R}} (\Psi; \Delta)_\sigma$ state

$$\frac{\Psi \vdash_{\Sigma,\mathcal{R}} \Delta \,\mathsf{stable}}{() :: (\Psi; \Delta)_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi; \Delta)_\sigma}\ ()$$

$$\frac{P :: (\Psi; \Theta\{z{:}\langle p^+_{lvl}\rangle\ lvl\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}{z, P :: (\Psi; \Theta\{\!\{p^+_{lvl}\}\!\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}\ \eta^+ \qquad \frac{P :: (\Psi; \Theta\{x{:}A^-\ ord\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}{x, P :: (\Psi; \Theta\{\!\{\downarrow A^-\}\!\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}\ \downarrow_L$$

$$\frac{P :: (\Psi; \Theta\{x{:}A^-\ eph\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}{x, P :: (\Psi; \Theta\{\!\{¡A^-\}\!\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}\ ¡_L \qquad \frac{P :: (\Psi; \Theta\{x{:}A^-\ pers\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}{x, P :: (\Psi; \Theta\{\!\{!A^-\}\!\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}\ !_L$$

$$\frac{P :: (\Psi; \Theta\{\cdot\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}{P :: (\Psi; \Theta\{\!\{\mathbf{1}\}\!\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}\ \mathbf{1}_L \qquad \frac{P :: (\Psi; \Theta\{A^+, B^+\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}{P :: (\Psi; \Theta\{\!\{A^+ \bullet B^+\}\!\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}\ \bullet_L$$

$$\frac{P :: (\Psi, a{:}\tau; \Theta\{A^+\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}{a, P :: (\Psi; \Theta\{\!\{\exists a{:}\tau.A^+\}\!\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}\ \exists_L$$

$$\frac{P :: (\Psi, [t/a]\Psi'; [t/a]\Theta\{\cdot\})_{[t/a]\sigma} \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}{t/a, P :: (\Psi, a{:}\tau, \Psi'; \Theta\{\!\{a \doteq_\tau t\}\!\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}}\ \doteq_L$$

Figure 4.7: SLS patterns

Figure 4.7. We preserve the side conditions from the previous chapter: when we frame off a inverting positive proposition in the process state, it is required to be the left-most one. As in focused OL_3, this justifies our omission of the variables associated with positive propositions: the positive proposition we frame off is always uniquely identified not by its associated variable but by its position in the context.

Note that there no longer appears to be a one-to-one correspondence between proof terms and rules: $\downarrow_L$, $¡_L$, and $!_L$ appear to have the same proof term, and $\mathbf{1}_L$ and $\bullet_L$ appear to have no proof term at all. To view patterns as being intrinsically typed – that is, to view them as actual representatives of (incomplete) derivations – we must think of patterns as carrying extra annotations that allow them to continue matching the structure of proof rules.

4.2.5 Values, terms, and spines

Notably missing from the SLS types are the upshifts $\uparrow A^+$ and right-permeable negative atomic propositions p^-_{lax}. The removal of these two propositions effectively means that the succedent of a stable SLS sequent can only be $\langle p^-\rangle\ true$ or $A^+\ lax$. The SLS framework only considers *complete* proofs of judgments $\langle p^-\rangle\ true$, whereas traces, associated with proofs of $A^+\ lax$ and introduced below in Section 4.2.6, are a proof term assignment for partial proofs. Excepting the proof term $\{\mathsf{let}\, T\, \mathsf{in}\, V\}$, which we present as part of the *concurrent* fragment of SLS in Section 4.2.6 below, the values, terms, and spines that stand for complete proofs will be referred

$\boxed{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} V : [A^+]}$ – presumes $\Psi \vdash_{\Sigma,\mathcal{R}} \Delta$ stable, and $\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} A^+$ prop$^+$

$$\frac{\Delta \textit{ matches } z{:}\langle A^+\rangle}{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} z : [A^+]}\ id^+$$

$$\frac{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} N : A^-}{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} N : [\downarrow A^-]}\ \downarrow_R \qquad \frac{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} N : A^-}{\Psi; \Delta\restriction_{eph} \vdash_{\Sigma,\mathcal{R}} \text{¡}N : [\text{¡}A^-]}\ \text{¡}_R \qquad \frac{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} N : A^-}{\Psi; \Delta\restriction_{pers} \vdash_{\Sigma,\mathcal{R}} !N : [!A^-]}\ !_R$$

$$\frac{\Delta \textit{ matches } \cdot}{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} () : [\mathbf{1}]}\ \mathbf{1}_R \qquad \frac{\Psi; \Delta_1 \vdash_{\Sigma,\mathcal{R}} V_1 : [A_1^+] \quad \Psi; \Delta_2 \vdash_{\Sigma,\mathcal{R}} V_2 : [A_2^+]}{\Psi; \Delta_1, \Delta_2 \vdash_{\Sigma,\mathcal{R}} V_1 \bullet V_2 : [A_1^+ \bullet A_2^+]}\ \bullet_R$$

$$\frac{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} t : \tau \quad \Psi; \Delta \vdash_{\Sigma,\mathcal{R}} V : [[t/a]A^+]}{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} t, V : [\exists a{:}\tau.A^+]}\ \exists_R \qquad \frac{\Delta \textit{ matches } \cdot}{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} \text{REFL} : t \doteq_\tau t}\ \doteq_R$$

Figure 4.8: SLS values

$\boxed{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} R : U}$ – presumes $\Psi \vdash_{\Sigma,\mathcal{R}} \Delta$ stable and $U = \langle C^-\rangle\ ord$

$$\frac{\Psi; \Theta\{[A^-]\} \vdash_{\Sigma,\mathcal{R}} Sp : U}{\Psi; \Theta\{\!\{x{:}A^-\}\!\} \vdash_{\Sigma,\mathcal{R}} x \cdot Sp : U}\ focus_L \qquad \frac{\mathsf{r} : A^- \in \Sigma \quad \Psi; \Theta\{[A^-]\} \vdash_{\Sigma,\mathcal{R}} Sp : U}{\Psi; \Theta\{\!\{\cdot\}\!\} \vdash_{\Sigma,\mathcal{R}} \mathsf{r} \cdot Sp : U}\ rule$$

Figure 4.9: SLS atomic terms

$\boxed{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} N : A^-\ ord}$ – presumes $\Psi \vdash_{\Sigma,\mathcal{R}} \Delta$ stable and $\Psi; \mathcal{C} \vdash_{\Sigma,\mathcal{R}} A^-$ prop$^-$

$$\frac{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} R : \langle p^-\rangle\ ord}{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} R : p^-\ ord}\ \eta^-$$

$$\frac{P :: (\Psi; A^+, \Delta)_{\mathsf{id}_\Psi} \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_\sigma \quad \Psi'; \Delta' \vdash_{\Sigma,\mathcal{R}} N : \sigma B^-\ ord}{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} \lambda P.N : A^+ \rightarrowtail B^-\ ord}\ \rightarrowtail_R$$

$$\frac{P :: (\Psi; \Delta, A^+)_{\mathsf{id}_\Psi} \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_\sigma \quad \Psi'; \Delta' \vdash_{\Sigma,\mathcal{R}} N : \sigma B^-\ ord}{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} \lambda P.N : A^+ \twoheadrightarrow B^-\ ord}\ \twoheadrightarrow_R$$

$$\frac{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} N_1 : A_1^-\ ord \quad \Psi; \Delta \vdash_{\Sigma,\mathcal{R}} N_2 : A_2^-\ ord}{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} N_1 \,\&\, N_2 : A_1^- \,\&\, A_2^-\ ord}\ \&_R$$

$$\frac{\Psi, a{:}\tau; \Delta \vdash_{\Sigma,\mathcal{R}} N : A^-\ ord}{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} \lambda a.N : \forall a{:}\tau.A^-\ ord}\ \forall_R$$

$$\frac{T :: (\Psi; \Delta)_{\mathsf{id}_\Psi} \rightsquigarrow^*_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'} \quad \Psi'; \Delta' \vdash_{\Sigma,\mathcal{R}} V : [\sigma A^+]}{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} \{\mathsf{let}\, T\, \mathsf{in}\, V\} : \bigcirc A^+\ ord}\ \bigcirc_R$$

Figure 4.10: SLS terms

$\boxed{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} Sp : U}$ – presumes $\Psi \vdash_{\Sigma,\mathcal{R}} \Delta \,\mathsf{infoc}$

$$\frac{\Delta \; \mathit{matches} \; [A^-]}{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} \textsc{nil} : \langle A^- \rangle \; \mathit{ord}} \; id^-$$

$$\frac{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} V : [A^+] \quad \Psi; \Theta\{[B^-]\} \vdash_{\Sigma,\mathcal{R}} Sp : U}{\Psi; \Theta\{\!\{\Delta, [A^+ \rightarrowtail B^-]\}\!\} \vdash_{\Sigma,\mathcal{R}} V; Sp : U} \rightarrowtail_L$$

$$\frac{\Psi; \Delta \vdash_{\Sigma,\mathcal{R}} V : [A^+] \quad \Psi; \Theta\{[B^-]\} \vdash_{\Sigma,\mathcal{R}} Sp : U}{\Psi; \Theta\{\!\{[A^+ \twoheadrightarrow B^-], \Delta\}\!\} \vdash_{\Sigma,\mathcal{R}} V; Sp : U} \twoheadrightarrow_L$$

$$\frac{\Psi; \Theta\{[A_1^-]\} \vdash_{\Sigma,\mathcal{R}} Sp : U}{\Psi; \Theta\{\!\{[A_1^- \,\&\, A_2^-]\}\!\} \vdash_{\Sigma,\mathcal{R}} \pi_1; Sp : U} \,\&_{L1} \qquad \frac{\Psi; \Theta\{[A_2^-]\} \vdash_{\Sigma,\mathcal{R}} Sp : U}{\Psi; \Theta\{\!\{[A_1^- \,\&\, A_2^-]\}\!\} \vdash_{\Sigma,\mathcal{R}} \pi_2; Sp : U} \,\&_{L2}$$

$$\frac{\Psi \vdash_{\Sigma,\mathcal{R}} t : \tau \quad [t/a]B^- = B'^- \quad \Psi; \Theta\{[B'^-]\} \vdash_{\Sigma,\mathcal{R}} Sp : U}{\Psi; \Theta\{\!\{[\forall a{:}\tau.B^-]\}\!\} \vdash_{\Sigma,\mathcal{R}} t; Sp : U} \,\forall_L$$

Figure 4.11: SLS spines

to as the *deductive fragment* of SLS.

SLS values (Figure 4.8)	$V ::= z \mid N \mid ¡N \mid !N \mid () \mid V_1 \bullet V_2 \mid t, V \mid \textsc{refl}$
SLS atomic terms (Figure 4.9)	$R ::= x \cdot Sp \mid \mathsf{r} \cdot Sp$
SLS terms (Figure 4.10)	$N ::= R \mid \lambda P.N \mid N_1 \,\&\, N_2 \mid \lambda a.N \mid \{\mathsf{let}\, T \,\mathsf{in}\, V\}$
SLS spines (Figure 4.11)	$Sp ::= \textsc{nil} \mid V; Sp \mid \pi_1; Sp \mid \pi_2; Sp \mid t; Sp$

In contrast to OL_3, we distinguish the syntactic category R of atomic terms that correspond to stable sequents. As with patterns, we appear to conflate the proof terms associated with different proof rules – we have a single $\lambda P.N$ constructor and a single $V; Sp$ spine rather than one term $\lambda^> N$ and spine $V^> Sp$ associated with propositions $A^+ \twoheadrightarrow B^-$ and another term $\lambda^< N$ and spine $V^< Sp$ associated with propositions $A^+ \rightarrowtail B^-$. As with patterns, it is possible to think of these terms as just having extra annotations ($\lambda^>$ or $\lambda^<$) that we have omitted. Without these annotations, proof terms carry less information than derivations, and the rules for values, terms, and spines in Figures 4.8–4.11 must be seen as typing rules. With these extra implicit annotations (or, possibly, with some of the technology of bidirectional typechecking), values, terms, and spines can continue to be seen as representatives of derivations.

Aside from $\bigcirc_R$ and its associated term $\{\mathsf{let}\, T \,\mathsf{in}\, V\}$, which belongs to the concurrent fragment of SLS, there is one rule in Figures 4.8–4.11 that does not have an exact analogues as a rule in OL_3, the rule labeled *rule* in Figure 4.9. This rule corresponds to an atomic term $\mathsf{r} \cdot Sp$ and accounts for the fact that there is an additional source of persistent facts in SLS, the signature Σ, that is not present in OL_3. To preserve the bijective correspondence between OL_3 and SLS proof terms, we need to place every rule $\mathsf{r} : A^-$ in the SLS signature Σ into the corresponding OL_3 context as a persistent proposition.

As with LF terms, we will use a shorthand for atomic terms $x \cdot Sp$ and $\mathsf{r} \cdot Sp$, writing $(\mathsf{foo}\, t\, s\, V\, V')$ instead of $\mathsf{foo} \cdot (t; s; V; V'; \textsc{nil})$ when we are not concerned with the fact that

$\boxed{S :: (\Psi;\Delta)_\sigma \rightsquigarrow_{\Sigma,\mathcal{R}} (\Psi';\Delta')_{\sigma'}}$ – presumes $\vdash_{\Sigma,\mathcal{R}} (\Psi;\Delta)_\sigma$ state and $\Psi \vdash_{\Sigma,\mathcal{R}} \Delta$ stable

$$\frac{\Psi;\Delta \vdash_{\Sigma,\mathcal{R}} R : \langle\bigcirc B^+\rangle\ ord \quad P :: (\Psi,\Theta\{B^+\})_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi';\Delta')_{\sigma'}}{\{P\} \leftarrow R :: (\Psi;\Theta\{\!\{\Delta\}\!\})_\sigma \rightsquigarrow_{\Sigma,\mathcal{R}} (\Psi';\Delta')_{\sigma'}}$$

Figure 4.12: SLS steps

$\boxed{T :: (\Psi;\Delta)_\sigma \rightsquigarrow^*_{\Sigma,\mathcal{R}} (\Psi';\Delta')_{\sigma'}}$ – presumes $\vdash_{\Sigma,\mathcal{R}} (\Psi;\Delta)_\sigma$ state and $\Psi \vdash_{\Sigma,\mathcal{R}} \Delta$ stable

$$\frac{}{\diamond :: (\Psi;\Delta)_\sigma \rightsquigarrow^*_{\Sigma,\mathcal{R}} (\Psi;\Delta)_\sigma} \qquad \frac{S :: (\Psi;\Delta)_\sigma \rightsquigarrow_{\Sigma,\mathcal{R}} (\Psi';\Delta')_{\sigma'}}{S :: (\Psi;\Delta)_\sigma \rightsquigarrow^*_{\Sigma,\mathcal{R}} (\Psi';\Delta')_{\sigma'}}$$

$$\frac{T :: (\Psi_1;\Delta_1)_{\sigma_1} \rightsquigarrow^*_{\Sigma,\mathcal{R}} (\Psi_2;\Delta_2)_{\sigma_2} \quad T' :: (\Psi_2;\Delta_2)_{\sigma_2} \rightsquigarrow^*_{\Sigma,\mathcal{R}} (\Psi_3;\Delta_3)_{\sigma_3}}{T;T' :: (\Psi_1;\Delta_1)_{\sigma_1} \rightsquigarrow^*_{\Sigma,\mathcal{R}} (\Psi_3;\Delta_3)_{\sigma_3}}$$

Figure 4.13: SLS traces

atomic terms consist of a variable and a spine.

4.2.6 Steps and traces

The deductive fragment of SLS presented in Figures 4.7–4.11 covers every SLS proposition except for the lax modality $\bigcirc A^+$. It is in the context of the lax modality that we will present proof terms corresponding to partial proofs; we call this fragment the *concurrent* fragment of SLS because of its relationship with concurrent equality, described in Section 4.3.

Steps	$S ::= \{P\} \leftarrow R$
Traces	$T ::= \diamond \mid T_1;T_2 \mid S$

A step $S = \{P\} \leftarrow x \cdot Sp$ corresponds precisely to the notion of a *synthetic inference rule* as discussed in Section 2.4. A step in SLS corresponds to a use of left focus, a use of the left rule for the lax modality, and a use of the admissible focal substitution lemma in OL_3:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Psi;\Theta'\{[A^-]\} \vdash \langle\bigcirc B^+\rangle\ true\end{array}}{\Psi;\Theta'\{x{:}A^-\ true\} \vdash \langle\bigcirc B^+\rangle\ ord}\ focus_L \quad \dfrac{\begin{array}{c}\Psi';\Delta' \vdash \sigma' A^+\ lax\\ \vdots\\ \Psi;\Theta\{B^+\} \vdash \sigma A^+\ lax\end{array}}{\Psi;\Theta\{[\bigcirc B^+]\} \vdash \sigma A^+\ lax}\ \bigcirc_L}{\Psi;\Theta\{\!\{\Theta'\{x{:}A^-\ ord\}\}\!\} \vdash \sigma A^+\ lax}\ subst^-$$

The spine Sp corresponds to the complete proof of $\Psi;\Theta'\{[A^-]\} \vdash \langle\bigcirc B^+\rangle\ ord$, and the pattern P corresponds to the partial proof from $\Psi';\Delta' \vdash \sigma' A^+\ lax$ to $\Psi;\Theta\{B^+\} \vdash \sigma A^+\ lax$. The typing rules for steps are given in Figure 4.12. Because we understand these synthetic inference rules as relations between process states, we call $(\Psi;\Delta) \rightsquigarrow_{\Sigma,\mathcal{R}} (\Psi';\Delta')$ a *synthetic transition*. Traces T

are monoids over steps – $\diamond$ is an empty trace, S is a trace consisting of a single step, and $T_1; T_2$ is the sequential composition of traces. The typing rules for traces in Figure 4.13 straightforwardly reflect this monoid structure. Both of the judgments $S :: (\Psi; \Delta)_\sigma \rightsquigarrow_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}$ and $T :: (\Psi; \Delta)_\sigma \rightsquigarrow^*_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}$ work like the rules for patterns, in that the step S or trace T is treated as an input along with the initial process state $(\Psi; \Delta)_\sigma$, whereas the final process state $(\Psi'; \Delta')_{\sigma'}$ is treated as an output.

Steps incorporate left focus and the left rule for $\bigcirc$, and *let-expressions* $\{\mathsf{let}\, T\, \mathsf{in}\, V\}$, which include traces in deductive terms, incorporate right focus and the right rule for the lax modality in OL_3:

$$\dfrac{\dfrac{\vdots}{\Psi'; \Delta' \vdash [\sigma' A^+]}}{\Psi'; \Delta' \vdash \sigma' A^+\ \mathit{lax}}\ \mathit{focus}_R$$

$$\dfrac{\dfrac{\vdots}{\Psi; \Delta \vdash A^+\ \mathit{lax}}}{\Psi; \Delta \vdash \bigcirc A^+}\ \bigcirc_R$$

The trace T represents the entirety of the partial proof from $\Psi; \Delta \vdash A^+\ \mathit{lax}$ to $\Psi'; \Delta' \vdash \sigma' A^+\ \mathit{lax}$ that proceeds by repeated use of steps or synthetic transitions, and the eventual conclusion V represents the complete proof of $\Psi'; \Delta' \vdash [\sigma' A^+]\ \mathit{lax}$ that follows the series of synthetic transitions.

Both of the endpoints of a trace are stable sequents, but it will occasionally be useful to talk about steps and traces that start from unstable sequents and immediately decompose positive propositions. We will use the usual trace notation $(\Psi; \Delta)_\sigma \rightsquigarrow^*_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}$ to describe the type of these partial proofs. The proof term associated with this type will be written as $\lambda P.T$, where $P :: (\Psi; \Delta)_\sigma \Longrightarrow_{\Sigma,\mathcal{R}} (\Psi''; \Delta'')_{\sigma''}$ and $T :: (\Psi''; \Delta'')_{\sigma''} \rightsquigarrow^*_{\Sigma,\mathcal{R}} (\Psi'; \Delta')_{\sigma'}$.

4.2.7 Presenting traces

To present traces in a readable way, we will use a notation that interleaves process states among the steps of a trace, a common practice in Hoare-style reasoning [Hoa71]. As an example, recall the series of transitions that our money-store-battery-robot system took in Section 2.3.9:

$6 (1) *battery-less robot (1)* *turn $6 into a battery (all you want)*	$\rightsquigarrow$	*battery (1)* *battery-less robot (1)* *turn $6 into a battery (all you want)*	$\rightsquigarrow$	*robot (1)* *turn $6 into a battery (all you want)*

This evolution can now be precisely captured as a trace in SLS:

$$(x{:}\langle \mathsf{6bucks}\rangle\ \mathit{eph},\ \ f{:}(\mathsf{battery} \rightarrowtail \bigcirc\mathsf{robot})\ \mathit{eph},\ \ g{:}(\mathsf{6bucks} \rightarrowtail \bigcirc\mathsf{battery})\ \mathit{pers})$$
$$\{y\} \leftarrow g\, x;$$
$$(y{:}\langle \mathsf{battery}\rangle\ \mathit{eph},\ \ f{:}(\mathsf{battery} \rightarrowtail \bigcirc\mathsf{robot})\ \mathit{eph},\ \ g{:}(\mathsf{6bucks} \rightarrowtail \bigcirc\mathsf{battery})\ \mathit{pers})$$
$$\{z\} \leftarrow f\, y$$
$$(z{:}\langle \mathsf{robot}\rangle\ \mathit{eph},\ \ g{:}(\mathsf{6bucks} \rightarrowtail \bigcirc\mathsf{battery})\ \mathit{pers})$$

4.2.8 Frame properties

The *frame rule* is a concept from separation logic [Rey02]. It states that if a property holds of some program, then the property holds under any extension of the mutable state. The frame rule increases the modularity of separation logic proofs, because two program fragments that reason about different parts of the state can be reasoned about independently.

Similar frame properties hold for SLS traces. The direct analogue of the frame rule is the observation that a trace can always have some extra state framed on to the outside. This is a generalization of weakening to SLS traces.

Theorem 4.2 (Frame weakening)**.**
If $T :: (\Psi; \Delta) \rightsquigarrow^*_{\Sigma,\mathcal{R}} (\Psi'; \Delta')$*, then* $T :: (\Psi, \Psi''; \Theta\{\Delta\}) \rightsquigarrow^*_{\Sigma,\mathcal{R}} (\Psi', \Psi''; \Theta\{\Delta\})$.

Proof. Induction on T and case analysis on the first steps of T, using admissible weakening and the properties of matching constructs at each step. □

The frame rule is a weakening property which ensures that new, irrelevant state can always be added to a state. Conversely, any state that is never accessed or modified by a trace can be always be removed without making the trace ill-typed. This property is a generalization of strengthening to SLS traces.

Theorem 4.3 (Frame strengthening)**.**
If $T :: (\Psi; \Theta\{x{:}Y\ lvl\}) \rightsquigarrow^*_{\Sigma,\mathcal{R}} (\Psi'; \Theta'\{x{:}Y\ lvl\})$ *and* x *is not free in any of the steps of* T*, then* $T :: (\Psi; \Theta\{\cdot\}) \rightsquigarrow^*_{\Sigma,\mathcal{R}} (\Psi'; \Theta\{\cdot\})$.

Proof. Induction on T and case analysis on the first steps of T, using a lemma to enforce that, if x is not free in an individual step, it is either not present in the context of the subderivation (if $lvl = ord$ or eph) or else it can be strengthened away (if $lvl = pers$). □

4.3 Concurrent equality

Concurrent equality is a notion of equivalence on traces that is coarser than the equivalence relation we would derive from partial OL_3 proofs. Consider the following SLS signature:

$$
\begin{aligned}
\Sigma = \cdot,\ & \mathsf{a} : \mathsf{prop\,lin},\ \mathsf{b} : \mathsf{prop\,lin},\ \mathsf{c} : \mathsf{prop\,lin},\ \mathsf{d} : \mathsf{prop\,lin},\ \mathsf{e} : \mathsf{prop\,lin},\ \mathsf{f} : \mathsf{prop\,lin},\\
& \mathsf{first} : \mathsf{a} \rightarrowtail \bigcirc(\mathsf{b} \bullet \mathsf{c}),\\
& \mathsf{left} : \mathsf{b} \rightarrowtail \bigcirc\mathsf{d},\\
& \mathsf{right} : \mathsf{c} \rightarrowtail \bigcirc\mathsf{e},\\
& \mathsf{last} : \mathsf{d} \bullet \mathsf{e} \rightarrowtail \bigcirc\mathsf{f}
\end{aligned}
$$

Under the signature Σ, we can create two traces with the type $x_a{:}\langle\mathsf{a}\rangle \rightsquigarrow^*_{\Sigma,\mathcal{R}} x_f{:}\langle\mathsf{f}\rangle$:

$$
\begin{aligned}
T_1 \ = \ & \{x_b, x_c\} \leftarrow \mathsf{first}\, x_a;\\
& \{x_d\} \leftarrow \mathsf{left}\, x_b;\\
& \{x_e\} \leftarrow \mathsf{right}\, x_c;\\
& \{x_f\} \leftarrow \mathsf{last}\,(x_d \bullet x_e)
\end{aligned}
\qquad \text{versus} \qquad
\begin{aligned}
T_2 \ = \ & \{x_b, x_c\} \leftarrow \mathsf{first}\, x_a;\\
& \{x_e\} \leftarrow \mathsf{right}\, x_c;\\
& \{x_d\} \leftarrow \mathsf{left}\, x_b;\\
& \{x_f\} \leftarrow \mathsf{last}\,(x_d \bullet x_e)
\end{aligned}
$$

In both cases, there is an $x_a{:}\langle \mathsf{a} \rangle$ resource that transitions to a resource $x_b{:}\langle \mathsf{b} \rangle$ and another resource $x_c{:}\langle \mathsf{c} \rangle$, and then $x_b{:}\langle \mathsf{b} \rangle$ transitions to $x_d{:}\langle \mathsf{d} \rangle$ while, independently, $x_c{:}\langle \mathsf{c} \rangle$ transitions to $x_d{:}\langle \mathsf{d} \rangle$. Then, finally, the $x_d{:}\langle \mathsf{d} \rangle$ and $x_e{:}\langle \mathsf{e} \rangle$ combine to transition to $x_f{:}\langle \mathsf{f} \rangle$, which completes the trace.

The independence here is key: if two steps consume different resources, then we want to treat them as independent concurrent steps that could have equivalently happened in the other order. However, if we define equivalence only in terms of the α-equivalence of partial OL_3 derivations, the two traces above are distinct. In this section, we introduce a coarser equivalence relation, *concurrent equality*, that allows us to treat traces that differ only in the interleaving of independent and concurrent steps as being equal. The previous section considered the proof terms of SLS as a fragment of OL_3 that is better able to talk about partial proofs. The introduction of concurrent equality takes a step beyond OL_3, because it breaks the bijective correspondence between OL_3 proofs and SLS proofs. As the example above indicates, there are simply more OL_3 proofs than SLS proofs when we quotient the latter modulo concurrent equality and declare T_1 and T_2 to be (concurrently) equal.

Concurrent equality was first was introduced and explored in the context of CLF [WCPW02], but our presentation follows the reformulation in [CPS+12], which defines concurrent equivalence based on an analysis of the variables that are used (inputs) and introduced (outputs) by a given step. Specifically, our strategy will be to take a particular well-typed trace T and define a set I of pairs of states (S_1, S_2) with the property that, if $S_1; S_2$ is a well-typed trace, then $S_2; S_1$ is a concurrently equivalent and well-typed trace. This *independency relation* allows us to treat the trace T as a *trace monoid*. Concurrent equality, in turn, is just α-equality of SLS proof terms combined with treating $\{\mathsf{let}\, T \,\mathsf{in}\, V\}$ and $\{\mathsf{let}\, T' \,\mathsf{in}\, V\}$ as equivalent if T and T' are equivalent according to the equivalence relation imposed by treating T and T' as trace monoids.

This formulation of concurrent equality facilitates applying the rich theory developed around trace monoids to SLS traces. For example, it is decidable whether two traces T and T' are equivalent as trace monoids, and there are algorithms for determining whether T' is a subtrace of T (that is, whether there exist T_{pre} and T_{post} such that T is equivalent $T_{pre}; T'; T_{post}$) [Die90]. A different sort of matching problem, in which we are given T, T_{pre}, and T_{post} and must determine whether there exists a T' such that T is equivalent $T_{pre}; T'; T_{post}$, was considered in [CPS+12].

Unfortunately, the presence of equality in SLS complicates our treatment of independency. The *interface* of a step is used to define independency on steps $S = (\{P\} \leftarrow R)$. Two components of the interface, the input variables ${}^{\bullet}S$ and the output variables $S^{\bullet}$ are standard in the literature on Petri nets – see, for example, [Mur89, p. 553]. The third component, unified variables ${}^{\circledast}S$, is unique to our presentation.

Definition 4.4 (Interface of a step)**.**

- *The* input variables *of a step, denoted* ${}^{\bullet}S$*, are all the LF variables* a *and SLS variables* x *free in the normal term* R*.*
- *The* output variables *of a step* $S = (\{P\} \leftarrow R)$*, denoted by* $S^{\bullet}$*, are all the LF variables* a *and SLS variables* x *bound by the pattern* P *that are not subsequently consumed by a substitution* t/a *in the same pattern.*
- *The* unified variables *of a step, denoted by* ${}^{\circledast}S$*, are the free variables of a step that are modified by unification. If* t/a *appears in a pattern and* a *is free in the pattern, then* $t = b$

for some other variable b; both a and b (if the latter is free in the pattern) are included in the step's unified variables.

Consider a well-typed trace $S_1; S_2$ with two steps. It is possible, by renaming variables bound in patterns, to ensure that $\emptyset = {}^\bullet S_1 \cap S_2{}^\bullet = {}^\bullet S_1 \cap S_1{}^\bullet = {}^\bullet S_2 \cap S_2{}^\bullet$. We will generally assume that, in the traces we consider, the variables introduced in each step's pattern are renamed to be distinct from the input or output variables of all previous steps.

If $S_1; S_2$ is a well-typed trace, then the order of S_1 and S_2 is fixed if S_1 introduces variables that are used by S_2 – that is, if $\emptyset \neq S_1{}^\bullet \cap {}^\bullet S_2$. For example, if $S_1 = (\{x_b, x_c\} \leftarrow \mathsf{first}\, x_a)$ and $S_2 = (\{x_d\} \leftarrow \mathsf{left}\, x_b)$, then $\{x_b\} = S_1{}^\bullet \cap {}^\bullet S_2$, and the two steps cannot be reordered relative to one another. Conversely, the condition that $\emptyset = S_1{}^\bullet \cap {}^\bullet S_2$ is sufficient to allow reordering in a CLF-like framework [CPS+12], and is also sufficient to allow reordering in SLS when neither step contains unified variables (that is, when $\emptyset = {}^\circledast S_1 = {}^\circledast S_2$). The unification driven by equality, however, can have subtle effects. Consider the following two-step trace:

$$
\begin{array}{ll}
 & (a{:}p, b{:}p;\ \ x{:}(\bigcirc(b \doteq_p a))\ eph,\ \ y{:}(\mathsf{foo}\, b \rightarrowtail \bigcirc(\mathsf{bar}\, a))\ eph,\ \ z{:}\langle \mathsf{foo}\, a\rangle\ eph) \\
\{b/a\} \leftarrow x; & \\
 & (b{:}p;\ \ y{:}(\mathsf{foo}\, b \rightarrowtail \bigcirc(\mathsf{bar}\, b))\ eph,\ \ z{:}\langle \mathsf{foo}\, b\rangle\ eph) \\
\{w\} \leftarrow y\, z & \\
 & (b{:}p;\ \ w{:}\langle \mathsf{bar}\, b\rangle\ eph)
\end{array}
$$

This trace cannot be reordered even though $\emptyset = \emptyset \cap \{y, z\} = (\{b/a\} \leftarrow x)^\bullet \cap {}^\bullet(\{w\} \leftarrow y\, z)$, because the atomic term $y\, z$ is only well typed after the LF variables a and b are unified. It is not even sufficient to compare the free and unified variables (requiring that $\emptyset = {}^\circledast S_1 \cap {}^\bullet S_2$), as in the example above ${}^\circledast(\{b/a\} \leftarrow x) = \{a, b\}$ and ${}^\bullet(\{w\} \leftarrow y\, z) = \{y, z\}$ – and obviously $\emptyset = \{a, b\} \cap \{y, z\}$.

The simplest solution is to forbid steps with unified variables from being reordered at all: we can say that $(S_1, S_2) \in I$ if $\emptyset = S_1{}^\bullet \cap {}^\bullet S_2 = {}^\circledast S_1 = {}^\circledast S_2$. It is unlikely that this condition is satisfying in general, but it is sufficient for all the examples in this dissertation. Therefore, we will define concurrent equality on the basis of this simple solution. Nevertheless, three other possibilities are worth considering; all three are equivalent to this simple solution as far as the examples given here are concerned.

Restricting open propositions Part of the problem with the example above was that there were variables free in the *type* of a transition that were not free in the *term*. A solution is to restrict propositions so that negative propositions in the context are always *closed* relative to the LF context (or at least relative to the part of the LF context that mentions types subject to pure variable equality, which would be simple enough to determine with a subordination-based analysis). This restriction means that a step $S = \{P\} \leftarrow R$ can only have the parameter a free in R's type if a is free in R, allowing us to declare that S_1 and S_2 are reorderable – meaning (S_1, S_2) and (S_2, S_1) are in the independency relation I – whenever $\emptyset = S_1{}^\bullet \cap {}^\bullet S_2 = {}^\circledast S_1 \cap {}^\bullet S_2 = {}^\bullet S_1 \cap {}^\circledast S_2$.

While this restriction would be sufficient for the examples in this dissertation, it would preclude a conjectured extension of the destination-adding transformation given in Chapter 7 to nested specifications (nested versus flat specifications are discussed in Section 5.1).

Re-typing Another alternative would be to follow CLF and allow any reordering permitted by the input and output interfaces, but then forbid those that cannot be re-typed. (This was necessary in CLF to deal with the presence of $\top$.) This is very undesirable, however, because it leads to strange asymmetries. The following trace would be reorderable by this definition, for example, but in a symmetric case where the equality was $b \doteq_p a$ instead of $a \doteq_p b$, that would no longer be the case.

$$
\begin{array}{ll}
 & (a{:}p, b{:}p;\ \ x{:}(\bigcirc(a \doteq_p b))\ eph,\ \ y{:}(\forall w.\, \mathsf{foo}\, w \rightarrowtail \bigcirc(\mathsf{bar}\, w))\ eph,\ \ z{:}\langle \mathsf{foo}\, b\rangle\ eph) \\
\{w\} \leftarrow y\, b\, z; & \\
 & (a{:}p, b{:}p;\ \ x{:}(\bigcirc(a \doteq_p b))\ eph,\ \ w{:}\langle \mathsf{bar}\, a\rangle\ eph) \\
\{b/a\} \leftarrow x & \\
 & (b{:}p;\ \ w{:}\langle \mathsf{bar}\, b\rangle\ eph)
\end{array}
$$

Process states with equality constraints A third possible solution is be to change the way we handle the interaction of process states and unification. In this formulation of SLS, the process state $(\Psi; \Delta)_\sigma$ uses σ to capture the constraints that have been introduced by equality. As an alternative, we could have process states mention an explicit constraint store of equality propositions that have been encountered, as in Jagadeesan et al.'s formulation of concurrent constraint programming [JNS05]. Process states with equality constraints might facilitate talking explicitly about the interaction of equality and typing, which in our current formulation is left rather implicit.

4.3.1 Multifocusing

Concurrent equality is related to the equivalence relation induced by *multifocusing* [CMS09]. Like concurrent equality, multifocusing imposes a coarser equivalence relation on focused proofs. The coarser equivalence relation is enabled by a somewhat different mechanism: we are allowed to begin focus on multiple propositions simultaneously.

Both multifocusing and concurrent equality seek to address the sequential structure of focused proofs. The sequential structure of a computation needs to be addressed somehow, because it obscures the fact that the interaction between resources in a focused proof has the structure of a directed acyclic graph (DAG), not a sequence. We sketch a radically different, vaguely Feynman-diagram-inspired, way of presenting traces in Figure 4.14. Resources are the edges in the DAG and steps or synthetic inference rules are the vertexes. (The crossed edges that exchange x_2 and x_3 are only well-formed because, in our example trace, e and d were both declared to be ephemeral propositions.) Multifocusing gives a unique normal form to proofs by gathering all the focusing steps that can be rotated all the way to the beginning, then all the focusing steps that can happen as soon as those first steps have been rotated all the way to the beginning, etc.

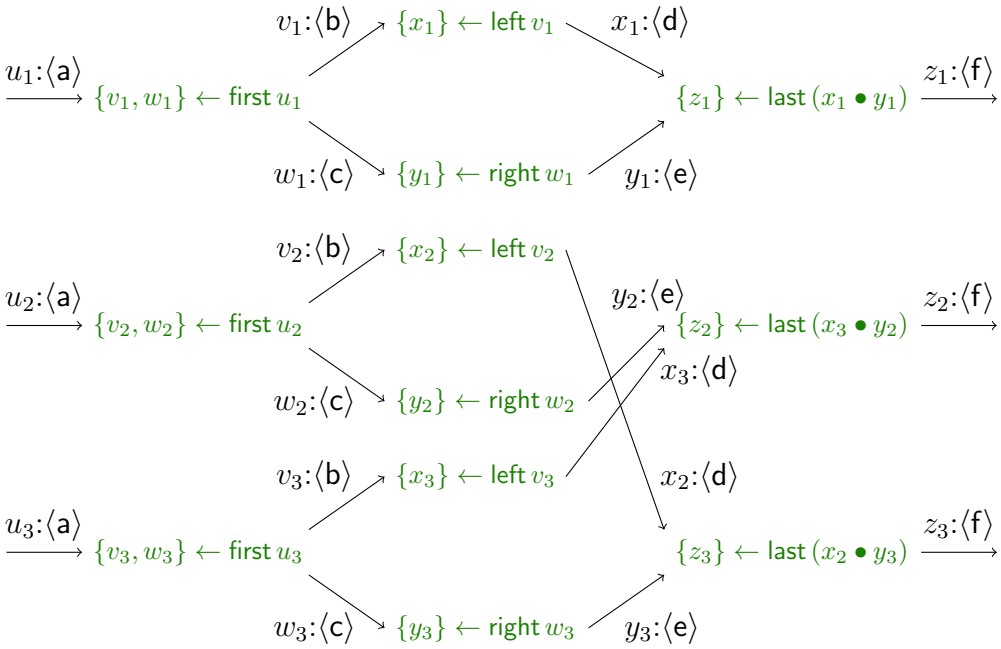


Figure 4.14: Interaction diagram for a trace $(u_1{:}\langle\mathsf{a}\rangle, u_2{:}\langle\mathsf{a}\rangle, u_3{:}\langle\mathsf{a}\rangle) \rightsquigarrow^*_\Sigma (z_1{:}\langle\mathsf{f}\rangle, z_2{:}\langle\mathsf{f}\rangle, z_3{:}\langle\mathsf{f}\rangle)$

In SLS, by contrast, we are content to represent the DAG structure as a list combined with the equivalence relation given by concurrent equality.

Multifocusing has only been explored carefully in the context of classical linear logic. We conjecture that derivations in OL_3 with a suitably-defined notion of multifocusing would be in bijective correspondence with SLS terms modulo concurrent equivalence, at least if we omit equality. Of course, without a formal notion of multifocusing for intuitionistic logic, this conjecture is impossible to state explicitly. The analogy with multifocusing may be able shed light on our difficulties in integrating concurrent equality and unification of pure variable types, because multifocusing has an independent notion of correctness: the equivalence relation given by multifocusing coincides with the the least equivalence relation that includes all permutations of independent rules in an unfocused sequent calculus proof [CMS09].

4.4 Adequate encoding

In Section 4.1.4 we discussed encoding untyped λ-calculus terms as LF terms of type exp, captured by the invertible function $\ulcorner e \urcorner$. Adequacy was extended to Linear LF (LLF) by Cervesato and Pfenning [CP02] and was extended to Ordered LF (OLF) by Polakow [Pol01]. The deductive fragment of SLS approximately extends both LLF and OLF, and the adequacy arguments made by Cervesato and Polakow extend straightforwardly to the deductive fragment of SLS. These adequacy arguments do not extend to the systems we want to encode in the concurrent fragment of SLS, however. The more general techniques we consider in this section will be explored further in Chapter 9 as a general technique for capturing invariants of SLS specifications.

The example that we will give to illustrate adequate encoding is the following signature, the SLS encoding of the push-down automata for parentheses matching from the introduction; we replace the atomic proposition $<$ with L and the proposition $>$ with R:

$$\begin{aligned}\Sigma_{PDA} = \cdot,\ & \mathsf{L} : \mathsf{prop\,ord},\\ & \mathsf{R} : \mathsf{prop\,ord},\\ & \mathsf{hd} : \mathsf{prop\,ord},\\ & \mathsf{push} : \mathsf{hd} \bullet \mathsf{L} \rightarrowtail \bigcirc(\mathsf{L} \bullet \mathsf{hd}),\\ & \mathsf{pop} : \mathsf{L} \bullet \mathsf{hd} \bullet \mathsf{R} \rightarrowtail \bigcirc(\mathsf{hd})\end{aligned}$$

We will relate this specification to a push-down automata defined in terms of stacks k and strings s, which we define inductively:

$$\begin{aligned}k &::= \cdot \mid k<\\ s &::= \cdot \mid <s \mid >s\end{aligned}$$

The transition system defined in terms of the stacks and strings has two transitions:

$$\begin{aligned}(k \rhd (<s)) &\mapsto ((k<) \rhd s)\\ ((k<) \rhd (>s)) &\mapsto (k \rhd s)\end{aligned}$$

Existing adequacy arguments for CLF specifications by Cervesato et al. [CPWW02] and by Schack-Nielsen [SN07] have a three-part structure structure. The first step is to define an encoding function $\ulcorner k \rhd s \urcorner = \Delta$ from PDA states $k \rhd s$ to process states Δ, so that, for example, the PDA state $(\cdot<<) \rhd (>>><\cdot)$ is encoded as the process state

$$x_2{:}\langle\mathsf{L}\rangle\ ord,\ \ x_1{:}\langle\mathsf{L}\rangle\ ord,\ \ h{:}\langle\mathsf{hd}\rangle\ ord,\ \ y_1{:}\langle\mathsf{R}\rangle\ ord,\ \ y_2{:}\langle\mathsf{R}\rangle\ ord,\ \ y_3{:}\langle\mathsf{R}\rangle\ ord,\ \ y_4{:}\langle\mathsf{L}\rangle\ ord$$

The second step is to prove a preservation-like property: if $\ulcorner k \rhd s \urcorner \rightsquigarrow_{\Sigma_{PDA}} \Delta'$, then $\Delta' = \ulcorner k' \rhd s' \urcorner$ for some k' and s'. The third step is the main adequacy result: that $\ulcorner k \rhd s \urcorner \rightsquigarrow_{\Sigma_{PDA}} \ulcorner k' \rhd s' \urcorner$ if and only if $k \rhd s \mapsto k' \rhd s'$.

The second step is crucial in general: without it, we might transition in SLS from the encoding of some $k \rhd s$ to a state Δ' that is not in the image of encoding. We will take the opportunity to re-factor Cervesato et al.'s approach, replacing the second step with a general statement about transitions in Σ_{PDA} preserving a well-formedness invariant. The invariant we discuss is a simple instance of the well-formedness invariants that we will explore further in Chapter 9.

The first step in our revised methodology is to describe a generative signature Σ_{Gen} that precisely captures the set of process states that encode machine states (Theorem 4.6 below). The second step is showing that the generative signature Σ_{Gen} describes an invariant of the signature Σ_{PDA} (Theorem 4.7). The third step, showing that $\ulcorner k \rhd s \urcorner \rightsquigarrow_{\Sigma_{PDA}} \ulcorner k' \rhd s' \urcorner$ if and only if $k \rhd s \mapsto k' \rhd s'$, is straightforward and follows other developments.

4.4.1 Generative signatures

A critical aspect of any adequacy argument is an understanding of the structure of the relevant context(s) (the LF context in LF encodings, the substructural context in CLF encodings, both in

$$
\begin{array}{lll}
\Sigma_{Gen} = \cdot, & \mathsf{L} : \mathsf{prop\,ord}, & \\
& \mathsf{R} : \mathsf{prop\,ord}, & \\
& \mathsf{hd} : \mathsf{prop\,ord}, & \\
& \mathsf{gen} : \mathsf{prop\,ord}, & \\
& \mathsf{gen\_stack} : \mathsf{prop\,ord}, & \\
& \mathsf{gen\_string} : \mathsf{prop\,ord}, & \\
& \mathsf{state} : \mathsf{gen} \rightarrowtail \bigcirc(\mathsf{gen\_stack} \bullet \mathsf{hd} \bullet \mathsf{gen\_string}) & G \rightarrow G_k\, \mathsf{hd}\, G_s \\
& \mathsf{stack/left} : \mathsf{gen\_stack} \rightarrowtail \bigcirc(\mathsf{L} \bullet \mathsf{gen\_stack}) & G_k \rightarrow < G_k \\
& \mathsf{stack/done} : \mathsf{gen\_stack} \rightarrowtail \bigcirc(\mathbf{1}) & G_k \rightarrow \epsilon \\
& \mathsf{string/left} : \mathsf{gen\_string} \rightarrowtail \bigcirc(\mathsf{gen\_string} \bullet \mathsf{L}) & G_s \rightarrow G_s < \\
& \mathsf{string/right} : \mathsf{gen\_string} \rightarrowtail \bigcirc(\mathsf{gen\_string} \bullet \mathsf{R}) & G_s \rightarrow G_s > \\
& \mathsf{string/done} : \mathsf{gen\_string} \rightarrowtail \bigcirc(\mathbf{1}) & G_s \rightarrow \epsilon
\end{array}
$$

Figure 4.15: Generative signature for PDA states and an analogous context-free grammar

SLS encodings). In the statement of adequacy for untyped λ-calculus terms (Section 4.1.4), for instance, it was necessary to require that the LF context Ψ take the form a_1:exp, ..., a_n:exp. In the adequacy theorems that have been presented for deductive logical frameworks, the structure of the context is quite simple. We can describe a set of building blocks that build small pieces of the context, and then define the set of valid process states (the *world*) any process state that can be built from a particular set of building blocks.

The structure of our PDA is too complex to represent as an arbitrary collection of building blocks. The process state is organized into three distinct zones:

[the stack] [the head] [the string being read]

We can't freely generate this structure with building blocks, but we can generate it with a context-free grammar. Conveniently, context-free grammars can be characterized within the machinery of SLS itself by describing *generative signatures* that can generate a set of process states we are interested in from a single seed context. The signature Σ_{Gen} in Figure 4.15 treats all the atomic propositions of Σ_{PDA} – the atomic propositions L, R and hd – as *terminals*, and introduces three *nonterminals* gen, gen_stack, and gen_string.

An informal translation of the signature Σ_{Gen} as a context-free grammar is given on the right-hand side of Figure 4.15. Observe that the sentences in the language G encode the states of our PDA as a string. We will talk much more about generative signatures in Chapter 9.

4.4.2 Restriction

The operation of *restriction* adapts the concept of "terminal" and "non-terminal" to SLS. Note that process states Δ such that $(x{:}\langle\mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen}} \Delta$ are only well-formed under the signature

Σ_{PDA} if they are free of nonterminals; we can define an operation of *restriction* that filters out the non-terminal process states by checking whether they are well-formed in a signature that only declares the terminals.

Definition 4.5 (Restriction)**.**

* *$\Psi\lightning_\Sigma$ is a total function that returns the largest context $\Psi' \subseteq \Psi$ such that $\vdash_\Sigma \Psi\ \mathsf{ctx}$ (defined in Figure 4.3) by removing all the LF variables in Ψ whose types are not well-formed in the context Σ.*
* *$(\Psi;\Delta)\lightning_\Sigma$ is a partial function that is defined exactly when, for every variable declaration $x{:}T\ ord$ or $x{:}T\ eph$ in Δ, we have that $(\Psi\lightning_\Sigma) \vdash_\Sigma T\ \mathsf{left}$ (defined in Figure 4.6). When it is defined, $(\Psi;\Delta)\lightning_\Sigma = ((\Psi\lightning_\Sigma);\Delta')$, where Δ' is Δ except for the variable declarations $x{:}T\ pers$ in Δ for which it was not the case that $(\Psi\lightning_\Sigma) \vdash_\Sigma T\ \mathsf{left}$.*
* *We will also use $(\Psi;\Delta)\lightning_\Sigma$ as a judgment which expresses that the function is defined.*

Because restriction is only defined if all the ordered and linear propositions in Δ are well-formed in Σ; this means that $(x{:}\langle\mathsf{gen}\rangle\ ord)\lightning_{\Sigma_{PDA}}$ is not defined. Restriction acts as a semi-permeable membrane on process states: some process states cannot pass through at all, and others pass through with some of their LF variables and persistent propositions removed. We can represent context restriction $(\Psi;\Delta)\lightning_\Sigma = (\Psi';\Delta')$ in a two-dimensional notation as a dashed line annotated with the restricting signature:

$$\begin{array}{c} (\Psi;\Delta) \\ \Sigma\ \text{////////} \\ (\Psi';\Delta') \end{array}$$

For all process states that evolve from the initial state $(x{:}\langle\mathsf{gen}\rangle\ ord)$ under the signature Σ_{Gen}, restriction to Σ_{PDA} is the identity function whenever it is defined. Therefore, in the statement of Theorem 4.6, we use restriction as a judgment $\Delta\lightning_{\Sigma_{PDA}}$ that holds whenever the partial function is defined.

Theorem 4.6 (Encoding)**.** *Up to variable renaming, there is a bijective correspondence between PDA states $k \rhd s$ and process states Δ such that $T :: (x{:}\langle\mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen}} \Delta$ and $\Delta\lightning_{\Sigma_{PDA}}$.*

Proof. To establish the bijective correspondence, we first define an encoding function from PDA states to process states:

* $\ulcorner k \rhd s \urcorner = \ulcorner k \urcorner,\ \ h{:}\langle\mathsf{hd}\rangle\ ord,\ \ \ulcorner s \urcorner$
* $\ulcorner . \urcorner = \cdot$
* $\ulcorner k < \urcorner = \ulcorner k \urcorner,\ \ x{:}\langle\mathsf{L}\rangle\ ord$
* $\ulcorner < s \urcorner = y{:}\langle\mathsf{L}\rangle\ ord,\ \ \ulcorner s \urcorner$
* $\ulcorner > s \urcorner = y{:}\langle\mathsf{R}\rangle\ ord,\ \ \ulcorner s \urcorner$

It is always the case that $\ulcorner k \rhd s \urcorner\lightning_{\Sigma_{PDA}}$ – the encoding only includes terminals.

It is straightforward to observe that if $\ulcorner k \rhd s \urcorner = \ulcorner k' \rhd s' \urcorner$ if an only if $k = k'$ and $s = s'$. The interesting part of showing that context interpretation is an injective function is just showing that it is a function: that is, showing that, for any $k \rhd s$, there exists a trace

$T :: (x{:}\langle\mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{Gen} \ulcorner k \triangleright s \urcorner$. To show that the encoding function is surjective, we must show that if $T :: (x{:}\langle\mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen}} \Delta$ and $\Delta \not\rightsquigarrow_{\Sigma_{PDA}}$ then $\Delta = \ulcorner k \triangleright s \urcorner$ for some k and s. This will complete the proof: an injective and surjective function is bijective.

Encoding is injective

We prove that for any $k \triangleright s$, there exists a trace $T :: (x{:}\langle\mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{Gen} \ulcorner k \triangleright s \urcorner$ with a series of three lemmas.

Lemma. *For all k, there exists $T :: (x{:}\langle\mathsf{gen\_stack}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen}} (\ulcorner k \urcorner, x'{:}\langle\mathsf{gen\_stack}\rangle\ ord)$.*

By induction on k.

- If $k = \cdot$, $T = \diamond :: (x{:}\langle\mathsf{gen\_stack}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen}} (x{:}\langle\mathsf{gen\_stack}\rangle\ ord)$
- If $k = k'<$, we have $T' :: (x{:}\langle\mathsf{gen\_stack}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen}} (\ulcorner k' \urcorner, x''{:}\langle\mathsf{gen\_stack}\rangle\ ord)$ by the induction hypothesis, so $T = (T'; \{x_1, x_2\} \leftarrow \mathsf{stack/left}\, x'') :: (x{:}\langle\mathsf{gen\_stack}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen}} (\ulcorner k' \urcorner, x_1{:}\langle\mathsf{L}\rangle\ ord, x_2{:}\langle\mathsf{gen\_stack}\rangle\ ord)$

Lemma. *For all s, there exists $T :: (y{:}\langle\mathsf{gen\_string}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen}} (y'{:}\langle\mathsf{gen\_string}\rangle\ ord, \ulcorner s \urcorner)$.*

By induction on s.

- If $s = \cdot$, $T = \diamond :: (y{:}\langle\mathsf{gen\_string}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen}} (y{:}\langle\mathsf{gen\_string}\rangle\ ord)$
- If $s = <s'$, we have $T' :: (y{:}\langle\mathsf{gen\_string}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen}} (y''{:}\langle\mathsf{gen\_string}\rangle\ ord, \ulcorner s' \urcorner)$ by the induction hyp., so $T = (T'; \{y_1, y_2\} \leftarrow \mathsf{string/left}\, y'') :: (y{:}\langle\mathsf{gen\_string}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen}} (y_1{:}\langle\mathsf{gen\_string}\rangle\ ord, y_2{:}\langle\mathsf{L}\rangle\ ord, \ulcorner s' \urcorner)$
- If $s = >s'$, we have $T' :: (y{:}\langle\mathsf{gen\_string}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen}} (y''{:}\langle\mathsf{gen\_string}\rangle\ ord, \ulcorner s' \urcorner)$ by the induction hyp., so $T = (T'; \{y_1, y_2\} \leftarrow \mathsf{string/right}\, y'') :: (y{:}\langle\mathsf{gen\_string}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen}} (y_1{:}\langle\mathsf{gen\_string}\rangle\ ord, y_2{:}\langle\mathsf{R}\rangle\ ord, \ulcorner s' \urcorner)$

Lemma. *For all k and s, there exists $T :: (g{:}\langle\mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen}} (\ulcorner k \triangleright s \urcorner)$.*

By straightforward construction using the first two lemmas and frame weakening (Theorem 4.2):

$$
\begin{array}{l}
\qquad (g{:}\langle\mathsf{gen}\rangle\ ord) \\
\{x, h, y\} \leftarrow \mathsf{state}\, g; \\
\qquad (x{:}\langle\mathsf{gen\_stack}\rangle\ ord,\ \ h{:}\langle\mathsf{hd}\rangle\ ord,\ \ y{:}\langle\mathsf{gen\_string}\rangle\ ord) \\
T_k;\ \ \textit{(given by the first lemma and frame weakening)} \\
\qquad (\ulcorner k \urcorner,\ \ x'{:}\langle\mathsf{gen\_stack}\rangle\ ord,\ \ h{:}\langle\mathsf{hd}\rangle\ ord,\ \ y{:}\langle\mathsf{gen\_string}\rangle\ ord) \\
\{()\} \leftarrow \mathsf{stack/done}\, x' \\
\qquad (\ulcorner k \urcorner,\ \ h{:}\langle\mathsf{hd}\rangle\ ord,\ \ y{:}\langle\mathsf{gen\_string}\rangle\ ord) \\
T_s;\ \ \textit{(given by the second lemma and frame weakening)} \\
\qquad (\ulcorner k \urcorner,\ \ h{:}\langle\mathsf{hd}\rangle\ ord,\ \ y'{:}\langle\mathsf{gen\_string}\rangle\ ord,\ \ \ulcorner s \urcorner) \\
\{()\} \leftarrow \mathsf{string/done}\, y' \\
\qquad (\ulcorner k \urcorner,\ \ h{:}\langle\mathsf{hd}\rangle\ ord,\ \ \ulcorner s \urcorner) \\
\qquad = \ulcorner k \triangleright s \urcorner
\end{array}
$$

Encoding is surjective

We prove that if $T :: (x{:}\langle \mathsf{gen} \rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen}} \Delta$ and $\Delta \lightning_{\Sigma_{PDA}}$ then $\Delta = \ulcorner k \triangleright s \urcorner$ for some k and s with a series of two lemmas.

Lemma. *If* $T :: (\ulcorner k \urcorner, x{:}\langle \mathsf{gen\_stack} \rangle\ ord, h{:}\langle \mathsf{hd} \rangle\ ord, y{:}\langle \mathsf{gen\_store} \rangle\ ord, \ulcorner s \urcorner) \rightsquigarrow^*_{\Sigma_{Gen}} \Delta$ *and* $\Delta \lightning_{\Sigma_{PDA}}$, *then* $\Delta = \ulcorner k' \triangleright s' \urcorner$ *for some* k' *and* s'.

By induction on the structure of T and case analysis on the first steps in T. Up to concurrent equality, there are four possibilities:

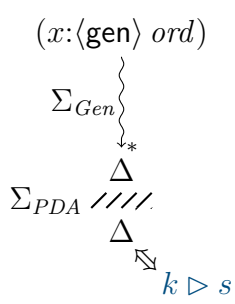
* $T = (\{()\} \leftarrow \mathsf{stack/done}\, x; \{()\} \leftarrow \mathsf{string/done}\, y)$ – this is a base case, and we can finish by letting $k' = k$ and $s' = s$.
* $T = (\{x_1, x_2\} \leftarrow \mathsf{stack/left}\, x; T')$ – apply the ind. hyp. (letting $x = x_2$, $k = k<$).
* $T = (\{y_1, y_2\} \leftarrow \mathsf{string/left}\, y; T')$ – apply the ind. hyp. (letting $y = y_1$, $s = <s$).
* $T = (\{y_1, y_2\} \leftarrow \mathsf{string/right}\, y; T')$ – apply the ind. hyp. (letting $y = y_1$, $s = >s$).

The proof above takes a number of facts about concurrent equality for granted. For example, the trace $T = (\{()\} \leftarrow \mathsf{stack/done}\, x; \{y_1, y_2\} \leftarrow \mathsf{string/right}\, y; T')$ does not syntactically match any of the traces above if we do not account for concurrent equality. Modulo concurrent equality, on the other hand, $T = (\{y_1, y_2\} \leftarrow \mathsf{string/right}\, y; \{()\} \leftarrow \mathsf{stack/done}\, x; T')$, matching the last branch of the case analysis. If we didn't implicitly rely on concurrent equality in this way, the resulting proof would have twice as many cases. We will take these finite uses of concurrent equality for granted when we specify that a proof proceeds by case analysis on the first steps of T (or, conversely, by case analysis on the last steps of T).

Lemma. *If* $T :: (g{:}\langle \mathsf{gen} \rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen}} \Delta$ *and* $\Delta \lightning_{\Sigma_{PDA}}$, *then* $\Delta = \ulcorner k' \triangleright s' \urcorner$ *for some* k' *and* s'.

This is a corollary of the previous lemma, as it can only be the case that $T = \{x, h, y\} \leftarrow \mathsf{state}\, g; T'$. We can apply the previous lemma to T', letting $k = s = \cdot$. This establishes that encoding is a surjective function, which in turn completes the proof. $\square$

Theorem 4.6 establishes that the generative signature Σ_{Gen} describes a world – a set of SLS process states – that precisely corresponds to the states of a push-down automata. We can (imperfectly) illustrate the content of this theorem in our two-dimensional notation as follows, where $\Delta \Leftrightarrow k \triangleright s$ indicates the presence of a bijection:



It is interesting to note how the proof of Theorem 4.6 takes advantage of the associative structure of traces: the inductive process that constructed traces in the first two lemmas treated

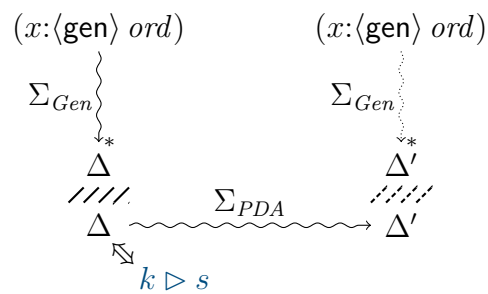
trace composition as left-associative, but the induction we performed on traces in the next-to-last lemma treated trace composition as right-associative.

4.4.3 Generative invariants

The generative signature Σ_{Gen} precisely captures the world of SLS process states that are in the image of the encoding $\ulcorner k \triangleright s \urcorner$ of PDA states as process states. In order for the signature Σ_{PDA} to encode a reasonable notion of transition between PDA states, we need to show that steps in this signature only take encoded PDA states to encoded PDA states. Because the generative signature Σ_{Gen} precisely captures the process states that represent encoded PDA states, we can describe and prove this property without reference to the actual encoding function:

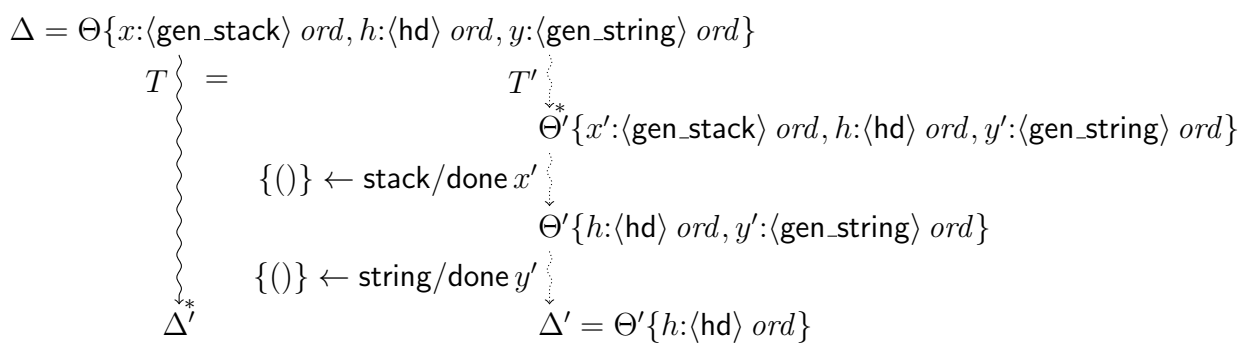
Theorem 4.7 (Preservation)**.** *If* $T_1 :: (x{:}\langle \mathsf{gen} \rangle\ ord) \leadsto^*_{\Sigma_{Gen}} \Delta_1$, $\Delta_1 \lightning_{\Sigma_{PDA}}$, *and* $S :: \Delta_1 \leadsto_{\Sigma_{PDA}} \Delta_2$, *then* $T_2 :: (x{:}\langle \mathsf{gen} \rangle\ ord) \leadsto^*_{\Sigma_{Gen}} \Delta_2$.

If we illustrate the given elements as solid lines and elements that we have to prove as dashed lines, the big picture of the encoding and preservation theorems is the following:



The proof of Theorem 4.7 relies on two lemmas, which we will consider before the proof itself. They are both *inversion lemmas*: they help uncover the structure of a trace based on the type of that trace. Treating traces modulo concurrent equality is critical in both cases.

Lemma. *Let* $\Delta = \Theta\{x{:}\langle \mathsf{gen\_stack} \rangle\ ord, h{:}\langle \mathsf{hd} \rangle\ ord, y{:}\langle \mathsf{gen\_string} \rangle\ ord\}$. *If* $T :: \Delta \leadsto^*_{\Sigma_{Gen}} \Delta'$ *and* $\Delta' \lightning_{\Sigma_{PDA}}$, *then* $T = (T'; \{()\} \leftarrow \mathsf{stack/done}\, x'; \{()\} \leftarrow \mathsf{string/done}\, y')$, *where* $T' :: \Delta \leadsto^*_{\Sigma_{Gen}} \Theta'\{x'{:}\langle \mathsf{gen\_stack} \rangle\ ord, h{:}\langle \mathsf{hd} \rangle\ ord, y'{:}\langle \mathsf{gen\_string} \rangle\ ord\}$ *and* $\Delta' = \Theta'\{h{:}\langle \mathsf{hd} \rangle\ ord\}$. *Or, as a picture:*



Proof. By induction on the structure of T and case analysis on the first steps in T. Up to concurrent equality, there are five possibilities:

* $T = (\{()\} \leftarrow \mathsf{stack/done}\, x; \{()\} \leftarrow \mathsf{string/done}\, y)$. Immediate, letting $T' = \diamond$.
* $T = (\{x_1, x_2\} \leftarrow \mathsf{stack/left}\, x; T'')$. By the induction hypothesis (where the new frame incorporates $x_1{:}\langle \mathsf{L}\rangle\ ord$), we have $T'' = (T'''; \{()\} \leftarrow \mathsf{stack/done}\, x; \{()\} \leftarrow \mathsf{string/done}\, y)$. Let $T' = (\{x_1, x_2\} \leftarrow \mathsf{stack/left}\, x; T''')$.
* $T = (\{y_1, y_2\} \leftarrow \mathsf{string/left}\, y; T'')$. By the induction hypothesis (where the new frame incorporates $y_1{:}\langle \mathsf{L}\rangle\ ord$), we have $T'' = (T'''; \{()\} \leftarrow \mathsf{stack/done}\, x; \{()\} \leftarrow \mathsf{string/done}\, y)$. Let $T' = (\{x_1, x_2\} \leftarrow \mathsf{string/left}\, y; T''')$.
* $T = (\{y_1, y_2\} \leftarrow \mathsf{string/right}\, y; T'')$. By the induction hypothesis (where the new frame incorporates $y_2{:}\langle \mathsf{R}\rangle\ ord$), we have $T'' = (T'''; \{()\} \leftarrow \mathsf{stack/done}\, x; \{()\} \leftarrow \mathsf{string/done}\, y)$. Let $T' = (\{x_1, x_2\} \leftarrow \mathsf{string/right}\, y; T''')$.
* $T = (S; T'')$, where x and y are not free in S. By the induction hypothesis, we have $T'' = (T'''; \{()\} \leftarrow \mathsf{stack/done}\, x; \{()\} \leftarrow \mathsf{string/done}\, y)$. Let $T' = (S; T''')$. (This case will not arise in the way we use this lemma, but the statement of the theorem leaves open the possibility that there are other nonterminals in Θ.)

This completes the proof. □

A corollary of this lemma is that if $T :: (g{:}\langle \mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen}} \Delta$ and $\Delta \lightning_{\Sigma_{PDA}}$, then $T = (T'; \{()\} \leftarrow \mathsf{stack/done}\, x; \{()\} \leftarrow \mathsf{string/done}\, y)$ – modulo concurrent equality, naturally – where $T' :: (g{:}\langle \mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen}} \Theta\{x'{:}\langle \mathsf{gen\_stack}\rangle\ ord, h{:}\langle \mathsf{hd}\rangle\ ord, y'{:}\langle \mathsf{gen\_string}\rangle\ ord\}$ and $\Delta = \Theta\{h{:}\langle \mathsf{hd}\rangle\ ord\}$. To prove the corollary, we observe that $T = (\{x, h, r\} \leftarrow \mathsf{state}\, g\, ; T'')$ and apply the lemma to T''.

Lemma. *The following all hold:*

* *If* $T :: (g{:}\langle \mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen}} \Theta\{x_1{:}\langle \mathsf{L}\rangle\ ord, x_2{:}\langle \mathsf{gen\_stack}\rangle\ ord\}$, *then* $T = (T'; \{x_1, x_2\} \leftarrow \mathsf{stack/left}\, x')$ *for some* x'.
* *If* $T :: (g{:}\langle \mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen}} \Theta\{y_1{:}\langle \mathsf{gen\_string}\rangle\ ord, y_2{:}\langle \mathsf{L}\rangle\ ord\}$, *then* $T = (T'; \{y_1, y_2\} \leftarrow \mathsf{string/left}\, y')$ *for some* y'.
* *If* $T :: (g{:}\langle \mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen}} \Theta\{y_1{:}\langle \mathsf{gen\_string}\rangle\ ord, y_2{:}\langle \mathsf{R}\rangle\ ord\}$, *then* $T = (T'; \{y_1, y_2\} \leftarrow \mathsf{string/right}\, y')$ *for some* y'.

To give the last of the three statements as a picture:

$$
\begin{array}{ccc}
g{:}\langle \mathsf{gen}\rangle\ ord & & g{:}\langle \mathsf{gen}\rangle\ ord \\
T \rightsquigarrow & = & T' \rightsquigarrow^* \\
 & & \Theta'\{y'{:}\langle \mathsf{gen\_string}\rangle\ ord\} \\
 & & \{y_1, y_2\} \leftarrow \mathsf{string/right}\, y' \rightsquigarrow \\
\Theta'\{y_1{:}\langle \mathsf{gen\_string}\rangle\ ord, y_2{:}\langle \mathsf{R}\rangle\ ord\} & & \Theta'\{y_1{:}\langle \mathsf{gen\_string}\rangle\ ord, y_2{:}\langle \mathsf{R}\rangle\ ord\}
\end{array}
$$

Proof. The proofs are all by induction on the structure of T and case analysis on the last steps in T; we will prove the last statement, as the other two are similar. Up to concurrent equality, there are two possibilities:

* $T = (T'; \{y_1, y_2\} \leftarrow \mathsf{string/right}\, y')$ – Immediate.
* $T = (T''; S)$, where y_1 and y_2 are not among the input variables ${}^{\bullet}S$ or the output variables $S^{\bullet}$. By the induction hypothesis, $T'' = (T'''; \{y_1, y_2\} \leftarrow \mathsf{string/right}\, y')$. Let $T' = (T''; S)$.

This completes the proof. □

Note that we do not consider any cases where $T = (T'; \{y_1', y_2\} \leftarrow \mathsf{string/right}\, y')$ (for $y_1 \neq y_1'$), $T = (T'; \{y_1', y_2\} \leftarrow \mathsf{string/right}\, y')$ (for $y_2 \neq y_2'$), or (critically) where $T = (T'; \{y_1, y_2'\} \leftarrow \mathsf{string/left}\, y')$. There is no way for any of these traces to have the correct type, which makes the resulting case analysis quite simple.

Proof of Theorem 4.7 (Preservation). By case analysis on the structure of S.

Case 1: $S = \{x', h'\} \leftarrow \mathsf{push}\,(h \bullet y)$, which means that we are given the following generative trace in Σ_{Gen}:

$$T \quad \begin{array}{l} (g{:}\langle\mathsf{gen}\rangle\ ord) \\ \Theta\{h{:}\langle\mathsf{hd}\rangle\ ord,\ \ y{:}\langle\mathsf{L}\rangle\ ord\} \end{array}$$

and we must construct a trace $(g{:}\langle\mathsf{gen}\rangle\ ord) \rightsquigarrow^{*}_{\Sigma_{Gen}} \Theta\{x'{:}\langle\mathsf{L}\rangle\ ord, h'{:}\langle\mathsf{hd}\rangle\ ord\}$. Changing h to h' is just renaming a bound variable, so we have

$$T' \quad \begin{array}{l} (g{:}\langle\mathsf{gen}\rangle\ ord) \\ \Theta\{h'{:}\langle\mathsf{hd}\rangle\ ord,\ \ y{:}\langle\mathsf{L}\rangle\ ord\} \end{array}$$

The corollary to the first inversion lemma above on T' gives us

$$\begin{array}{ll} T' = & (g{:}\langle\mathsf{gen}\rangle\ ord) \\ & T''; \\ & \quad \Theta\{x_g{:}\langle\mathsf{gen\_stack}\rangle\ ord,\ \ h'{:}\langle\mathsf{hd}\rangle\ ord,\ \ y_g{:}\langle\mathsf{gen\_string}\rangle\ ord,\ \ y{:}\langle\mathsf{L}\rangle\ ord\} \\ & \{()\} \leftarrow \mathsf{stack/done}\, x_g; \\ & \{()\} \leftarrow \mathsf{string/done}\, y_g \\ & \quad \Theta\{h'{:}\langle\mathsf{hd}\rangle\ ord,\ \ y{:}\langle\mathsf{L}\rangle\ ord\} \end{array}$$

The second inversion lemma (second part) on T'' gives us

$$\begin{array}{ll} T'' = & (g{:}\langle\mathsf{gen}\rangle\ ord) \\ & T'''; \\ & \quad \Theta\{x_g{:}\langle\mathsf{gen\_stack}\rangle\ ord,\ \ h'{:}\langle\mathsf{hd}\rangle\ ord,\ \ y_g'{:}\langle\mathsf{gen\_string}\rangle\ ord\} \\ & \{y_g, y\} \leftarrow \mathsf{string/left}\, y_g' \\ & \quad \Theta\{x_g{:}\langle\mathsf{gen\_stack}\rangle\ ord,\ \ h'{:}\langle\mathsf{hd}\rangle\ ord,\ \ y_g{:}\langle\mathsf{gen\_string}\rangle\ ord,\ \ y{:}\langle\mathsf{L}\rangle\ ord\} \end{array}$$

Now, we can construct the trace we need using T''':

$$
\begin{array}{l}
\quad (g{:}\langle\mathsf{gen}\rangle\ ord) \\
T'''; \\
\quad \Theta\{x_g{:}\langle\mathsf{gen\_stack}\rangle\ ord,\ \ h'{:}\langle\mathsf{hd}\rangle\ ord,\ \ y'_g{:}\langle\mathsf{gen\_string}\rangle\ ord\} \\
\{x', x'_g\} \leftarrow \mathsf{stack/left}\, x_g; \\
\quad \Theta\{x'{:}\langle\mathsf{L}\rangle\ ord,\ \ x'_g{:}\langle\mathsf{gen\_stack}\rangle\ ord,\ \ h'{:}\langle\mathsf{hd}\rangle\ ord,\ \ y'_g{:}\langle\mathsf{gen\_string}\rangle\ ord\} \\
\{()\} \leftarrow \mathsf{stack/done}\, x'_g; \\
\{()\} \leftarrow \mathsf{string/done}\, y'_g \\
\quad \Theta\{x'{:}\langle\mathsf{L}\rangle\ ord,\ \ h'{:}\langle\mathsf{hd}\rangle\ ord\}
\end{array}
$$

Case 2: $S = \{h'\} \leftarrow \mathsf{pop}\,(x \bullet h \bullet y)$, which means that we are given the following generative trace in Σ_{Gen}:

$$
\begin{array}{l}
\quad (g{:}\langle\mathsf{gen}\rangle\ ord) \\
T \\
\quad \Theta\{x{:}\langle\mathsf{L}\rangle\ ord,\ \ h{:}\langle\mathsf{hd}\rangle\ ord,\ \ y{:}\langle\mathsf{R}\rangle\ ord\}
\end{array}
$$

and we must construct a trace $(g{:}\langle\mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen}} \Theta\{h'{:}\langle\mathsf{hd}\rangle\ ord\}$. Changing h to h' is just renaming a bound variable, so we have

$$
\begin{array}{l}
\quad (g{:}\langle\mathsf{gen}\rangle\ ord) \\
T' \\
\quad \Theta\{x{:}\langle\mathsf{L}\rangle\ ord,\ \ h'{:}\langle\mathsf{hd}\rangle\ ord,\ \ y{:}\langle\mathsf{R}\rangle\ ord\}
\end{array}
$$

The corollary to the first inversion lemma above on T' gives us

$$
\begin{array}{l}
T' = \quad (g{:}\langle\mathsf{gen}\rangle\ ord) \\
\quad T''; \\
\quad\quad \Theta\{x{:}\langle\mathsf{L}\rangle\ ord,\ \ x_g{:}\langle\mathsf{gen\_stack}\rangle\ ord,\ \ h'{:}\langle\mathsf{hd}\rangle\ ord,\ \ y_g{:}\langle\mathsf{gen\_string}\rangle\ ord,\ \ y{:}\langle\mathsf{R}\rangle\ ord\} \\
\quad \{()\} \leftarrow \mathsf{stack/done}\, x_g; \\
\quad \{()\} \leftarrow \mathsf{string/done}\, y_g \\
\quad\quad \Theta\{x{:}\langle\mathsf{L}\rangle\ ord,\ \ h'{:}\langle\mathsf{hd}\rangle\ ord,\ \ y{:}\langle\mathsf{R}\rangle\ ord\}
\end{array}
$$

The second inversion lemma (first part) on T'' gives us

$$
\begin{array}{l}
T'' = \quad (g{:}\langle\mathsf{gen}\rangle\ ord) \\
\quad T'''; \\
\quad\quad \Theta\{x'_g{:}\langle\mathsf{gen\_stack}\rangle\ ord,\ \ h'{:}\langle\mathsf{hd}\rangle\ ord,\ \ y_g{:}\langle\mathsf{gen\_string}\rangle\ ord,\ \ y{:}\langle\mathsf{R}\rangle\ ord\} \\
\quad \{x, x_g\} \leftarrow \mathsf{stack/left}\, x'_g \\
\quad\quad \Theta\{x{:}\langle\mathsf{L}\rangle\ ord,\ \ x_g{:}\langle\mathsf{gen\_stack}\rangle\ ord,\ \ h'{:}\langle\mathsf{hd}\rangle\ ord,\ \ y_g{:}\langle\mathsf{gen\_string}\rangle\ ord,\ \ y{:}\langle\mathsf{R}\rangle\ ord\}
\end{array}
$$

The second inversion lemma (third part) on T''' gives us

$$
\begin{array}{rl}
T''' = & (g{:}\langle\mathsf{gen}\rangle\ ord) \\
& T'''';\\
& \quad \Theta\{x'_g{:}\langle\mathsf{gen\_stack}\rangle\ ord,\ \ h'{:}\langle\mathsf{hd}\rangle\ ord,\ \ y'_g{:}\langle\mathsf{gen\_string}\rangle\ ord\} \\
& \{y_g, y\} \leftarrow \mathsf{string/right}\ y'_g \\
& \quad \Theta\{x'_g{:}\langle\mathsf{gen\_stack}\rangle\ ord,\ \ h'{:}\langle\mathsf{hd}\rangle\ ord,\ \ y_g{:}\langle\mathsf{gen\_string}\rangle\ ord,\ \ y{:}\langle\mathsf{R}\rangle\ ord\}
\end{array}
$$

Now, we can construct the trace we need using T'''':

$$
\begin{array}{l}
\quad (g{:}\langle\mathsf{gen}\rangle\ ord) \\
T'''';\\
\quad \Theta\{x'_g{:}\langle\mathsf{gen\_stack}\rangle\ ord,\ \ h'{:}\langle\mathsf{hd}\rangle\ ord,\ \ y'_g{:}\langle\mathsf{gen\_string}\rangle\ ord\} \\
\{()\} \leftarrow \mathsf{stack/done}\ x'_g; \\
\{()\} \leftarrow \mathsf{string/done}\ y'_g \\
\quad \Theta\{h'{:}\langle\mathsf{hd}\rangle\ ord\}
\end{array}
$$

These two cases represent the only two synthetic transitions that are possible under the signature Σ_{PDA}, so we are done. □

Theorem 4.7 establishes that the generative signature Σ_{Gen} is a *generative invariant* of the signature Σ_{PDA}. We consider theorems of this form further in Chapter 9, but they all essentially follow the structure of Theorem 4.7. First, we enumerate the synthetic transitions associated with a given signature. Second, in each of those cases, we use the type of the synthetic transition to perform inversion on the structure of the given generative trace. Third, we construct a generative trace that establishes the fact that the invariant was preserved.

4.4.4 Adequacy of the transition system

The hard work of adequacy is established by the preservation theorem; the actual adequacy theorem is just an enumeration in both directions.

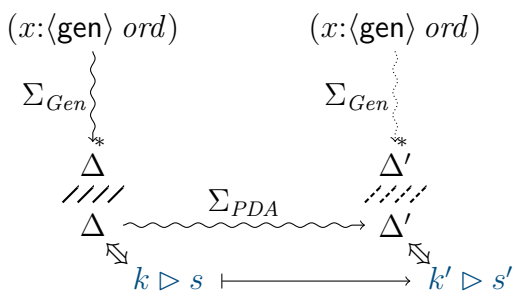
Theorem 4.8 (Adequacy). $\ulcorner k \rhd s \urcorner \rightsquigarrow_{\Sigma_{PDA}} \ulcorner k' \rhd s' \urcorner$ *if and only if* $k \rhd s \mapsto k' \rhd s'$.

Proof. Both directions can be established by case analysis on the structure of k and s. □

As an immediate corollary of this theorem and preservation (Theorem 4.7), we have the stronger adequacy property that $\ulcorner k \rhd s \urcorner \rightsquigarrow_{\Sigma_{PDA}} \Delta'$, then $\Delta' = \ulcorner k' \rhd s' \urcorner$ for some k and s' such that $k \rhd s \mapsto k' \rhd s'$. In our two-dimensional notation, the complete discussion of adequacy for SLS is captured by the following picture:

$\downarrow A^- =$ `A`	$\bigcirc A^+ =$ `{A}`	$\lambda a.t =$ `\a.t`
$¡A^- =$ `$A`	$A^+ \rightarrowtail B^- =$ `A >-> B`	$\mathsf{foo}\, t_1 \ldots t_n =$ `foo t1...tn`
$!A^- =$ `!A`	$A^+ \twoheadrightarrow B^- =$ `A ->> B`	
$\mathbf{1} =$ `one`	$A^- \,\&\, B^- =$ `A & B`	$\Pi a{:}\tau.\nu =$ `Pi x.nu`
$A^+ \bullet B^+ =$ `A * B`	$\forall a.\tau.A^- =$ `All x.A`	$\tau \rightarrow \nu =$ `tau -> nu`
$\exists a.\tau.A^+ =$ `Exists x.A`	$¡A^- \rightarrowtail B^- =$ `A -o B`	$\mathsf{bar}\, t_1 \ldots t_n =$ `bar t1...tn`
$t \doteq_\tau s =$ `t == s`	$!A^- \rightarrowtail B^- =$ `A -> B`	

Figure 4.16: Mathematical and ASCII representations of propositions, terms, and classifiers



4.5 The SLS implementation

The prototype implementation of SLS contains a parser and typechecker for the SLS language, and is available from `https://github.com/robsimmons/sls`. Code that is checked by this prototype implementation will appear frequently in the rest of this document, always in a `fixed-width font`.

The checked SLS code differs slightly from mathematical SLS specifications in a few ways – the translation between the mathematical notation we use for SLS propositions and the ASCII representation used in the implementation is outlined in Figure 4.16. Following CLF and the Celf implementation, we write the lax modality $\bigcirc A$ in ASCII as `{A}` – recall that in Section 4.2 we introduced the $\{A^+\}$ notation from CLF as a synonym for Fairtlough and Mendler's $\bigcirc A^+$. The exponential $¡A$ doesn't have an ASCII representation, so we write `$A`. Upshifts and downshifts are always inferred: this means that we can't write down $\uparrow\downarrow A$ or $\downarrow\uparrow A$, but neither of these OL_3 propositions are part of the SLS fragment anyway.

The SLS implementation also supports conventional abbreviations for arrows that we won't use in mathematical notation: $¡A^- \rightarrowtail B^-$ can be written as `A -o B` or `$A >-> B` in the SLS implementation, and $!A^- \rightarrowtail B^-$ can be written as `A -> B` or `!A >-> B`. This final proposition is ambiguous, because `X -> Y` can be an abbreviation for $!X \rightarrowtail Y$ or $\Pi a{:}X.Y$, but SLS can figure out which form was intended by analyzing the structure of `Y`. Also note that we could have just as easily made `A -o B` an abbreviation for `$A ->> B`, but we had to pick one and the choice absolutely doesn't matter. All arrows can also be written backwards: `B <-< A`

is equivalent to `A >-> B`, `B o- A` is equivalent to `A -o B`, and so on.

Also following traditional conventions, upper-case variables that are free in a rule will be treated as implicitly quantified. Therefore, the line

```
rule: foo X <- (bar Y -> baz Z).
```

will be reconstructed as the SLS declaration

$$\mathsf{rule} : \forall Y{:}\tau_1.\, \forall Z{:}\tau_2.\, \forall X{:}\tau_3.\, !(!\mathsf{bar}\, Y \rightarrowtail \mathsf{baz}\, Z) \rightarrowtail \mathsf{foo}\, X$$

where the implementation infers the types τ_1, τ_2, and τ_3 appropriately from the declarations of the negative predicates foo, bar, and baz. The type annotation associated with equality is similarly inferred.

Another significant piece of syntactic sugar introduced for the sake of readability is less conventional, if only because positive atomic propositions are not conventional. If `P` is a persistent atomic proposition, we can optionally write `!P` wherever `P` is expected, and if `P` is a linear atomic proposition, we can write `$P` wherever `P` is expected. This means that if a, b, and c are (respectively) ordered, linear, and persistent positive atomic propositions, we can write the positive proposition $\mathsf{a} \bullet \mathsf{b} \bullet \mathsf{c}$ in the SLS implementation as `(a * b * c)`, `(a * $b * c)`, `(a * b * !c)`, or `(a * $b * !c)`. Without these annotations, it is difficult to tell at a glance which propositions are ordered, linear, or persistent when a signature uses more than one variety of proposition. When all of these optional annotations are included, the rules in a signature that uses positive atomic propositions look the same as rules in a signature that uses the pseudo-positive negative atomic propositions described in Section 4.7.1.

In the code examples given in the remainder of this document, we will use these optional annotations in a consistent way. We will omit the optional `$A` annotations only in specifications with no ordered atomic propositions, and we will omit the optional `!A` annotations in specifications with no ordered or linear atomic propositions. This makes the mixture of different exponentials explicit while avoiding the need for rules like `($a * $b * $c >-> {$d * $e})` when specifications are entirely linear (and likewise when specifications are entirely persistent).

4.6 Logic programming

One logic programming interpretation of CLF was explored by the Lollimon implementation [LPPW05] and adapted by the Celf implementation [SNS08, SN11]. Logic programming interpretations of SLS are not a focus this dissertation, but we will touch on a few points in this section.

Logic programming is important because it provides us with operational intuitions about the intended behavior of the systems we specify in SLS. One specific set of intuitions will form the basis of the operationalization transformations on SLS specifications considered in Chapter 6. Additionally, logic programming intuitions are relevant because they motivated the design of SLS, in particular the presentation of the concurrent fragment in terms of partial, rather than complete, proofs. We discuss this point in Section 4.6.2.

4.6.1 Deductive computation and backward chaining

Deductive computation in SLS is the search for *complete* proofs of sequents of the form $\Psi; \Delta \vdash \langle p^- \rangle\ true$. A common form of deductive computation is *goal-directed search*, or what Andreoli calls the *proof construction paradigm* [And01]. In SLS, goal-directed search for the proof of a sequent $\Psi; \Delta \vdash \langle p^- \rangle\ true$ can only proceed by focusing on a proposition like $\downarrow p_n^- \rightarrowtail \ldots \rightarrowtail \downarrow p_1^- \rightarrowtail p^-$ which has a head p^- that matches the succedent. This replaces the goal sequent $\Psi; \Delta \vdash \langle p^- \rangle\ true$ with n subgoals: $\Psi; \Delta_1 \vdash \langle p_1^- \rangle\ true \ldots \Psi; \Delta_n \vdash \langle p_n^- \rangle\ true$, where Δ matches $\Delta_1, \ldots, \Delta_n$.

When goal-directed search only deals with the unproved subgoals of a single coherent derivation at a time, it is called *backward chaining*, because we're working backwards from the goal we want to prove.[3] The term *top-down logic programming* is also used, and refers to the fact that, in the concrete syntax of Prolog, the rule $\downarrow p_n^- \rightarrowtail \ldots \rightarrowtail \downarrow p_1^- \rightarrowtail p^-$ would be written with p^- on the first line, p_1^- on the second, etc. This is exactly backwards from a proof-construction perspective, as we think of backward chaining as building partial proofs from the bottom up, the root towards the leaves, so we will avoid this terminology.

The backward-chaining interpretation of intuitionistic logics dates back to the work by Miller et al. on uniform proofs [MNPS91]. An even older concept, Clark's *negation-as-failure* [Cla87], is based on a *partial completeness* criteria for logic programming interpreters. Partial completeness demands that if the interpreter gives up up on finding a proof, no proof should exist. (The interpreter is allowed to run forever without succeeding or giving up.) Partial completeness requires *backtracking* in backward-chaining search: if we we try to prove $\Psi; \Delta \vdash \langle p^- \rangle\ true$ by focusing on a particular proposition and one of the resulting subgoals fails to be provable, we have to consider any other propositions that could have been used to prove the sequent before giving up. Backtracking can be extremely powerful in certain cases and incredibly expensive in others, and so most logic programming languages have an escape hatch that modifies or limits backtracking at the user's discretion, such as the Prolog cut (no relation to the admissible rule cut) or Twelf's deterministic declarations. Non-backtracking goal-oriented deductive computation is called *flat resolution* [AK99].

One feature of backward chaining and goal directed search is that it usually allows for terms that are not completely specified – these unspecified pieces are are traditionally called *logic variables*. Because LF variables are also "logic variables," the literature on λProlog and Twelf calls unspecified pieces of terms *existential variables*, but as they bear no relation to the variables introduced by the left rule for $\exists a{:}\tau.A^+$, that terminology is also unhelpful here. Consider the

[3]The alternative is to try and derive the same sequent in multiple ways simultaneously, succeeding whenever some way of proving the sequent is discovered. Unlike backward chaining, this strategy of breadth-first search is complete: if a proof exists, it will be found. Backward chaining as we define it is only nondeterministically or partially complete, because it can fail to terminate when a proof exists. We will call this alternative to backtracking *breadth-first theorem proving*, as it amounts to taking a breadth-first, instead of depth-first, view of the so-called *failure continuation* [Pfe12a].

following SLS signature:

$$\begin{aligned}\Sigma_{Add} = \cdot,\ & \mathsf{nat} : \mathsf{type},\ \ \mathsf{z} : \mathsf{nat},\ \ \mathsf{s} : \mathsf{nat} \to \mathsf{nat},\\ & \mathsf{plus} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat} \to \mathsf{prop},\\ & \mathsf{plus/z} : \forall N{:}\mathsf{nat}.\, (\mathsf{plus}\, \mathsf{z}\, N\, N),\\ & \mathsf{plus/s} : \forall N{:}\mathsf{nat}.\, \forall M{:}\mathsf{nat}.\, \forall P{:}\mathsf{nat}.\, !(\mathsf{plus}\, N\, M\, P) \rightarrowtail (\mathsf{plus}\, (\mathsf{s}\, N)\, M\, (\mathsf{s}\, P))\end{aligned}$$

In addition to searching for a proof of $\mathsf{plus}\,(\mathsf{s}\,\mathsf{z})\,(\mathsf{s}\,\mathsf{z})\,(\mathsf{s}\,(\mathsf{s}\,\mathsf{z}))$ (which will succeed, as $1 + 1 = 2$) or searching for a proof of $\mathsf{plus}\,(\mathsf{s}\,\mathsf{z})\,(\mathsf{s}\,\mathsf{z})\,(\mathsf{s}\,(\mathsf{s}\,(\mathsf{s}\,\mathsf{z})))$ (which will fail, as $1 + 1 \neq 3$), we can use goal-oriented deductive computation to search for $\mathsf{plus}\,(\mathsf{s}\,\mathsf{z})\,(\mathsf{s}\,\mathsf{z})\,X$, where X represents an initially unspecified term. This search will succeed, reporting that $X = (\mathsf{s}\,(\mathsf{s}\,\mathsf{z}))$. Unification is generally used in backward-chaining logic programming languages as a technique for implementing partially unspecified terms, but this implementation technique should not be confused with our use of unification-based equality $t \doteq s$ as a proposition in SLS.

We say that plus in the signature above is a *well-moded* predicate with *mode* $(\mathsf{plus} + + -)$, because whenever we perform deductive computation to derive $(\mathsf{plus}\, n\, m\, p)$ where n and m are fully specified, any unspecified portion of p must be fully specified in any completed derivation. Well-moded predicates can be treated as nondeterministic (in the sense of potentially having zero, one, or many outputs) partial functions from their inputs (the indices marked "+" in the mode) to their outputs (the indices marked "−" in the mode). A predicate can sometimes be given more than one mode: $(\mathsf{plus} + - +)$ is a valid mode for plus, but $(\mathsf{plus} + - -)$ is not.

The implementation of backward chaining in substructural logic has been explored by Hodas [HM94], Polakow [Pol00, Pol01], Armelín and Pym [AP01], and others. Efficient implementation of these languages is complicated by the problem of *resource management*. In linear logic proof search, it would be technically correct but highly inefficient to perform proof search by enumerating the ways that a context can be split and then backtracking over each possible split. Resource management allows the interpreter to avoid this potentially exponential backtracking, but describing resource management and proving it correct, especially for richer substructural logics, can be complex and subtle [CHP00].

The term *deductive computation* is meant to be interpreted very broadly, and goal-directed search is not the only form of deductive computation. Another paradigm for deductive computation is the *inverse method*, where the interpreter attempts to prove a sequent $\Psi; \Delta \vdash \langle p^- \rangle\ \mathit{true}$ by creating and growing database of sequents that are derivable, attempting to build the appropriate derivation from the leaves down. The inverse method is generally associated with theorem proving and not logic programming. However, Chaudhuri, Pfenning, and Price have shown that that deductive computation with the inverse method in a focused linear logic can simulate both backward chaining and forward chaining (considered below) for persistent Horn-clause logic programs [CPP08].

Figure 4.17 gives an taxonomy (incomplete and imperfect) of the forms of deductive computation mentioned in this section. Note that, while we will generally use *backward chaining* to describe backtracking search, backward chaining does not always imply full backtracking and partial completeness. This illustration, and the preceding discussion, leaves out many important categories, especially tabled logic programming, and many potentially relevant implementation choices, such as breath-first versus depth-first or parallel exploration of the success continuation.

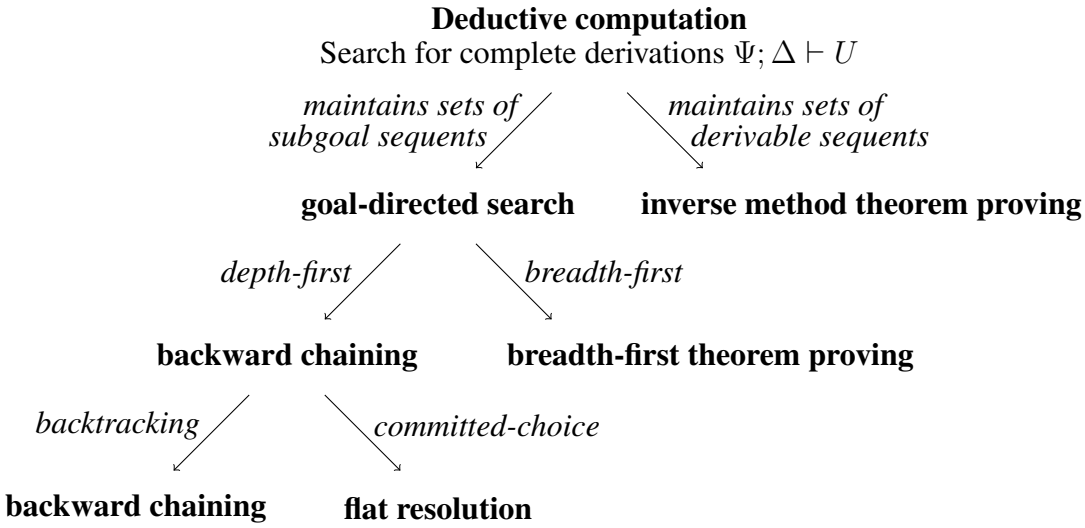


Figure 4.17: A rough taxonomy of deductive computation

4.6.2 Concurrent computation

Concurrent computation is the search for *partial* proofs of sequents. As the name suggests, in SLS concurrent computation is associated with the search for partial proofs of the judgment $A^+\ lax$, which correspond to traces $(\Psi; \Delta) \rightsquigarrow^* (\Psi'; \Delta')$.

The paradigm we will primarily associate with concurrent computation is *forward chaining*, which implies that we take an initial process state $(\Psi; \Delta)$ and allow it to evolve freely by the application of synthetic transitions. Additional conditions can be imposed on forward chaining: for instance, synthetic transitions like $(\Delta, x{:}\langle p^+_{pers}\rangle\ pers) \rightsquigarrow (\Delta, x{:}\langle p^+_{pers}\rangle\ pers, y{:}\langle p^+_{pers}\rangle\ pers)$ that do not meaningfully change the state can be excluded (if a persistent proposition already exists, two copies of that proposition don't add anything).[4] Forward chaining with this restriction in a purely-persistent logic is strongly associated with the Datalog language and its implementations; we will refer to forward chaining in persistent logics as *saturating logic programming* in Chapter 8. Forward chaining does not always deal with partially-unspecified terms; when persistent logic programming languages support forward chaining with partially-unspecified terms, it is called *hyperresolution* [FLHT01].

The presence of ephemeral or ordered resources in substructural logic means that a process state may evolve in multiple mutually-incompatible ways. *Committed choice* is a version of forward chaining that never goes back and reconsiders alternative evolutions from the initial state. Just as the default interpretation of backward chaining includes backtracking, we will consider the default interpretation of forward chaining to be committed choice, following [LPPW05]. An alternative interpretation would consider multiple evolutionary paths, which is a version of *exhaustive search*. Trace computation that works backwards from a final state instead of forward from an initial state can also be considered, and *planning* can be seen as specifying both the initial and final process states and trying to extrapolate a trace between them by working in both

[4]Incidentally, Lollimon implements this restriction and Celf, as of version 2.9, does not.

directions.

Outside of this work and Saurin's work on Ludics programming [Sau08], there is not much work on explicitly characterizing and searching for partial proofs in substructural logics.[5] Other forms of computation can be characterized as trace computation, however. Multiset rewriting and languages like GAMMA can be partially or completely understood in terms of forward chaining in linear logic [CS09, BG96], and the ordered aspects of SLS allow it to capture fragments of rewriting logic. Rewriting logic, and in particular the Maude implementation of rewriting logic [CDE+11], implements both the committed choice and the exhaustive search interpretations, as well as a *model checking* interpretation that characterize sets of process states or traces using logical formulas. Constraint handling rules [BRF10] and concurrent constraint programming [JNS05] are other logic programming models can be characterized as forms of concurrent computation.

4.6.3 Integrating deductive and trace computation

In the logic programming interpretation of CLF used by Lollimon and Celf, backtracking backward chaining is associated with the deductive fragment, and committed-choice forward chaining is associated with the lax modality. We will refer to an adaptation of the Lollimon/Celf semantics to SLS as "the Lollimon semantics" for brevity in this section.

Forward chaining and backward chaining have an uneasy relationship in the Lollimon semantics. Consider the following SLS signature:

$$\begin{aligned}\Sigma_{Demo} = \cdot,\ & \mathsf{posA} : \mathsf{prop}\,\mathsf{ord},\ \ \mathsf{posB} : \mathsf{prop}\,\mathsf{ord},\ \ \mathsf{posC} : \mathsf{prop}\,\mathsf{ord},\ \ \mathsf{negD} : \mathsf{prop},\\ & \mathsf{fwdruleAB} : \mathsf{posA} \rightarrowtail \bigcirc\mathsf{posB},\\ & \mathsf{fwdruleAC} : \mathsf{posA} \rightarrowtail \bigcirc\mathsf{posC},\\ & \mathsf{bwdrule} : (\mathsf{posA} \rightarrowtail \bigcirc\mathsf{posB}) \rightarrowtail \mathsf{negD}\end{aligned}$$

In an empty context, there is only one derivation of negD under this signature: it is represented by the proof term $\mathsf{bwdrule}\,(\lambda x.\,\{\mathsf{let}\,\{y\} \leftarrow \mathsf{fwdruleAB}\,x\,\mathsf{in}\,y\})$. The partially complete interpretation of backward chaining stipulates that an interpreter tasked with finding a proof of negD should either find this proof or never terminate, but the Lollimon semantics only admits this interpretation for purely deductive proofs. To see why, consider backward-chaining search attempting to prove negD in a closed context. This can only be done with the rule bwdrule, generating the subgoal $\mathsf{posA} \rightarrowtail \bigcirc\mathsf{posB}$. At this point, the Lollimon semantics will switch from backward chaining to forward chaining and attempt to satisfy this subgoal by constructing a trace $(x{:}\langle\mathsf{posA}\rangle\ ord) \rightsquigarrow (y{:}\langle\mathsf{posB}\rangle\ ord)$.

There are *two* nontrivial traces in this signature starting from the process state $(x{:}\langle\mathsf{posA}\rangle\ ord)$ – the first is $(\{y\} \leftarrow \mathsf{fwdruleAB}\,x) :: (x{:}\langle\mathsf{posA}\rangle\ ord) \rightsquigarrow (y{:}\langle\mathsf{posB}\rangle\ ord)$, and the second is $(\{y\} \leftarrow \mathsf{fwdruleAC}\,x) :: (x{:}\langle\mathsf{posA}\rangle\ ord) \rightsquigarrow (y{:}\langle\mathsf{posC}\rangle\ ord)$. Forward chaining can plausibly come up with either one, and if it happens to derive the second one, the subgoal fails. Lollimon then tries to backtrack to find other rules that can prove the conclusion negD, but there are none, so the Lollimon semantics will report a failure to prove negD.

[5] As such, "concurrent computation," while appropriate for SLS, may or may not prove to be a good name for the general paradigm.

This example indicates that it is difficult to make backward chaining (in its default backtracking form) reliant on committed-choice forward chaining (in its default committed-choice form) in the Lollimon semantics. Either we can restrict forward chaining to confluent systems (excluding Σ_{Demo}) or else we can give up on the usual partially complete interpretation of backward chaining. In the other direction, however, it is entirely natural to make forward chaining dependent upon backward chaining. The fragment of CLF that encodes this kind of computation was labeled the *semantic effects* fragment by DeYoung [DP09]. At the logical level, the semantic effects fragment of SLS removes the right rule for $\bigcirc A^+$, which corresponds to the proof term $\{\mathsf{let}\, T\, \mathsf{in}\, V\}$. As discussed in Section 4.2.6, these let-expressions are the only point where traces are included into the language of deductive terms.

4.7 Design decisions

Aside from ordered propositions, there are several significant differences between the framework SLS presented in this chapter and the existing logical framework CLF, including the presence of positive atomic propositions, the introduction of traces as an explicit notation for partial proofs, the restriction of the term language to LF, and the presence of equality $t \doteq_\tau s$ as a proposition. In this section, we will discuss design choices that were made in terms of each of these features, their effects, and what choices could have been made differently.

4.7.1 Pseudo-positive atoms

Unlike SLS, the CLF framework does not include positive atomic propositions. Positive atomic propositions make it easy to characterize the synthetic transitions associated with a particular rule. For example, if foo, bar, and baz are all linear atomic propositions, then the presence of a rule somerule : $(\mathsf{foo} \bullet \mathsf{bar} \rightarrowtail \bigcirc\mathsf{baz})$ in the signature is associated with synthetic transitions of the form $(\Psi; \Delta, x{:}\langle\mathsf{foo}\rangle\, eph, y{:}\langle\mathsf{bar}\rangle\, eph) \rightsquigarrow (\Psi; \Delta, z{:}\langle\mathsf{baz}\rangle\, eph)$. The presence of the rule somerule enables steps of this form, and every step made by focusing on the rule has this form.

CLF has no positive propositions, so the closest analogue that we can consider is where foo, bar, and baz are negative propositions, and the rule $¡\mathsf{foo} \bullet ¡\mathsf{bar} \rightarrowtail \bigcirc(¡\mathsf{baz})$ appears in the signature. Such a rule is associated with synthetic transitions of the form $(\Psi; \Delta, \Delta_1, \Delta_2) \rightsquigarrow (\Psi; \Delta, z{:}\mathsf{baz}\ ord)$ such that $\Psi; \Delta_1\uparrow_{eph} \vdash \langle\mathsf{foo}\rangle\, true$ and $\Psi; \Delta_2\uparrow_{eph} \vdash \langle\mathsf{bar}\rangle\, true$. In SLS, it is a relatively simple syntactic criterion to enforce that a sequent like $\Psi; \Delta_1 \vdash \langle\mathsf{foo}\rangle\, true$ can only be derived if Δ_1 matches $x{:}\mathsf{foo}$; we must simply ensure that there are no propositions of the form $\ldots \rightarrowtail \mathsf{foo}$ or $\ldots \twoheadrightarrow \mathsf{foo}$ in the signature or context. (In fact, this is essentially the SLS version of the subordination criteria that allows us to conclude that an LF type is only inhabited by variables in Section 4.2.) Note that, in full OL_3, this task would not be so easy: we might prove $\langle\mathsf{foo}\rangle\, true$ indirectly by forward chaining. This is one reason why association of traces with the lax modality is so important!

When it is the case that $\Psi; \Delta_1 \vdash \langle\mathsf{foo}\rangle\, true$ can only be derived if Δ_1 matches $x{:}\mathsf{foo}$, we can associate the rule $¡\mathsf{foo} \bullet ¡\mathsf{bar} \rightarrowtail \bigcirc(\downarrow(¡\mathsf{baz}))$ with a unique synthetic transition of the form $(\Psi; \Delta, x{:}\mathsf{foo}\ lvl, y{:}\mathsf{bar}\ lvl') \rightsquigarrow (\Psi; \Delta, z{:}\langle\mathsf{baz}\rangle\, eph)$ under the condition that neither lvl or lvl' are ord. Negative atomic propositions that can only be concluded when they are the sole member

of the context, like foo and bar in this example, can be called *pseudo-positive*. Pseudo-positive atoms can actually be used a bit more generally than SLS's positive atomic propositions. A positive atomic proposition is necessarily associated with one of the three judgments ord, eph, or $pers$, but pseudo-positive propositions can associate with any of the contexts. This gives pseudo-positive atoms in CLF or SLS the flavor of positive atomic propositions under Andreoli's atom optimization (Section 2.5.1).

It is, of course, possible to consistently associate particular pseudo-positive propositions with particular modalities, which means that pseudo-positive propositions can subsume the positive propositions of SLS. The trade-off between positive and pseudo-positive propositions could be resolved either way. By including positive atomic propositions, we made SLS more complicated, but in a local way – we needed a few more kinds (the kinds prop ord, prop lin, and prop pers, to be precise) and a few more rules. On the other hand, if we used pseudo-positive propositions, the notion of synthetic transitions would be intertwined with the subordination-like analysis that enforces their correct usage.

4.7.2 The need for traces

One of the most important differences between SLS and its predecessors, especially CLF, is that traces are treated as first-class syntactic objects. This allows us to talk about partial proofs and thereby encode our earlier money-store-battery-robot example as a trace with this type:

$$\begin{aligned}&(x{:}\langle \mathsf{6bucks}\rangle\ eph,\ \ f{:}(\mathsf{battery} \rightarrowtail \bigcirc\mathsf{robot})\ eph,\ \ g{:}(\mathsf{6bucks} \rightarrowtail \bigcirc\mathsf{battery})\ pers)\\ \rightsquigarrow^*\ &(z{:}\langle \mathsf{robot}\rangle\ eph,\ \ g{:}(\mathsf{6bucks} \rightarrowtail \bigcirc\mathsf{battery})\ pers)\end{aligned}$$

It is also possible to translate the example from Chapter 2 as a *complete* proof of the following proposition:

$$\mathsf{6bucks} \bullet {\text{¡}}(\mathsf{battery} \rightarrowtail \bigcirc\mathsf{robot}) \bullet !(\mathsf{6bucks} \rightarrowtail \bigcirc\mathsf{battery}) \rightarrowtail \bigcirc\mathsf{robot}$$

Generally speaking, we can try to represent a trace $T :: (\Psi; \Delta) \rightsquigarrow^* (\Psi'; \Delta')$ as a closed deductive proof $\lambda P.\{\mathsf{let}\, T\, \mathsf{in}\, V\}$ of the proposition $(\exists\Psi.\bullet\Delta) \rightarrowtail \bigcirc(\exists\Psi'.\bullet\Delta)$,[6] where the pattern P re-creates the initial process state $(\Psi; \Delta)$ and all the components of the final state are captured in the value V. The problem with this approach is that the final proposition is under no particular obligation to faithfully capture the structure of the final process state. This can be seen in the example above: to actually capture the structure of the final process state, we should have concluded $\mathsf{robot} \bullet !(\mathsf{6bucks} \rightarrowtail \bigcirc\mathsf{battery})$ instead of simply robot. It is also possible to conclude any of the following:

1. $\mathsf{robot} \bullet !(\mathsf{6bucks} \rightarrowtail \bigcirc\mathsf{battery}) \bullet !(\mathsf{6bucks} \rightarrowtail \bigcirc\mathsf{battery})$, or
2. $\mathsf{robot} \bullet \downarrow(\mathsf{6bucks} \rightarrowtail \bigcirc\mathsf{battery}) \bullet {\text{¡}}(\mathsf{6bucks} \rightarrowtail \bigcirc\mathsf{battery})$, or even
3. $\mathsf{robot} \bullet {\text{¡}}(\mathsf{6bucks} \bullet !(\mathsf{battery} \rightarrowtail \bigcirc\mathsf{robot}) \rightarrowtail \bigcirc\mathsf{robot}) \bullet \downarrow(\mathsf{robot} \rightarrowtail \bigcirc\mathsf{robot})$.

[6]The notation $\bullet\Delta$ fuses together all the propositions in the context. For example, if $\Delta = w{:}\langle p^+_{eph}\rangle\ eph \bullet x{:}A^-\ ord, y{:}B^-\ eph, z{:}C^-\ pers$, then $\bullet\Delta = p^+_{eph} \bullet \downarrow A^- \bullet {\text{¡}}B^- \bullet !C^-$. The notation $\exists\Psi.A^+$ turns all the bindings in the context $\Psi = a_1{:}\tau_1, \ldots, a_n{:}\tau_n$ into existential bindings $\exists a_1{:}\tau_1 \ldots \exists a_n{:}\tau_n.A^+$.

The problem with encoding traces as complete proofs, then, is that values cannot be forced to precisely capture the structure of contexts, especially when dealing with variables or persistent propositions. Cervesato and Scedrov approach this problem by severely restricting the logic and changing the interpretation of the existential quantifier so that it acts like a nominal quantifier on the right [CS09]. The introduction of traces allows us to avoid similar restrictions in SLS.

Despite traces being proper syntactic objects, they are not first-class concepts in the theory: they are derived from focused OL_3 terms and interpreted as partial proofs. Because hereditary substitution, identity expansion, and focalization are only defined on complete OL_3 proofs, these theorems and operations only apply by analogy to the deductive fragment of SLS; they do not apply to traces. In joint work with Deng and Cervesato, we considered a presentation of logic that treats process states and traces as first-class concepts and reformulates the usual properties of cut and identity in terms of coinductive simulation relations on process states [DCS12]. We hope that this work will eventually lead to a better understanding of traces, but the gap remains quite large.

4.7.3 LF as a term language

The decision to use LF as a first-order domain of quantification rather than using a fully-dependent system is based on several considerations. First and foremost, this choice was sufficient for our purposes here. In fact, for the purposes of this dissertation, we could have used an even simpler term language of simply-typed LF [Pfe08]. Two other logic programming interpreters for SLS-like frameworks, Lollimon [LPPW05] and Ollibot [PS09], are in fact based on simply-typed term languages. Canonical LF and Spine Form LF are, at this point, sufficiently well understood that the additional overhead of fully dependently-typed terms is not a significant burden, and there are many examples beyond the scope of this dissertation where dependent types are useful.

On a theoretical level, it is a significant simplification when we restrict ourselves to *any* typed term language with a reasonable notion of equality and simultaneous substitution. The conceptual priority in this chapter is clear: Section 4.1 describes LF terms, Section 4.2 describes proof terms as a fragment of focused OL_3, and Section 4.3 describes a coarser equivalence on proof terms, concurrent equality. If the domain of first-order of quantification was SLS terms, these three considerations would be mutually dependent – we would need to characterize concurrent equality before presenting the logic itself. For the purposes of showing that a logical framework can be carved out from a focused logic – the central thesis of this and the previous two chapters – it is easiest to break this circular dependency. We conjecture that this complication is no great obstacle, but our approach avoids the issue.

On a practical level, there are advantages to using a well-understood term language. The SLS prototype implementation (Section 4.5) uses the mature type reconstruction engine of Twelf to reconstruct LF terms. Schack-Nielsen's implementation of type reconstruction for Celf is complicated by the requirements of dealing with type reconstruction for a substructural term language, a completely orthogonal consideration [SNS08].

Finally, it is not clear that the addition of full CLF-like dependency comes with great expressive benefit. In LF and Twelf, the ability to use full dependent types is critical in part because it allows us to express *metatheorems* – theorems about the programming languages and logics we have encoded, like progress and preservation for a programming language or cut admissibility for

a logic. Substructural logical frameworks like LLF and CLF, in contrast, have not been successful in capturing metatheorems with dependent types. Instead, metatheorems about substructural logics have thus far generally been performed in logical frameworks based on persistent logics. Crary proved theorems about linear logics and languages in LF using the technique of explicit contexts [Cra10]. Reed was able to prove cut admissibility for linear logic and preservation for the LLF encoding of Mini-ML in HLF, a persistent extension to LF that uses an equational theory to capture the structure of substructural contexts [Ree09a].

Part II

Substructural operational semantics

Chapter 5

On logical correspondence

In Part I, we defined SLS, a logical framework of substructural logical specifications. For the purposes of this dissertation, we are primarily interested in using SLS as a framework for specifying the operational semantics of programming languages, especially stateful and concurrent programming languages. This is not a new idea: one of the original case studies on CLF specification described the semantics of Concurrent ML [CPWW02] in a specification style termed *substructural operational semantics*, or SSOS, by Pfenning [Pfe04].

The design space of substructural operational semantics is extremely rich, and many styles of SSOS specification have been proposed previously. It is therefore helpful to have design principles that allow us to both *classify* different styles of presentation and *predict* what style(s) we should adopt based on what our goals are. In this chapter, we sketch out a classification scheme for substructural operational semantics based on three major specification styles:

* The *natural semantics*, or big-step operational semantics, is an existing and well-known specification style (and not a substructural operational semantics). It is convenient for the specification of pure programming languages.
* The *ordered abstract machine semantics* is a generalization of abstract machine semantics that can be naturally specified in SLS; this specification style naturally handles stateful and parallel programming language features [PS09].
* The *destination-passing semantics* is the style of substructural operational semantics first explored in CLF by Cervesato et al. [CPWW02]. It allows for the natural specification of features that incorporate communication and non-local transfer of control.

Each of these three styles is, in a formal sense, more expressive than the last: there are automatic and provably-correct transformations from the less expressive styles (natural semantics and ordered abstract machines) to the more expressive styles (ordered abstract machines and destination-passing, respectively). Our investigation of provably-correct transformations on SLS specifications therefore justifies our classification scheme for SSOS specifications. We call this idea the *logical correspondence*, and it is the focus of this refinement of our central thesis:

> **Thesis (Part II):** *A logical framework based on a rewriting interpretation of substructural logic supports many styles of programming language specification. These styles can be formally classified and connected by considering general transformations on logical specifications.*

In this introductory chapter, we will outline our use of logical correspondence and connect it to previous work. The development of the logical correspondence as presented in this chapter, as well as the operationalization and defunctionalization transformations presented in the next chapter, represent joint work with Ian Zerny.

5.1 Logical correspondence

As stated above, we will primarily discuss and connect three different styles that are used for specifying the semantics of programming languages. The two styles of SSOS semantics, ordered abstract machines and destination-passing semantics, are considered because they do a good job of subsuming existing work on substructural operational semantics, a point we will return to at the end of this section. We consider natural semantics, a high-level, declarative style of specification that was inspired by Plotkin's structural operational semantics (SOS) [Plo04, Kah87], because natural semantics specifications are the easiest style to connect to substructural operational semantics. While we hope to extend the logical correspondence to other specification styles, such extensions are outside the scope of this dissertation.

While Kahn et al. defined the term broadly, natural semantics has been consistently connected with the big-step operational semantics style discussed in the introduction, where the judgment $e \Downarrow v$ expresses that the the expression e evaluates to the value v:

$$\frac{}{\lambda x.e \Downarrow \lambda x.e}\ \mathsf{ev/lam} \qquad \frac{e_1 \Downarrow \lambda x.e \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{e_1\, e_2 \Downarrow v}\ \mathsf{ev/app}$$

Early work on natural semantics emphasized a dual interpretation of specifications. The primary interpretation of natural semantics specifications was *operational*. Natural semantics were implemented in the (non-logical) specification framework TYPOL that compiled natural semantics specifications into Prolog programs; the backward-chaining Prolog interpreter then gave an operational semantics to the specification [CDD+85]. It is also possible to view natural semantics specifications as inductive definitions; this interpretation allows proofs about terminating evaluations to be performed by induction over the structure of a natural semantics derivation [CDDK86].

The operational interpretation of natural semantics assigns a more specific meaning to expressions than the inductive definition does. For example, the rule ev/app as an inductive definition does not specify whether e_1 or e_2 should be evaluated in some particular order or in parallel; the TYPOL-to-Prolog compiler could have reasonably made several choices in such a situation. More fundamentally, the logic programming interpretation inserts semantic information into a natural semantics specification that is not present when we view the specification as an inductive definition (though it might be just as accurate to say that the logic programming interpretation preserves meaning that is lost when the specification is viewed as an inductive definition). The interpretation of the rules above as an inductive definition does not allow us to distinguish nontermination (searching forever for a v such that $e \Downarrow v$) from failure (concluding finitely that there is no v such that $e \Downarrow v$). The logic programming interpreter, on the other hand, will either succeed, run forever, or give up, thereby distinguishing two cases that are indistinguishable when the specification is interpreted as an inductive definition.

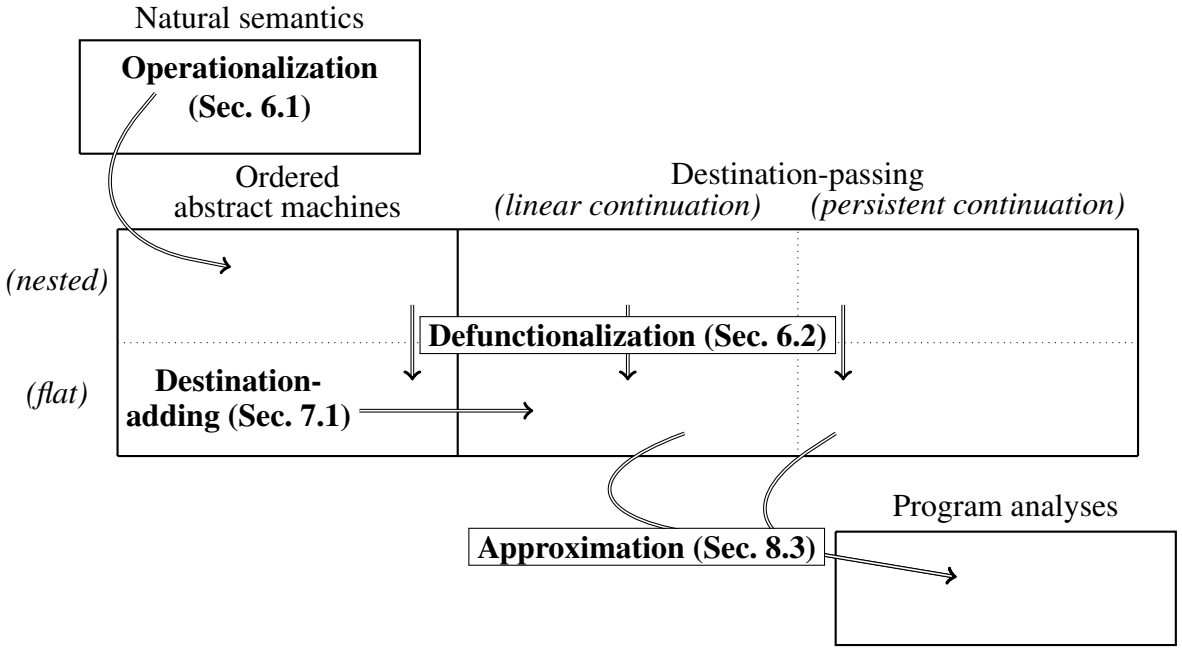


Figure 5.1: Major transformations on SLS specifications

We will present a transformation called *operationalization* from SLS-encoded natural semantics specifications into ordered abstract machines. The transformation from natural semantics to ordered abstract machines is only an instance of a much more general picture. The basic idea of operationalization is to model backward-chaining logic programming (in the sense of Section 4.6.1) as forward-chaining logic programming (in the sense of Section 4.6.2). The transformation reifies and exposes the internal structure of backward-chaining search, making evaluation order and parallelism explicit. That exposed structure enables us to reason about the difference between non-termination and failure. In turn, ordered abstract machine specifications can be transformed into destination-passing specifications by a transformation called *destination-adding*, which reifies and exposes control flow information that is implicit in the ordered context of an ordered abstract machine. Destination-passing specifications can then be transformed into a collecting semantics by *approximation*, which lets us obtain program analyses like control flow analysis. The operationalization and destination-adding transformations have been implemented within the SLS prototype. Approximation, on the other hand, requires significant input from the user and so is less reasonable to implement as an automatic transformation.

These major transformations are presented graphically in Figure 5.1 in terms of the three classification styles – natural semantics, ordered abstract machines, and destination-passing – discussed above. There are many other smaller design decisions that can be made in the creation of a substructural operational semantics, two of which are represented in this figure. One distinction, destination-passing with linear continuations versus persistent continuations, has to do with whether it is possible to return to a previous point in a program's execution and is discussed, along with first-class continuations, in Section 7.2.4.

Another distinction is between *nested* and *flat* specifications. This distinction applies to all

$$
\begin{aligned}
x_1{:}\langle \mathsf{p}_2(\mathsf{c})\rangle\ ord,\ x_2{:}\langle \mathsf{p}_1(\mathsf{c})\rangle\ ord,\ x_3{:}(\forall x.\, \mathsf{p}_1(x) \rightarrowtail \{\mathsf{p}_2(x) \rightarrowtail \{\mathsf{p}_3(x)\}\})\ ord,\ x_4{:}(\mathsf{p}_3(\mathsf{c}) \rightarrowtail \{\mathsf{p}_4\})\ ord \\
\leadsto\ x_1{:}\langle \mathsf{p}_2(\mathsf{c})\rangle\ ord,\ x_5{:}(\mathsf{p}_2(\mathsf{c}) \rightarrowtail \{\mathsf{p}_3(\mathsf{c})\})\ ord,\ x_4{:}(\mathsf{p}_3(\mathsf{c}) \rightarrowtail \{\mathsf{p}_4\})\ ord \\
\leadsto\ x_6{:}\langle \mathsf{p}_3(\mathsf{c})\rangle\ ord,\ x_4{:}(\mathsf{p}_3(\mathsf{c}) \rightarrowtail \{\mathsf{p}_4\})\ ord \\
\leadsto\ x_7{:}\langle \mathsf{p}_4\rangle\ ord
\end{aligned}
$$

Figure 5.2: Evolution of a nested SLS process state

concurrent SLS specifications, not just those that specify substructural operational semantics. Flat specifications include rewriting rules $(p_1 \bullet \ldots \bullet p_n \rightarrowtail \{q_1 \bullet \ldots \bullet q_m\})$ where the head of the rule $\{q_1 \bullet \ldots \bullet q_m\}$ contains only atomic propositions. Nested SLS specifications, on the other hand, may contain *rules* in the conclusions of rules; when the rule fires, the resulting process state contains the rule. A rule $A^+ \rightarrowtail \{B^+\}$ in the context can only fire if a piece of the context matching A^+ appears to its left, so $(x{:}\langle \mathsf{p}_1(\mathsf{c})\rangle,\ y{:}(\mathsf{p}_1(\mathsf{c}) \rightarrowtail \{\mathsf{p}_2(\mathsf{c})\})\ ord) \leadsto (z{:}\langle \mathsf{p}_2(\mathsf{c})\rangle)$, whereas $(y{:}(\mathsf{p}_1(\mathsf{c}) \rightarrowtail \{\mathsf{p}_2(\mathsf{c})\})\ ord,\ x{:}\langle \mathsf{p}_1(\mathsf{c})\rangle) \not\leadsto$. Another example of the evolution of a process state with nested rules is given in Figure 5.2. (Appendix A gives a summary of the notation used for process states.) The choice of nested versus flat specification does not impact expressiveness, but it does influence our ability to read specifications (opinions differ as to which style is clearer), as well as our ability to reason about specifications. The methodology of describing the invariants of substructural logical specifications with *generative signatures*, which we introduced in Section 4.4 and which we will consider further in Chapter 9, seems better-adapted to describing the invariants of flat specifications.

Other distinctions between SSOS specifications can be understood in terms of nondeterministic choices that can be made by the various transformations we consider. For example, the operationalization transformation can produce ordered abstract machines that evaluate subcomputations in parallel or in sequence. In general, one source specification (a natural semantics or an ordered abstract machine specification) can give rise to several different target specifications (ordered abstract machine specifications or destination-passing specifications). The correctness of the transformation then acts as a simple proof of the equivalence of the several target specifications. (The prototype implementations of these transformations only do one thing, but the nondeterministic transformations we prove correct would justify giving the user a set of additional controls – for instance, the user could make the operationalization transformation be tail-call-optimizing or not and parallelism-enabling or not.)

The nondeterministic choices that transformations can make give us a rigorous vocabulary for describing choices that otherwise seem unmotivated. An example of this can be found in the paper that introduced the destination-adding and approximation transformations [SP11a]. In that article, we had to motivate an ad hoc change to the usual abstract machine semantics. In this dissertation, by the time we encounter a similar specification in Chapter 8, we will be able to see that this change corresponds to omitting tail-recursion optimization in the process of operationalization.

Our taxonomy does a good job of capturing the scope of existing work on SSOS specifications. Figure 5.3 shows previous published work on SSOS specifications mapped onto a version of the diagram from Figure 5.1. With the possible exception of certain aspects of the SSOS pre-

sentation in Pfenning's course notes [Pfe12e], the taxonomy described above captures the scope of previous work.

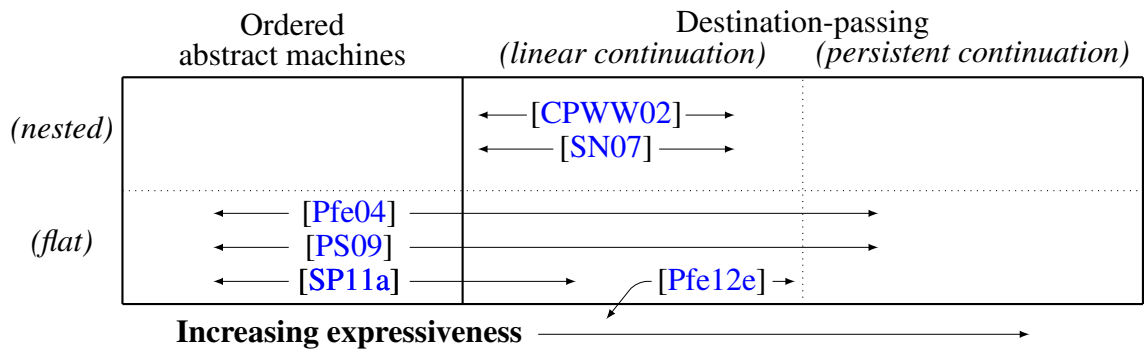


Figure 5.3: Classification of existing work on SSOS specifications

5.2 Related work

This part of the dissertation draws from many different sources of inspiration. In this section, we survey this related work and, where applicable, outline how our use of logical correspondence differs from existing work.

Partiality in deductive computation

The genesis of the operationalization transformation discussed in Chapter 6 can be found in the treatment of the operational semantics of LF in Tom Murphy VII's dissertation [Mur08]; this treatment can be seen as a synthesis of the operational interpretation of natural semantics explored in Clément's et al.'s early work on natural semantics in TYPOL and the approach to theorem proving pioneered by Twelf [PS99b].

In his dissertation, Murphy described a natural semantics for Lambda 5, a distributed programming language, and encoded that specification in Twelf. He then wanted to interpret that natural semantics as an *operational* semantics for Lambda 5 in the style of Clément et al., which is a natural application of Twelf's logic program interpretation [MP92]. However, Murphy also wanted to prove a safety property for his language in Twelf, and the usual approach to theorem proving in Twelf involves treating specifications as inductive definitions. As discussed above, natural semantics do not distinguish non-termination (which is safe) from failure (which indicates underspecification and is therefore unsafe).

Theorem proving in Twelf involves interpreting proofs as backward chaining logic programs that do not backtrack (recall that we called this the *flat resolution* interpretation in Section 4.6.1). Murphy was able to use the checks Twelf performs on proofs to describe a special purpose partiality directive. If a logic program passed his series of checks, Murphy could conclude that well-moded, flat resolution would never fail and never backtrack, though it might diverge. This check amounted to a proof of safety (progress and preservation) for the operational interpretation

of his natural semantics via flat resolution. It seems that every other existing proof of safety[1] for a big-step operational semantics is either classical (like Leroy and Grall's approach, described below) or else depends on a separate proof of equivalence with a small-step operational semantics.

Murphy's proof only works because his formulation of Lambda 5 is *intrinsically typed*, meaning that, using the facilities provided by LF's dependent types, he enforced that only well-typed terms could possibly be evaluated. (His general proof technique should apply more generally, but it would take much more work to express the check in Twelf.) The operationalization transformation is a way to automatically derive a correct small-step semantics from the big-step semantics by making the internal structure of a backward chaining computation explicit as a specification in the concurrent fragment of SLS. Having made this structure accessible, we can explicitly represent complete, unfinished, and stuck (or failing) computations as concurrent traces and reason about these traces with a richer set of tools than the limited set Murphy successfully utilized.

A coinductive interpretation

Murphy proved safety for a natural semantics specification by recovering the original operational interpretation of natural semantics specifications as logic programs and then using Twelf's facilities for reasoning about logic programs. Leroy and Grall, in [LG09], suggest a novel *coinductive* interpretation of natural semantics specifications. Coevaluation $e \Downarrow^{\mathsf{co}} v$ is defined as the *greatest* fixed point of the following rules:

$$\frac{}{\lambda x.e \Downarrow^{\mathsf{co}} \lambda x.e} \qquad \frac{e_1 \Downarrow^{\mathsf{co}} \lambda x.e \quad e_2 \Downarrow^{\mathsf{co}} v_2 \quad [v_2/x]e \Downarrow^{\mathsf{co}} v}{e_1\, e_2 \Downarrow^{\mathsf{co}} v}$$

Aside from the co annotation and the different interpretation, these rules are syntactically identical to the natural semantics above that were implicitly given an inductive interpretation.

Directly reinterpreting the inductive specification as a coinductive specification doesn't quite produce the right result in the end. For some diverging terms like $\omega = (\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$, we can derive $\omega \Downarrow^{\mathsf{co}} e$ for any expression e, including expressions that are not values and expressions with no relation to the original term. Conversely, there are diverging terms Div such that $Div \Downarrow^{\mathsf{co}} e$ is not derivable for *any* e.[2] As a result, Leroy and Grall also give a coinductive definition of diverging terms $e \Downarrow^{\infty}$ that references the inductively-defined evaluation judgment $e \Downarrow v$:

$$\frac{e_1 \Downarrow^{\infty}}{e_1\, e_2 \Downarrow^{\infty}} \qquad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow^{\infty}}{e_1\, e_2 \Downarrow^{\infty}} \qquad \frac{e_1 \Downarrow \lambda x.\, e \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow^{\infty}}{e_1\, e_2 \Downarrow^{\infty}}$$

Now diverging expressions are fully characterized as derivations for which $e \Downarrow^{\infty}$ is derivable with an infinite derivation tree. With this definition, Leroy and Grall prove a type safety property: if e has type τ, then either $e \Downarrow v$ or $e \Downarrow^{\infty}$. However, the disjunctive character of this theorem means that a constructive proof of type safety would be required to take a typing derivation $e : \tau$ as

[1] Progress in particular is the theorem of concern: proving preservation for a big-step operational semantics is straightforward.

[2] Leroy and Grall discuss a counterexample due to Filinski: $Div = YFx$, where Y is the fixed-point combinator $\lambda f.\, (\lambda x.\, f\, (\lambda v.\, (x\, x)\, v))\, (\lambda x.\, f\, (\lambda v.\, (x\, x)\, v))$ and F is $\lambda f.\, \lambda x.\, (\lambda g.\, \lambda y.\, g\, y)\, (f\, x)$ [LG09].

input and produce as output either a proof of termination $e \Downarrow v$ or a proof of divergence $e \Downarrow^{\infty}$. This implies that a constructive type safety theorem would need to decide termination, and so it is unsurprising that type safety is proved classically by Leroy and Grall.

We suggest that the operationalization transformation, seen as a logical extension to Murphy's methodology, is superior to the coinductive (re)interpretation as a way of understanding the behavior of diverging evaluations in the natural semantics. Both approaches reinterpret natural semantics in an operational way, but the operationalization transformation gives us a satisfactory treatment of diverging terms without requiring the definition of an additional coinductive judgment $e \Downarrow^{\infty}$. And even *with* the addition of the coinductively defined judgment $e \Downarrow^{\infty}$, coinductive big-step operational semantics have significant issues handling nondeterministic languages, a point that we will elaborate on in Section 6.4.

The functional correspondence

The ordered abstract machine that results from our operationalization transformation corresponds to a standard abstract machine model (a statement that is made precise Section 6.3). In this sense, the logical correspondence has a great deal in common with the *functional correspondence* of Ager, Danvy, Midtgaard, and others [ABDM03, ADM04, ADM05, Dan08, DMMZ12].

The goal of the functional correspondence is to encode various styles of semantic specifications (natural semantics, abstract machines, small-step structural operational semantics, environment semantics, etc.) as functional programs. It is then possible to show that these styles can be related by off-the-shelf and fully correct transformations on functional programs. The largest essential difference between the functional and logical correspondences, then, is that the functional correspondence acts on functional programs, whereas the logical correspondence acts on specifications encoded in a logical framework (in our case, the logical framework SLS).

The functional correspondence as given assumes that semantic specifications are adequately represented as functional programs; the equivalence of the encoding and the "on paper" semantics is an assumed prerequisite. In contrast, by basing the logical correspondence upon the SLS framework, we make it possible to reason formally and precisely about adequate representation by the methodology outlined in Section 4.4. The functional correspondence also shares some of the coinductive reinterpretation's difficulties in dealing with nondeterministic and parallel execution. The tools we can use to express the semantics are heavily influenced by the semantics of the host programming language, and so the specifics of the host language can make it dramatically more or less convenient to encode nondeterministic or parallel programming language features.

Transformation on specifications

Two papers by Hannan and Miller [HM92] and Ager [Age04] are the most closely related to our operationalization transformation. Both papers propose operationalizing natural semantics specifications as abstract machines by provably correct and general transformations on logical specifications (in the case of Hannan and Miller) or on specifications in the special-purpose framework of L-attributed natural semantics (in the case of Ager). A major difference in this

case is that both lines of work result in *deductive* specifications of abstract machines. Our translation into the concurrent fragment of SLS has the advantage of exploiting parallelism, and also opens up specifications to the modular inclusion of stateful and concurrent features, as we will foreshadow in Section 5.3 below and discuss further in Section 6.5.

The transformation we call defunctionalization in Section 6.2, as well as its inverse, refunctionalization, makes appearances throughout the literature under various names. The transformation is not strictly analogous to Reynold's defunctionalization transformations on functional programs [Rey72], but it is based upon the same idea: we take an independently transitioning object like a function (or, in our case, a negative proposition in the process state) and turn it into data and an application function. In our case, the data is a positive atomic proposition in the process state and the application function is a rule in the signature that explains how the positive atomic proposition can participate in transitions. The role of defunctionalization within our work on the logical correspondence is very similar to the role of Reynold's defunctionalization within work on the functional correspondence [Dan08]. Defunctionalization is related to the process of representing a process calculus object in the chemical abstract machine [BB90]. It is also related to a transformation discussed by Miller in [Mil02] in which new propositions are introduced and existentially quantified locally in order to hide the internal states of processes.

The destination-adding transformation described in Section 7.1 closely follows the contours of work by Morrill, Moot, and Piazza on translating ordered logic into linear logic [Mor95, MP01]. That work is, in turn, based on van Benthem's relational models of ordered logic [vB91]. Their transformations handle a more uniform logical fragment, whereas the transformation we describe handles a specific (though useful) fragment of the much richer logic of SLS propositions.

Related work for program analysis methodology covered in Chapter 8 is discussed further in Section 8.6.

Abstract machines in substructural logic

With the exception of our encodings of natural semantics, all our work on the logical correspondence takes place in the concurrent (rewriting-like) fragment of SLS. This is consistent with the tradition of substructural operational semantics, but there is another tradition of encoding abstract machines in substructural logical frameworks using frameworks that can be seen as *deductive* fragments of SLS. The resulting logical specifications are functionally similar to the big-step abstract machine specifications derived by Hannan and Miller, but like SSOS specifications they can take advantage of the substructural context for the purpose of modular extension (as discussed in the next section).

This line of work dates back to Cervesato and Pfenning's formalization of Mini-ML with references in Linear LF [CP02]; a mechanized preservation property for this specification was given by Reed [Ree09a]. An extension to this technique, which uses Polakow's Ordered LF to represent control stacks, is presented by Felty and Momigliano and used to mechanize a preservation property [FM12]. Both these styles of deductive big-step specification are useful for creating language specifications that can be modularly extended with stateful and control features, but neither does a good job with modular specification of concurrent or parallel features.

Both of these specifications should be seen as different points in a larger story of logical

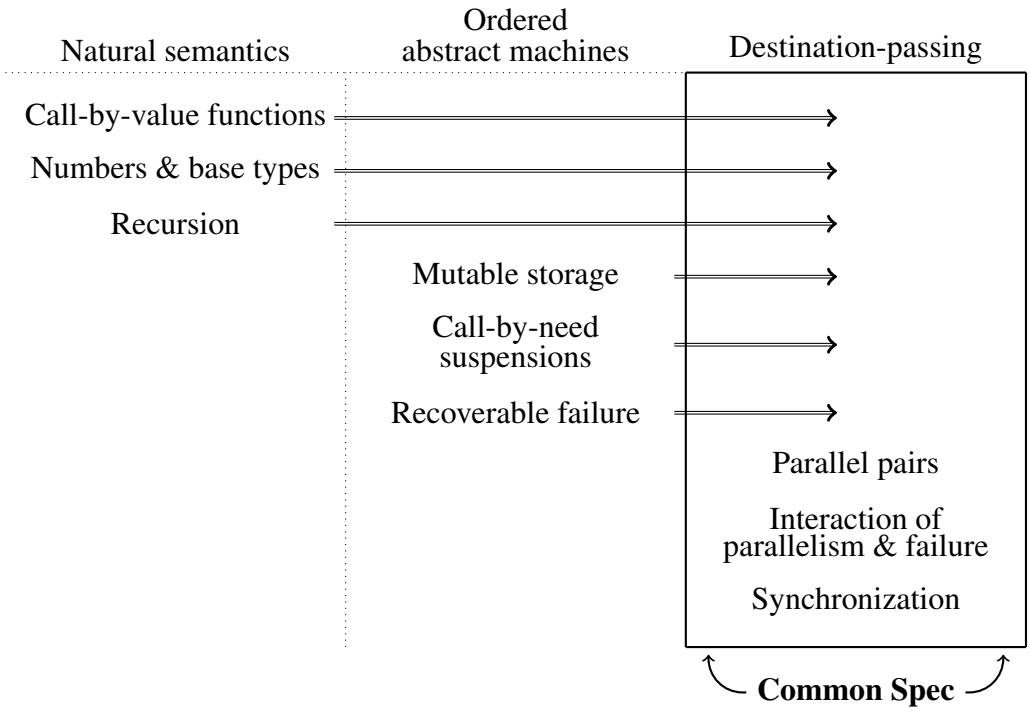


Figure 5.4: Using the logical correspondence for modular language extension

correspondence that we are only beginning to explore in this dissertation. The use of the ordered context in Felty and Momigliano's specification, in particular, is exactly analogous to the non-parallel ordered abstract machines in Chapter 6. We therefore posit the existence of a general transformation, similar to operationalization, that connects the two.

5.3 Transformation and modular extension

All the related work described in the previous section is concerned with *correspondence*. That is, the authors were interested in the process of transforming natural semantics into abstract machines and in the study of abstract machines that are in the image of this translation. It is possible to view the logical correspondence in the same light, but that is not how logical correspondence will be used in this document. Indeed, it is not our intent to advocate strongly for the use of natural semantics specifications at all; recall that natural semantics were used to illustrate problems with *non*-modularity in language specification in Section 1.2.

Instead, we will view the transformations illustrated as arrows in Figure 5.1 in an expressly directed fashion, operationalizing natural semantics as ordered abstract machines and transforming ordered abstract machines into destination-passing semantics without giving too much thought to the opposite direction. In the context of this dissertation, the reason that transformations are important is that they expose more of the semantics to manipulation and modular extension. The operationalization transformation in Chapter 7 exposes the order of evaluation, and the SLS framework then makes it possible to modularly extend the language with stateful features:

this is exactly what we demonstrated in Section 1.2 and will demonstrate again in Section 6.5. The destination-adding transformation exposes the control structure of programs; this makes it possible to discuss first-class continuations as well as the interaction of parallelism and failure (though not necessarily at the same time, as discussed in Section 7.2.4). The control structure exposed by the destination-adding transformation is the basis of the control flow analysis in Chapter 8.

In the next three chapters that make up Part II of this dissertation, we will present natural semantics specifications and substructural operational semantics specifications in a number of styles. We do so with the confidence that these specifications can be automatically transformed into the "lowest common denominator" of flat destination-passing specifications. Certainly, this means that we should be unconcerned about using a higher-level style such as the ordered abstract machine semantics, or even natural semantics, when that seems appropriate. If we need the richer structure of destination-passing semantics later on, the specification can be automatically transformed. Using the original, high-level specifications, the composition of different language features may appear to be a tedious and error-prone process of revision, but after transformation into the lowest-common-denominator specification style, composition can be performed by simply concatenating the specifications.

Taking this idea to its logical conclusion, Appendix B presents the hybrid operational semantics specification mapped out in Figure 5.4. Individual language features are specified at the highest-level specification style that is reasonable and then automatically transformed into a single compatible specification by the transformations implemented in the SLS prototype. In such a specification, a change to a high-level feature (turning call-by-value functions to call-by-name functions, for instance) can be made at the level of natural semantics and then propagated by transformation into the common (destination-passing style) specification.

Chapter 6

Ordered abstract machines

This chapter centers around two transformations on logical specifications. Taken together, the operationalization transformation (Section 6.1) and the defunctionalization transformation (Section 6.2) allow us to establish the logical correspondence between the deductive SLS specification of a natural semantics and the concurrent SLS specification of an abstract machine.

Natural semantics specifications are common in the literature, and are also easy to encode in either the deductive fragment of SLS or in a purely deductive logical framework like LF. We will continue to use the natural semantics specification of call-by-value evaluation for the lambda calculus as a running example:

$$\frac{}{\lambda x.e \Downarrow \lambda x.e}\ \mathsf{ev/lam} \qquad \frac{e_1 \Downarrow \lambda x.e \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{e_1\, e_2 \Downarrow v}\ \mathsf{ev/app}$$

Natural semantics are a *big-step* semantics: the judgment $e \Downarrow v$ describes the relationship between an initial expression e and the value v to which it will eventually evaluate.

The alternative to a big-step semantics is a *small-step* semantics, which describes the relationship between one intermediate state of a computation and another intermediate state after a single transition. One form of small-step semantics is a structural operational semantics (SOS) specification [Plo04]. The SOS specification of call-by-value evaluation for the lambda calculus is specified in terms of two judgments: $v\ \mathsf{value}$, which expresses that v is a value that is not expected to make any more transitions, and $e_1 \mapsto e_2$, which expresses that e_1 transitions to e_2 by reducing a β-redex.

$$\frac{}{\lambda x.e\ \mathsf{value}} \qquad \frac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2} \qquad \frac{e_1\ \mathsf{value} \quad e_2 \mapsto e_2'}{e_1\, e_2 \mapsto e_1\, e_2'} \qquad \frac{v\ \mathsf{value}}{(\lambda x.e)v \mapsto [v/x]e}$$

Abstract machine semantics are another important small-step semantics style. The most well-known abstract semantics is almost certainly Landin's SECD machine [Lan64], though our abstract machine presentation below is much closer to Danvy's SC machine from [Dan03] and Harper's $\mathcal{K}\{\mathsf{nat}{\rightharpoonup}\}$ system from [Har12, Chapter 27]. This abstract machine semantics is defined in terms of states s. The state $s = k \rhd e$ represents the expression e being evaluated on top of the stack k, and the state $s = k \lhd v$ represents the value v being returned to the stack k. Stacks k are have the form $((\ldots(\mathsf{halt}; f_1); \ldots); f_n)$ – they are left-associative sequences of

frames f terminated by halt, where $\mathsf{halt} \rhd e$ is the initial state in the evaluation of e and $\mathsf{halt} \lhd v$ is a final state that has completed evaluating to a value v. Each frame f either has the form $\Box\, e_2$ (an application frame waiting for an evaluated function to be returned to it) or the form $(\lambda x.e)\, \Box$ (an application frame with an evaluated function waiting for an evaluated value to be returned to it). Given states, stacks, and frames, we can define a "classical" abstract machine for call-by-value evaluation of the lambda calculus as a transition system with four transition rules:

$$
\begin{array}{rl}
\mathsf{absmachine/lam}: & k \rhd \lambda x.e \;\mapsto\; k \lhd \lambda x.e \\
\mathsf{absmachine/app}: & k \rhd e_1\, e_2 \;\mapsto\; (k; \Box\, e_2) \rhd e_1 \\
\mathsf{absmachine/app1}: & (k; \Box\, e_2) \lhd \lambda x.e \;\mapsto\; (k; (\lambda x.e)\, \Box) \rhd e_2 \\
\mathsf{absmachine/app2}: & (k; (\lambda x.e)\, \Box) \lhd v_2 \;\mapsto\; k \rhd [v_2/x]e
\end{array}
$$

The operational intuition for these rules is precisely the same as the operational intuition for the rewriting rules given in Section 1.2. This is not coincidental: the SLS specification from the introduction adequately encodes the transition system $s \mapsto s'$ defined above, a point that we will make precise in Section 6.3. The SLS specification from the introduction is *also* the result of applying the operationalization and defunctionalization transformations to the SLS encoding of the natural semantics given above. Therefore, these two transformations combined with the adequacy arguments at either end constitute a logical correspondence between natural semantics and abstract machines.

As discussed in Section 5.3, it is interesting to put existing specification styles into logical correspondence, but that is not our main reason for investigating logical correspondence in the context of this thesis. Rather, we are primarily interested in exploring the set of programming language features that can be modularly integrated into a transformed SLS specification and that could *not* be integrated into a natural semantics specification in a modular fashion. In Section 6.5 we explore a selection of these features, including mutable storage, call-by-need evaluation, and recoverable failure.

6.1 Logical transformation: operationalization

The intuition behind operationalization is rather simple: we examine the structure of backward chaining and then specify that computational process as an SLS specification. Before presenting the general transformation, we will motivate this transformation using our natural semantics specification of call-by-value evaluation.

The definition of $e \Downarrow v$ is moded with e as an input and v as an output, so it is meaningful to talk about being given e and using deductive computation to search for a v such that $e \Downarrow v$ is derivable. Consider a recursive search procedure implementing this particular deductive computation:

* If $e = \lambda x.e'$, it is possible to derive $\lambda x.e' \Downarrow \lambda x.e'$ with the rule $\mathsf{ev/lam}$.
* If $e = e_1\, e_2$, attempt to derive $e_1\, e_2 \Downarrow v$ using the rule $\mathsf{ev/app}$ by doing the following:
    1. Search for a v_1 such that $e_1 \Downarrow v_1$ is derivable.
    2. Assess whether $v_1 = \lambda x.e'$ for some e'; fail if it is not.

3. Search for a v_2 such that $e_2 \Downarrow v_2$ is derivable.
4. Compute $e' = [v_2/x]e$
5. Search for a v such that $e' \Downarrow v$ is derivable.

The goal of the operationalization transformation is to represent this deductive computation as a specification in the concurrent fragment of SLS. (To be sure, "concurrent" will seem like a strange word to use at first, as the specifications we write in the concurrent fragment of SLS will be completely sequential until Section 6.1.4.) The first step in this process, representing the syntax of expressions as LF terms of type exp, was discussed in Section 4.1.4. The second step is to introduce two new ordered atomic propositions. The proposition $\mathsf{eval}\ulcorner e \urcorner$ is the starting point, indicating that we want to search for a v such that $e \Downarrow v$, and the proposition $\mathsf{retn}\ulcorner v \urcorner$ indicates the successful completion of this procedure. Therefore, searching for a v such that $e \Downarrow v$ is derivable will be analogous to building a trace $T :: x_e{:}\langle \mathsf{eval}\ulcorner e \urcorner\rangle \rightsquigarrow^* x_v{:}\langle \mathsf{retn}\ulcorner v \urcorner\rangle$.

Representing the first case is straightforward: if we are evaluating $\lambda x.e$, then we have succeeded and can return $\lambda x.e$. This is encoded as the following proposition:

$$\forall E.\, \mathsf{eval}\,(\mathsf{lam}\,\lambda x.\, E\,x) \rightarrowtail \{\mathsf{retn}\,(\mathsf{lam}\,\lambda x.\, E\,x)\}$$

The natural deduction rule ev/app involves both recursion and multiple subgoals. The five steps in our informal search procedure are turned into three phases in SLS, corresponding to the three recursive calls to the search procedure – steps 1 and 2 are combined, as are steps 4 and 5. Whenever we make a recursive call to the search procedure, we leave a negative ordered proposition $A^-\ ord$ in the context that awaits the return of a proposition $\mathsf{retn}\ulcorner v' \urcorner$ to its left and then continues with the search procedure. Thus, each of the recursive calls to the search procedure will involve a sub-trace of the form

$$x_e{:}\langle \mathsf{eval}\ulcorner e' \urcorner\rangle, y{:}A^-\ ord, \Delta \rightsquigarrow^* x_v{:}\langle \mathsf{retn}\ulcorner v' \urcorner\rangle, y{:}A^-\ ord, \Delta$$

where A^- is a negative proposition that is prepared to interact with the subgoal's final $\mathsf{retn}\ulcorner v' \urcorner$ proposition to kickstart the rest of the computation. This negative proposition is, in effect, the calling procedure's continuation.

The nested rule for evaluating $e_1\ e_2$ to a value is the following proposition, where the three phases are indicated with dashed boxes:

$$\begin{array}{l}
\forall E_1.\, \forall E_2.\, \mathsf{eval}\,(\mathsf{app}\,E_1\,E_2) \\
\quad \rightarrowtail \{\mathsf{eval}\,E_1\ \bullet \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \longleftarrow \mathit{Step}_{1,2}(E_1, E_2) \\
\qquad \downarrow(\forall E.\, \mathsf{retn}\,(\mathsf{lam}\,\lambda x.\, E\,x) \\
\qquad\quad \rightarrowtail \{\mathsf{eval}\,E_2\ \bullet \qquad\qquad\qquad\qquad\qquad\qquad \longleftarrow \mathit{Step}_3(E_2, E) \\
\qquad\qquad \downarrow(\forall V_2.\, \mathsf{retn}\,V_2 \\
\qquad\qquad\quad \rightarrowtail \{\mathsf{eval}\,(E\,V_2)\ \bullet \qquad\qquad \longleftarrow \mathit{Step}_{4,5}(E, V_2) \\
\qquad\qquad\qquad \downarrow(\forall V.\, \mathsf{retn}\,V \rightarrowtail \{\mathsf{retn}\,V\})\}\})\}\}
\end{array}$$

Let's work backwards through this three-phase protocol. In the third phase, which corresponds to the fourth and fifth steps of our informal search procedure, we have found $\lambda x.\, E\,x = \lambda x.\ulcorner e \urcorner$

(where e potentially has x free) and $V_2 = \ulcorner v_2 \urcorner$. The recursive call is to $\mathsf{eval}\,\ulcorner [v_2/x]e \urcorner$, which is the same thing as $\mathsf{eval}\,(E\,V_2)$. If the recursive call successfully returns, the context will contain a suspended atomic proposition of the form $\mathsf{retn}\,V$ where $V = \ulcorner v \urcorner$, and the search procedure as a whole has been completed: the answer is v. Thus, the negative proposition that implements the continuation can be written as $(\forall V.\,\mathsf{retn}\,V \rightarrowtail \{\mathsf{retn}\,V\})$. (This continuation is the identity; we will show how to omit it when we discuss tail-recursion elimination in Section 6.1.3.) The positive proposition that will create this sub-computation can be written as follows:

$$\mathit{Step}_{4,5}(E, V_2) \equiv \mathsf{eval}\,(E\,V_2) \bullet \downarrow(\forall V.\,\mathsf{retn}\,V \rightarrowtail \{\mathsf{retn}\,V\})$$

Moving backwards, in the second phase (step 3 of the 5-step procedure) we have an expression $E_2 = \ulcorner e_2 \urcorner$ that we were given and $\lambda x.\,E\,x = \lambda x.\ulcorner e \urcorner$ that we have computed. The recursive call is to $\mathsf{eval}\,\ulcorner e_2 \urcorner$, and assuming that it completes, we need to begin the fourth step. The positive proposition that will create this sub-computation can be written as follows:

$$\mathit{Step}_3(E_2, E) \equiv \mathsf{eval}\,E_2 \bullet \downarrow(\forall V_2.\,\mathsf{retn}\,V_2 \rightarrowtail \{\mathit{Step}_{4,5}(E, V_2)\})$$

Finally, the first two steps, like the fourth and fifth steps, are handled together. We have $E_1 = \ulcorner e_1 \urcorner$ and $E_2 = \ulcorner e_2 \urcorner$; the recursive call is to $\mathsf{eval}\,\ulcorner e_1 \urcorner$. Once the recursive call completes, we enforce that the returned value has the form $\ulcorner \lambda x.e \urcorner$ before proceeding to the continuation.

$$\mathit{Step}_{1,2}(E_1, E_2) \equiv \mathsf{eval}\,E_1 \bullet \downarrow(\forall E.\,\mathsf{retn}\,(\mathsf{lam}\,\lambda x.\,E\,x) \rightarrowtail \{\mathit{Step}_3(E_2, E)\})$$

Thus, the rule implementing this entire portion of the search procedure is

$$\forall E_1.\,\forall E_2.\,\mathsf{eval}\,(\mathsf{app}\,E_1\,E_2) \rightarrowtail \{\mathit{Step}_{1,2}(E_1, E_2)\}$$

The SLS encoding of our example natural semantics is shown in Figure 6.1 alongside the transformed specification, which has the form of an ordered abstract machine semantics, though it is different than the ordered abstract machine semantics presented in the introduction. The specification in Figure 6.1 is *nested*, as ev/app is a rule that, when it participates in a transition, produces a new rule $(\forall E.\,\mathsf{retn}\,(\mathsf{lam}\,\lambda x.\,E\,x) \rightarrowtail \{\ldots\})$ that lives in the context. (In contrast, the ordered abstract machine semantics from the introduction was *flat*.) We discuss the defunctionalization transformation, which allows us to derive flat specifications from nested specifications, in Section 6.2 below.

The intuitive connection between natural semantics specifications and concurrent specifications has been explored previously and independently in the context of CLF by Schack-Nielsen [SN07] and by Cruz and Hou [CH12]; Schack-Nielsen proves the equivalence of the two specifications, whereas Cruz and Hou used the connection informally. The contribution of this section is to describe a general transformation (of which Figure 6.1 is one instance) and to prove the transformation correct in general. We have implemented the operationalization and defunctionalization transformations within the prototype implementation of SLS.

In Section 6.1.1 we will present the subset of specifications that our operationalization transformation handles, and in Section 6.1.2 we present the most basic form of the transformation. In Sections 6.1.3 and 6.1.4 we extend the basic transformation to be both tail-recursion optimizing and parallelism-enabling. Finally, in Section 6.1.5, we establish the correctness of the overall transformation.

```
#mode ev + -.                    eval: exp -> prop ord.
ev: exp -> exp -> prop.          retn: exp -> prop ord.

ev/lam:                          ev/lam:
ev (lam \x. E x)                 eval (lam \x. E x)
    (lam \x. E x).                >-> {retn (lam \x. E x)}.

ev/app:                          ev/app:
ev (app E1 E2) V                 eval (app E1 E2)
                                  >-> {eval E1 *
 <- ev E1 (lam \x. E x)                (All E. retn (lam \x. E x)
                                         >-> {eval E2 *
 <- ev E2 V2                                  (All V2. retn V2
                                                >-> {eval (E V2) *
 <- ev (E V2) V.                                     (All V. retn V
                                                       >-> {retn V})})})}.
```

Figure 6.1: Natural semantics (left) and ordered abstract machine (right) for CBV evaluation

6.1.1 Transformable signatures

The starting point for the operationalization transformation is a deductive signature that is well-moded in the sense described in Section 4.6.1. Every declared negative predicate will either remain defined by deductive proofs (we write the atomic propositions built with these predicates as p_d^-, d for deductive) or will be transformed into the concurrent fragment of SLS (we write these predicates as $\mathsf{a_c}$, $\mathsf{b_c}$ etc. and write the atomic propositions built with these predicates as p_c^-, c for concurrent).

For the purposes of describing and proving the correctness of the operationalization transformation, we will assume that all transformed atomic propositions p_c^- have two arguments where the first argument is moded as an input and the second is an output. That is, they are declared as follows:

$$\#\mathsf{mode}\ \mathsf{a_c} + -.$$
$$\mathsf{a_c} : \tau_1 \to \tau_2 \to \mathsf{prop}.$$

Without dependency, two-place relations are sufficient for describing n-place relations.[1] It should be possible to handle dependent predicates (that is, those with declarations of the form $\mathsf{a_c} : \Pi x{:}\tau_1.\, \tau_2(x) \to \mathsf{type}$), but we will not do so here.

The restriction on signatures furthermore enforces that all rules must be of the form $\mathsf{r} : C$ or $\mathsf{r} : D$, where C and D are refinements of the negative propositions of SLS that are defined as

[1] As an example, consider addition defined as a three-place relation $\mathsf{add}\, M\, N\, P$ (where add has kind $\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat} \to \mathsf{prop}$) with the usual mode ($\mathsf{add} + + -$). We can instead use a two-place relation $\mathsf{add}'\ (\mathsf{add\_inputs}\, M\, N)\, P$ with mode ($\mathsf{add}' + -$). The kind of add' is $\mathsf{add\_in} \to \mathsf{nat} \to \mathsf{type}$, where add_in is a new type with only one constructor $\mathsf{add\_inputs} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{add\_in}$ that effectively pairs together the two natural numbers that are inputs.

follows:

$$
\begin{aligned}
C &::= p_c^- \mid \forall x{:}\tau.\, C \mid p_{pers}^+ \rightarrowtail C \mid !p_c^- \rightarrowtail C \mid !G \rightarrowtail C \\
D &::= p_d^- \mid \forall x{:}\tau.\, D \mid p_{pers}^+ \rightarrowtail D \mid !p_c^- \rightarrowtail D \mid !G \rightarrowtail C \\
G &::= p_d^- \mid \forall x{:}\tau.\, G \mid p_{pers}^+ \rightarrowtail G \mid !D \rightarrowtail G
\end{aligned}
$$

For most of this chapter, we will restrict our attention to signatures where all atomic propositions have the form p_c^- and where all rules have the form C. This makes the classes p_d^-, D, and G irrelevant and effectively restricts rules to the Horn fragment. Propositions p_d^- that remain deductively defined by rules D will only be considered towards the end of this chapter in Section 6.6.2 when we consider various transformations on SOS specifications and in Section 6.6.3 when we consider transforming the natural semantics for Davies' $\lambda^{\bigcirc}$.

Note, however, that if a signature is well-formed given that a certain atomic proposition is assigned to the class p_c^- of transformed atomic propositions, the signature will *remain* well-formed if we instead assign that proposition to the class p_d^- of atomic propositions that get left in the deductive fragment. The only effect is that some rules that were previously of the form $\mathsf{r} : C$ will become rules of the form $\mathsf{r} : D$.[2] If we turn this dial all the way, we won't operationalize anything! If *all* atomic propositions are of the form p_d^- so that they remain deductive, then the propositions p_c^- and C are irrelevant, and the restriction above describes all persistent, deductive specifications – essentially, any signature that could be executed by the standard logic programming interpretation of LF [Pfe89]. The operationalization transformation will be the identity on such a specification.

All propositions C are furthermore equivalent (at the level of synthetic inference rules) to propositions of the form $\forall\overline{x_0} \ldots \forall\overline{x_n}.\, A_n^+ \rightarrowtail \ldots \rightarrowtail A_1^+ \rightarrowtail \mathsf{a_c}\, t_0\, t_{n+1}$, where the $\forall\overline{x_i}$ are shorthand for a series of universal quantifiers $\forall x_{i1}{:}\tau_{i1} \ldots \forall x_{ik}{:}\tau_{ik}$, and where each variable in $\overline{x_i}$ does not appear in t_0 (unless $i = 0$) nor in any A_j^+ with $j < i$ but does appear in A_i^+ (or t_0 if $i = 0$). Therefore, when we consider moded proof search, the variables bound in $\overline{x_0}$ are all fixed by the query and those bound in the other $\overline{x_i}$ are all fixed by the output position of the ith subgoal.

Each premise A_i^+ either has the form p_{pers}^+, $!p_c^-$, or $!G$. The natural deduction rule ev/app, which has three premises, can be represented in this standard form as follows:

$$
\begin{array}{llllll}
\forall\overline{x_0}. & \forall\overline{x_1}.\forall\overline{x_2}.\forall\overline{x_3}.\, A_3^+ \rightarrowtail & A_2^+ \rightarrowtail & A_1^+ \rightarrowtail & \mathsf{a_c}\, t_0 & t_4 \\
\forall E_1.\forall E_2.\forall E.\forall V_2.\forall V.\, !(\mathsf{ev}\,(E\,V_2)\,V) \rightarrowtail & !(\mathsf{ev}\,E_2\,V_2) \rightarrowtail & !(\mathsf{ev}\,E_1\,(\mathsf{lam}\,\lambda x.\,E\,x)) \rightarrowtail & \mathsf{ev}\,(\mathsf{app}\,E_1\,E_2)\,V & &
\end{array}
$$

From here on out we will assume without loss of generality that any proposition C actually has this very specific form.

6.1.2 Basic transformation

The operationalization transformation $Op(\Sigma)$ operates on SLS signatures Σ that have the form described in the previous section. We will first give the transformation on signatures; the transformation of rule declarations $\mathsf{r} : C$ is the key case.

[2]The reverse does not hold: the proposition $!(\forall x{:}\tau.\, p_{pers}^+ \rightarrowtail p_d^-) \rightarrowtail p_d^-$ has the form D, but the proposition $!(\forall x{:}\tau.\, p_{pers}^+ \rightarrowtail p_c^-) \rightarrowtail p_c^-$ does *not* have the form C.

Each two-place predicate $\mathsf{a_c}$ gets associated with two one-place predicates $\mathsf{eval\_a}$ and $\mathsf{retn\_a}$: both $\mathsf{eval\_a}\, t$ and $\mathsf{retn\_a}\, t$ are positive ordered atomic propositions. We will write $X^\dagger$ for the operation of substituting all occurrences of $p_c^- = \mathsf{a_c}\, t_1\, t_2$ with $(\mathsf{eval\_a}\, t_1 \rightarrowtail \{\mathsf{retn\_a}\, t_2\})$ in X. This substitution operation is used on propositions and contexts; it appears in the transformation of rules $\mathsf{r} : D$ below.

* $Op(\cdot) = \cdot$
* $Op(\Sigma, \mathsf{a_c} : \tau_1 \to \tau_2 \to \mathsf{prop}) = Op(\Sigma),\ \mathsf{eval\_a} : \tau_1 \to \mathsf{prop\,ord},\ \mathsf{retn\_a} : \tau_2 \to \mathsf{prop\,ord}$
* $Op(\Sigma, \mathsf{a_c} : \tau_1 \to \tau_2 \to \mathsf{prop}) = Op(\Sigma),\ \mathsf{eval\_a} : \tau_1 \to \mathsf{prop\,ord}$
  (if $\mathsf{retn\_a}$ *is already defined)*
* $Op(\Sigma, \mathsf{b} : \kappa) = Op(\Sigma),\ \mathsf{b} : \kappa$ *(if* $\mathsf{b} \neq \mathsf{a_c}$*)*
* $Op(\Sigma, \mathsf{c} : \tau) = Op(\Sigma),\ \mathsf{c} : \tau$
* $Op(\Sigma, \mathsf{r} : C) = Op(\Sigma),\ \mathsf{r} : \forall \overline{x_0}.\, \mathsf{eval\_a}\, t_0 \rightarrowtail [\![A_1^+, \ldots, A_n^+]\!](\mathsf{a}, t_{n+1}, \mathsf{id})$
  (where $C = \forall \overline{x_0} \ldots \forall \overline{x_n}.\, A_n^+ \rightarrowtail \ldots \rightarrowtail A_1^+ \rightarrowtail \mathsf{a_c}\, t_0\, t_{n+1}$*)*
* $Op(\Sigma, \mathsf{r} : D) = Op(\Sigma),\ \mathsf{r} : D^\dagger$

The transformation of a proposition $C = \forall \overline{x_0} \ldots \forall \overline{x_n}.\, A_n^+ \rightarrowtail \ldots \rightarrowtail A_1^+ \rightarrowtail \mathsf{a_c}\, t_0\, t_{n+1}$ involves the definition $[\![A_i^+, \ldots, A_n^+]\!](\mathsf{a}, t_{n+1}, \sigma)$, where σ substitutes only for variables in $\overline{x_j}$ where $j < i$. The function is defined inductively on the length of the sequence $A_i^+, \ldots, A_n^+$.

* $[\![]\!](\mathsf{a}, t_{n+1}, \sigma) = \{\mathsf{retn\_a}\, (\sigma t_{n+1})\}$
* $[\![p_{pers}^+, A_{i+1}^+, \ldots, A_n^+]\!](\mathsf{a}, t_{n+1}, \sigma) = \forall \overline{x_i}.\, (\sigma p_{pers}^+) \rightarrowtail [\![A_{i+1}^+, \ldots, A_n^+]\!](\mathsf{a}, t_{n+1}, \sigma)$
* $[\![!p_c^-, A_{i+1}^+, \ldots, A_n^+]\!](\mathsf{a}, t_{n+1}, \sigma)$
  $= \{\mathsf{eval\_b}\, (\sigma t_i^{in}) \bullet (\forall \overline{x_i}.\, \mathsf{retn\_b}\, (\sigma t_i^{out}) \rightarrowtail [\![A_{i+1}^+, \ldots, A_n^+]\!](\mathsf{a}, t_{n+1}, \sigma))\}$
  (where p_c^- *is* $\mathsf{b_c}\, t_i^{in}\, t_i^{out}$*)*
* $[\![!G, A_{i+1}^+, \ldots, A_n^+]\!](\mathsf{a}, t_{n+1}, \sigma) = \forall \overline{x_i}.\, !(\sigma G^\dagger) \rightarrowtail [\![A_{i+1}^+, \ldots, A_n^+]\!](\mathsf{a}, t_{n+1}, \sigma)$

This operation is slightly more general than it needs to be to describe the transformation on signatures, where the substitution σ will always just be the identity substitution id. Non-identity substitutions arise during the proof of correctness, which is why we introduce them here.

Figure 6.1, relating the natural semantics to the encoding of the search procedure as an ordered abstract machine, is an instance of this transformation.

6.1.3 Tail-recursion

Consider again our motivating example, the procedure for that takes expressions e and searches for expressions v such that $e \Downarrow v$ is derivable. If we were to implement that procedure as a functional program, the procedure would be *tail-recursive*. In the procedure that handles the case when $e = e_1\, e_2$, the last step invokes the search procedure recursively. If and when that callee returns v, then the caller will also return v.

Tail-recursion is significant in functional programming because tail-recursive calls can be implemented without allocating a stack frame: when a compiler makes this more efficient choice, we say it is performing *tail-recursion optimization*.[3] An analogous opportunity for tail-recursion

[3] Or *tail-call optimization*, as a tail-recursive function call is just a specific instance of a tail call.

optimization arises in our logical compilation procedure. In our motivating example, the last step in the $e_1\, e_2$ case was operationalized as a positive proposition of the form $\mathsf{eval}\,(E\,V_2) \bullet (\forall V.\, \mathsf{retn}\, V \rightarrowtail \{\mathsf{retn}\, V\})$. In a successful search, the process state

$$x{:}\mathsf{eval}\,(E\,V_2), y{:}(\forall V.\, \mathsf{retn}\, V \rightarrowtail \{\mathsf{retn}\, V\})\; ord, \Delta$$

will evolve until the state

$$x'{:}\mathsf{retn}\, V', y{:}(\forall V.\, \mathsf{retn}\, V \rightarrowtail \{\mathsf{retn}\, V\})\; ord, \Delta$$

is reached, at which point the next step, focusing on y, takes us to the the process state

$$y'{:}\mathsf{retn}\, V', \Delta$$

If we operationalize the last step in the $e_1\, e_2$ case as $\mathsf{eval}\,(E\,V_2)$ instead of as $\mathsf{eval}\,(E\,V_2) \bullet (\forall V.\, \mathsf{retn}\, V \rightarrowtail \{\mathsf{retn}\, V\})$, we will reach the same final state with one fewer transition. The tail-recursion optimizing version of the operationalization transformation creates concurrent computations that avoid these useless steps.

We cannot perform tail recursion in general because the output of the last subgoal may be different from the output of the goal. For example, the rule $\mathsf{r} : \forall X.\, \forall Y.\, !\mathsf{a}\, X\, Y \rightarrowtail \mathsf{a}\,(\mathsf{c}\, X)\,(\mathsf{c}\, Y)$ will translate to

$$\mathsf{r} : \forall X.\, \mathsf{eval\_a}\,(\mathsf{c}\, X) \rightarrowtail \{\mathsf{eval\_a}\, X \bullet (\forall Y.\, \mathsf{retn\_a}\, Y \rightarrowtail \{\mathsf{retn\_a}\,(\mathsf{c}\, Y)\})\}$$

There is no opportunity for tail-recursion optimization, because the output of the last search procedure, $t_n^{out} = Y$, is different than the value returned down the stack, $t_{n+1} = \mathsf{c}\, Y$. This case corresponds to functional programs that cannot be tail-call optimized.

More subtly, we cannot even eliminate all cases where $t_n^{out} = t_{n+1}$ unless these terms are *fully general*. The rule $\mathsf{r} : \forall X.\, !\mathsf{test}\, X\, \mathsf{true} \rightarrowtail \mathsf{test}\, \mathsf{s}\, X\, \mathsf{true}$, for example, will translate to

$$\mathsf{r} : \forall X.\, \mathsf{eval\_a}\, \mathsf{s}\, X \rightarrowtail \{\mathsf{eval\_a}\, X \bullet (\mathsf{retn\_a}\, \mathsf{true} \rightarrowtail \mathsf{retn\_a}\, \mathsf{true})\}$$

It would invalid to tail-call optimize in this situation. Even though the proposition $\mathsf{retn\_a}\, \mathsf{true} \rightarrowtail \mathsf{retn\_a}\, \mathsf{true}$ is an identity, if the proposition $\mathsf{retn\_a}\, \mathsf{false}$ appears to its left, the process state will be unable to make a transition. This condition doesn't have an analogue in functional programming, because it corresponds to the possibility that moded deductive computation can perform pattern matching on *outputs* and fail if the pattern match fails.

We say that t_{n+1} with type τ is fully general if all of its free variables are in $\overline{x_n}$ (and therefore not fixed by the input of any other subgoal) and if, for any variable-free term t' of type τ, there exists a substitution σ such that $t = \sigma t_{n+1}$. The simplest way to ensure this is to require that $t_{n+1} = t_n^{out} = y$ where $y = \overline{x_n}$.[4]

The tail-recursive procedure can be described by adding a new case to the definition of $[\![A_i^+, \ldots, A_n^+]\!](\mathsf{a}, t_{n+1}, \sigma)$:

[4]It is also possible to have a fully general $t_{n+1} = \mathsf{c}\, y_1\, y_2$ if, for instance, c has type $\tau_1 \rightarrow \tau_2 \rightarrow \mathsf{foo}$ and there are no other constructors of type foo. However, we also have to check that there are no other first-order variables in Ψ with types like $\tau_3 \rightarrow \mathsf{foo}$ that could be used to make other terms of type foo.

```
ev/lam: eval (lam \x. E x) >-> {retn (lam \x. E x)}.

ev/app: eval (app E1 E2)
         >-> {eval E1 *
              (All E. retn (lam \x. E x)
                >-> {eval E2 *
                      (All V2. retn V2 >-> {eval (E V2)})})}.
```

Figure 6.2: Tail-recursion optimized semantics for CBV evaluation

- $*$ *...(four other cases from Section 6.1.2)...*
- $*$ $[\![!\mathsf{b}\, t_n^{in}\, t_{n+1}]\!](\mathsf{a}, t_{n+1}, \sigma) = \{\mathsf{eval\_a}\,(\sigma t_n^{in})\}$
  (where t_{n+1} is fully general and $\mathsf{retn\_a} = \mathsf{retn\_b}$*)*

This case overlaps with the third case of the definition given in Section 6.1.2, which indicates that tail-recursion optimization can be applied or not in a nondeterministic manner.

Operationalizing the natural semantics from 6.1 with tail-recursion optimization gives us the ordered abstract machine in Figure 6.2.

6.1.4 Parallelism

Both the basic and the tail-recursive transformations are sequential: if $x{:}\mathsf{eval}\ulcorner e \urcorner \rightsquigarrow^* \Delta$, then the process state Δ contains at most one proposition $\mathsf{eval}\ulcorner e' \urcorner$ or $\mathsf{retn}\ulcorner v \urcorner$ that can potentially be a part of any further transition. Put differently, the first two versions of the operationalization transformation express deductive computation as a concurrent computation that does not exhibit any parallelism or concurrency (sequential computation being a special case of concurrent computation).

Sometimes, this is what we want: in Section 6.3 we will see that the sequential tail-recursion-optimized abstract machine adequately represents a traditional on-paper abstract machine for the call-by-value lambda calculus. In general, however, when distinct subgoals do not have input-output dependencies (that is, when none of subgoal i's outputs are inputs to subgoal $i + 1$), deductive computation can search for subgoal i and $i + 1$ simultaneously, and this can be represented in the operationalization transformation.

Parallelism will change the way we think about the structure of the ordered context: previously we were encoding a stack-like structure in the ordered context, and now we will encode a tree-like structure in the ordered context. It's really easy to encode a stack in an ordered context, as we have seen: we just write down the stack! Trees are only a little bit more complicated: we encode them in an ordered context by writing down an ordered tree traversal. Our translation uses a postfix traversal, so it is always possible to reconstruct a tree from the ordered context for the same reason that a postfix notations like Reverse Polish notation are unambiguous: there's always only one way to reconstruct the tree of subgoals.

In the previous transformations, our process states were structured such that every negative proposition A^- was waiting on a single retn to be computed to its left; at that point, the negative proposition could be focused upon, invoking the continuation stored in that negative proposition.

```
eval: exp -> prop ord.
retn: exp -> prop ord.

ev/lam: eval (lam \x. E x) >-> {retn (lam \x. E x)}.

ev/app: eval (app E1 E2)
         >-> {eval E1 * eval E2 *
              (All E. All V2. retn (lam \x. E x) * retn V2
                >-> {eval (E V2)})}.
```

Figure 6.3: Parallel, tail-recursion optimized semantics for CBV evaluation

If we ignore the first-order structure of the concurrent computation, these intermediate states look like this:

$$(\ldots \textit{subgoal 1} \ldots) \qquad y{:}(\mathsf{retn} \rightarrowtail \mathit{cont})\ \mathit{ord}$$

Note that *subgoal 1* is intended to represent some nonempty sequence of ordered propositions, not a single proposition. With the parallelism-enabling transformation, subgoal 1 will be able to perform parallel search for its own subgoals:

$$\textit{(subgoal 1.1)} \quad \textit{(subgoal 1.2)} \quad y_1{:}(\mathsf{retn}_{1.1} \bullet \mathsf{retn}_{1.2} \rightarrowtail \mathit{cont}_1)\ \mathit{ord}, \quad y{:}(\mathsf{retn} \rightarrowtail \mathit{cont})\ \mathit{ord}$$

The two subcomputations *(subgoal 1.1)* and *(subgoal 1.2)* are next to one another in the ordered context, but the postfix structure imposed on the process state ensures that the only way they can interact is if they both finish (becoming $z_{1.1}{:}\langle \mathsf{retn}_{1.1} \rangle$ and $z_{1.2}{:}\langle \mathsf{retn}_{1.2} \rangle$, respectively), which will allow us to focus on y_1 and begin working on the continuation cont_1.

To allow the transformed programs to enable parallel evaluation, we again add a new case to the function that transforms propositions C. The new case picks out $j - i$ premises $A_i, \ldots, A_j = !p^-_{ci}, \ldots, !p^-_{cj}$, requiring that those $j - i$ premises are *independent*. Each $p^-_{ck} = \mathsf{bk_c}\, t^{in}_k\, t^{out}_k$, where the term t^{out}_k is what determines the assignments for the variables in $\overline{x_k}$ when we perform moded proof search. Independence between premises requires that the free variables of t^{in}_k cannot include any variables in $\overline{x_m}$ for $i \leq m < k$; the well-modedness of the rule already ensures that t^{in}_k does not contain any variables in $\overline{x_m}$ for $k \leq m \leq j$.

$*$ *...(four other cases from Section 6.1.2, one other case from Section 6.1.3)...*

$*$ $[\![!p^-_{ci}, \ldots, !p^-_{cj}, A^+_{j+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma)$

$$= \{\mathsf{eval\_bi}\,(\sigma t^{in}_i) \bullet \ldots \bullet \mathsf{eval\_bj}\,(\sigma t^{in}_j) \bullet (\forall \overline{x_i} \ldots \forall \overline{x_j}.\ \mathsf{retn\_bi}\,(\sigma t^{out}_i) \bullet \ldots \bullet \mathsf{retn\_bj}\,(\sigma t^{out}_j) \rightarrowtail [\![A^+_{j+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma))\}$$

(where p^-_{ck} is $\mathsf{bk_c}\, t^{in}_k\, t^{out}_k$ and $\emptyset = FV(t^{in}_k) \cap (\overline{x_i} \cup \ldots \cup \overline{x_j})$ for $i \leq k \leq j$)

This new case subsumes the old case that dealt with sequences of the form $!p^-_c, A^+_{i+1}, \ldots, A^+_n$; that old case is now an instance of the general case where $i = j$. Specifically, the second side condition on the free variables, which is necessary if the resulting rule is to be well-scoped, is trivially satisfied in the sequential case where $i = j$.

The result of running the natural semantics from Figure 6.1 through the parallel and tail-recursion optimizing ordered abstract machine is shown in Figure 6.3; it shows that we can

search for the subgoals $e_1 \Downarrow \lambda x.e$ and $e_2 \Downarrow v_2$ in parallel. We cannot run either of these subgoals in parallel with the third subgoal $[v_2/x]e \Downarrow v$ because the input $[v_2/x]e$ mentions the outputs of both of the previous subgoals.

6.1.5 Correctness

We have presented, in three steps, a nondeterministic transformation. One reason for presenting a nondeterministic transformation is that the user can control this nondeterminism to operationalize with or without parallelism and with or without tail-call optimization. (The transformation as implemented in the SLS prototype only has one setting: it optimizes tail-calls but does not enable parallel evaluation.) The other reason for presenting a nondeterministic transformation is that we can prove the correctness of all the variants we have presented so far in one fell swoop by proving the correctness of the nondeterministic transformation.

Correctness is fundamentally the property that we have $\Psi; \Gamma \vdash_\Sigma \langle p_d^- \rangle$ if and only if we have $\Psi; \Gamma^\dagger \vdash_{Op(\Sigma)} \langle p_d^- \rangle$ and that we have $\Psi; \Gamma \vdash_\Sigma \langle \mathsf{a_c}\, t_1\, t_2 \rangle$ if and only if we have a trace $(\Psi; \Gamma, \mathsf{eval\_a}\, t_1) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Gamma, \mathsf{retn}\,(\mathsf{retn\_a}\, t_2))$. We label the forward direction "completeness" and the backward direction "soundness," but directional assignment is (as usual) somewhat arbitrary. Completeness is a corollary of Theorem 6.2, and soundness is a corollary of Theorem 6.4. We use Theorem 6.1 pervasively and usually without mention.

Theorem 6.1 (No effect on the LF fragment)**.** *$\Psi \vdash_\Sigma t : \tau$ if and only if $\Psi \vdash_{Op(\Sigma)} t : \tau$.*

Proof. Straightforward induction in both directions; the transformation leaves the LF-relevant part of the signature unchanged. □

Completeness

Theorem 6.2 (Completeness of operationalization)**.** *If all propositions in Γ have the form $x{:}D$ pers or $z{:}\langle p^+_{pers} \rangle$, then*

1. *If $\Psi; \Gamma \vdash_\Sigma \langle p_d^- \rangle$, then $\Psi; \Gamma^\dagger \vdash_{Op(\Sigma)} \langle p_d^- \rangle$.*
2. *If $\Psi; \Gamma, [D] \vdash_\Sigma \langle p_d^- \rangle$, then $\Psi; \Gamma^\dagger, [D^\dagger] \vdash_{Op(\Sigma)} \langle p_d^- \rangle$.*
3. *If $\Psi; \Gamma \vdash_\Sigma G$, then $\Psi; \Gamma^\dagger \vdash_{Op(\Sigma)} G^\dagger$.*
4. *If Δ matches $\Theta\{\!\{\Gamma\}\!\}$ and $\Psi; \Gamma \vdash_\Sigma \langle p_c^- \rangle$ (where $p_c^- = \mathsf{a_c}\, t\, s$), then $(\Psi; \Theta^\dagger\{x{:}\langle \mathsf{eval\_a}\, t \rangle\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Theta^\dagger\{y{:}\langle \mathsf{retn\_a}\, s \rangle\})$.*

Proof. Mutual induction on the structure of the input derivation.

The first three parts are straightforward. In part 1, we have $\Psi; \Gamma \vdash_\Sigma h \cdot Sp : \langle p_d^- \rangle$ where either $h = x$ and $x{:}D \in \Gamma$ or else $h = \mathsf{r}$ and $\mathsf{r} : D \in \Sigma$. In either case the necessary result is $h \cdot Sp'$, where we get Sp' from the induction hypothesis (part 2) on Sp.

In part 2, we proceed by case analysis on the proposition D in focus. The only interesting case is where $D = !p_c^- \rightarrowtail D'$

- If $D = p_d^-$, then $Sp = \text{NIL}$ and NIL gives the desired result.

* If $D = \forall x{:}\tau.\, D'$ or $D = p^+_{pers} \rightarrowtail D'$, then $Sp = (t; Sp')$ or $Sp = (z; Sp')$ (respectively). The necessary result is $(t; Sp'')$ or $(z; Sp'')$ (respectively) where we get Sp'' from the induction hypothesis (part 2) on Sp'.
* If $D = !p^-_c \rightarrowtail D'$ and $p^-_c = \mathsf{a_c}\, t_1\, t_2$, then $Sp = (!N; Sp')$ and $D^\dagger = !(\mathsf{eval\_a}\, t_1 \rightarrowtail \bigcirc(\mathsf{retn\_a}\, t_2) \rightarrowtail D'^\dagger)$.[5]

$$
\begin{array}{ll}
\Psi; \Gamma \vdash_\Sigma N : \langle \mathsf{a_c}\, t_1\, t_2 \rangle & \text{(given)} \\
\Psi; \Gamma, [D] \vdash_\Sigma Sp' : \langle p^-_d \rangle & \text{(given)} \\
T :: (\Psi; \Gamma^\dagger, x{:}\langle \mathsf{eval\_a}\, t_1 \rangle) \leadsto^*_{Op(\Sigma)} (\Psi; \Gamma^\dagger, y{:}\langle \mathsf{retn\_a}\, t_2 \rangle) & \text{(ind. hyp. (part 4) on } N) \\
\Psi; \Gamma, [D'^\dagger] \vdash_{Op(\Sigma)} Sp'' : \langle p^-_d \rangle & \text{(ind. hyp. (part 2) on } Sp') \\
\Psi; \Gamma^\dagger \vdash_{Op(\Sigma)} \lambda x.\, \{\mathsf{let}\, T\, \mathsf{in}\, y\} : \mathsf{eval\_a}\, t_1 \rightarrowtail \bigcirc(\mathsf{retn\_a}\, t_2) & \text{(construction)} \\
\Psi; \Gamma^\dagger, [D^\dagger] \vdash_{Op(\Sigma)} !(\lambda x.\, \{\mathsf{let}\, T\, \mathsf{in}\, y\}); Sp' : \langle p^-_d \rangle & \text{(construction)}
\end{array}
$$

* If $D = !G \rightarrowtail D'$, then $Sp = !N; Sp'$. The necessary result is $!N'; Sp''$. We get N' from the induction hypothesis (part 3) on N and get Sp'' from the induction hypothesis (part 2) on Sp'.

The cases of part 3 are straightforward invocations of the induction hypothesis (part 1 or part 3). For instance, if $G = !D \rightarrowtail G'$ then we have a derivation of the form $\lambda x.N$ where $\Psi; \Gamma, x{:}D\ pers \vdash_\Sigma N : G'$. By the induction hypothesis (part 3) we have $\Psi; \Gamma^\dagger, x{:}D^\dagger\ pers \vdash_{Op(\Sigma)} N' : G'^\dagger$, and we conclude by constructing $\lambda x.N'$.

In part 4, we have $\Psi; \Gamma \vdash_\Sigma \mathsf{r} \cdot Sp : \langle p^-_d \rangle$, where $\mathsf{r}{:}C \in \Sigma$ and the proposition C is equivalent to $\forall \overline{x_0} \ldots \forall \overline{x_n}.\, A^+_n \rightarrowtail \ldots \rightarrowtail A^+_1 \rightarrowtail \mathsf{a_c}\, t_0\, t_{n+1}$ as described in Section 6.1.2. This means that, for each $0 \leq i \leq n$, we can decompose Sp to get $\sigma_i = (\overline{s_0}/\overline{x_0}, \ldots, \overline{s_i}/\overline{x_i})$ (for some terms $\overline{s_0} \ldots \overline{s_i}$ that correspond to the correct variables) and we have a value $\Psi; \Gamma \vdash_\Sigma V_i : [\sigma_i A^+_i]$. We also have $t = \sigma_0 t_0$ and $s = \sigma_n t_{n+1}$. It suffices to show that, for any $1 \leq i \leq n$, there is

* a spine Sp such that $\Psi; \Gamma^\dagger, [[\![A^+_i, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_0)] \vdash_{Op(\Sigma)} Sp : \langle \bigcirc C^+ \rangle$,
* a pattern+trace $\lambda P.T :: (\Psi; \Theta^\dagger\{C^+\}) \leadsto^*_{Op(\Sigma)} (\Psi; \Theta^\dagger\{y{:}\mathsf{retn\_a}\, (\sigma_n t_{n+1})\})$.[6]

Once we prove this, the trace we need to show is $\{P\} \leftarrow \mathsf{r} \cdot (s_0; Sp); T$.

We will prove this by induction on the length of the sequence sequence $A_i, \ldots, A_n$, and proceed by case analysis on the definition of the operationalization transformation:

* $[\![]\!](\mathsf{a}, t_{n+1}, \sigma_n) = \bigcirc(\mathsf{retn\_a}\, (\sigma_n t_{n+1}))$

  This is a base case: let $Sp = \textsc{nil}$, $P = y$, and $T = \diamond$, and we are done.

* $[\![!\mathsf{a_c}\, t^{in}_n\, t_{n+1}]\!](\mathsf{a}, t_{n+1}, \sigma_{n-1}) = \bigcirc(\mathsf{eval\_a}\, (\sigma_{n-1} t^{in}_n))$

  We are given a value $\Psi; \Gamma \vdash_\Sigma !N : [!\mathsf{a_c}\, (\sigma_n t^{in}_n)\, (\sigma_n t_{n+1})]$; observe that $\sigma_{n-1} t^{in}_n = \sigma_n t^{in}_n$. This is also a base case: let $Sp = \textsc{nil}$, and let $P = x_n :: (\Psi; \Theta^\dagger\{\mathsf{eval\_a}\, (\sigma_n t^{in}_n)\}) \Longrightarrow_{Op(\Sigma)} (\Psi; \Theta^\dagger\{x_n{:}\langle \mathsf{eval\_a}\, (\sigma_n t^{in}_n) \rangle\})$. We need a trace $T :: (\Psi; \Theta^\dagger\{x_n{:}\langle \mathsf{eval\_a}\, (\sigma_n t^{in}_n) \rangle\}) \leadsto^*_{Op(\Sigma)}$

[5] Recall that $\bigcirc C^+$ and $\{C^+\}$ are synonyms for the internalization of the lax modality; we will use the $\bigcirc$ in the course of this proofs in this section.

[6] The derived pattern+trace form is discussed at the end of Section 4.2.6.

$(\Psi; \Theta^\dagger\{y{:}\langle \mathsf{retn\_a}\,(\sigma_n t_{n+1})\rangle\})$; this follows from the outer induction hypothesis (part 4) on N.

$*$ $[\![p^+_{pers}, A^+_{i+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1}) = \forall\overline{x_i}.\, \sigma_{i-1}p^+{}_{pers} \rightarrowtail [\![A^+_{i+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1})$

$\Psi; \Gamma \vdash_\Sigma z : [\sigma_i p^+_{pers}]$ (given)
$\sigma_i = (\sigma_{i-1}, \overline{s_i}/\overline{x_i})$ (definition of σ_i)
$\Psi; \Gamma, [[\![A^+_{i+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_i)] \vdash_{Op(\Sigma)} Sp' : \langle \bigcirc C^+\rangle$ (by inner ind. hyp.)
$\lambda P.T :: (\Psi; \Theta^\dagger\{C^+\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Theta^\dagger\{y{:}\mathsf{retn\_a}\,(\sigma_n t_{n+1})\})$ ”
$\Psi; \Gamma, [\forall\overline{x_i}.\, (\sigma_{i-1}p^+{}_{pers}) \rightarrowtail [\![A^+_{i+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1})] \vdash_{Op(\Sigma)} (\overline{s_i}; z; Sp') : \langle \bigcirc C^+\rangle$ (construction)

We conclude by letting $Sp = \overline{s_i}; z; Sp'$.

$*$ $[\![!p^-_{ci}, \ldots, !p^-_{cj}, A^+_{j+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1})$

$$= \bigcirc \left(\begin{array}{l} \mathsf{eval\_bi}\,(\sigma_{i-1}t^{in}_i) \bullet \ldots \bullet \mathsf{eval\_bj}\,(\sigma_{i-1}t^{in}_j) \bullet \\ (\forall\overline{x_i} \ldots \forall\overline{x_j}.\, \mathsf{retn\_bi}\,(\sigma_{i-1}t^{out}_i) \bullet \ldots \bullet \mathsf{retn\_bj}\,(\sigma_{i-1}t^{out}_j) \\ \quad \rightarrowtail [\![A^+_{j+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1}))) \end{array} \right)$$

(where p^-_{ck} is $\mathsf{bk_c}\, t^{in}_k\, t^{out}_k$ and $\emptyset = FV(t^{in}_k) \cap (\overline{x_i} \cup \ldots \cup \overline{x_j})$ for $i \leq k \leq j$)

Let $Sp = \textsc{nil}$ and $P = y_i, \ldots, y_j, y_{ij}$. It suffices to show that there is a trace

$$\begin{aligned} T :: \ & (\Psi, \Theta^\dagger\{y_i{:}\langle \mathsf{eval\_bi}\,(\sigma_{i-1}t^{in}_i)\rangle, \ldots, y_j{:}\langle \mathsf{eval\_bj}\,(\sigma_{i-1}t^{in}_j)\rangle, \\ & \qquad y_{ij}{:}(\forall\overline{x_i} \ldots \forall\overline{x_j}.\, \mathsf{retn\_bi}\,(\sigma_{i-1}t^{out}_i) \bullet \ldots \bullet \mathsf{retn\_bj}\,(\sigma_{i-1}t^{out}_j) \\ & \qquad\qquad \rightarrowtail [\![A^+_{j+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1}))\ ord\}) \\ & \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Theta^\dagger\{z{:}\langle \mathsf{retn\_a}\,(\sigma_n t_{n+1})\rangle\}) \end{aligned}$$

$\Psi; \Gamma \vdash_\Sigma !N_k : [!\mathsf{bk_c}\,(\sigma_k t^{in}_k)\,(\sigma_k t^{out}_k)]$ $(i \leq k \leq j)$ (given)
$\Psi; \Gamma \vdash_\Sigma !N_k : [!\mathsf{bk_c}\,(\sigma_{i-1}t^{in}_k)\,(\sigma_j t^{out}_k)]$ $(i \leq k \leq j)$ (condition on translation, defn. of σ_k)

$$\begin{aligned} T_0 :: \ & (\Psi, \Theta^\dagger\{y_i{:}\langle \mathsf{eval\_bi}\,(\sigma_{i-1}t^{in}_i)\rangle, \ldots, y_j{:}\langle \mathsf{eval\_bj}\,(\sigma_{i-1}t^{in}_j)\rangle, \\ & \qquad y_{ij}{:}(\forall\overline{x_i} \ldots \forall\overline{x_j}.\, \mathsf{retn\_bi}\,(\sigma_{i-1}t^{out}_i) \bullet \ldots \bullet \mathsf{retn\_bj}\,(\sigma_{i-1}t^{out}_j) \\ & \qquad\qquad \rightarrowtail [\![A^+_{j+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1}))\ ord\}) \\ & \rightsquigarrow^*_{Op(\Sigma)} (\Psi, \Theta^\dagger\{z_i{:}\langle \mathsf{retn\_bi}\,(\sigma_j t^{out}_i)\rangle, \ldots, z_j{:}\langle \mathsf{retn\_bj}\,(\sigma_j t^{out}_j)\rangle, \\ & \qquad y_{ij}{:}(\forall\overline{x_i} \ldots \forall\overline{x_j}.\, \mathsf{retn\_bi}\,(\sigma_{i-1}t^{out}_i) \bullet \ldots \bullet \mathsf{retn\_bj}\,(\sigma_{i-1}t^{out}_j) \\ & \qquad\qquad \rightarrowtail [\![A^+_{j+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1}))\ ord\}) \end{aligned}$$

(by outer ind. hyp. (part 4) on each of the N_k in turn)
$\Psi; \Gamma, [[\![A^+_{j+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_j)] \vdash_{Op(\Sigma)} Sp' : \langle \bigcirc C^+\rangle$ (by inner ind. hyp.)
$\lambda P'.T' :: (\Psi, \Theta^\dagger\{C^+\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi, \Theta^\dagger\{y{:}\langle \mathsf{retn\_a}\, s\rangle\})$ ”

We conclude by letting $T = (T_0; \{P'\} \leftarrow y_{ij} \cdot (\overline{s_i}; \ldots \overline{s_j}; (y_i \bullet \ldots \bullet y_j); Sp'); T')$.

$*$ $[\![!G, A^+_{i+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1}) = \forall\overline{x_i}.\, !(\sigma_{i-1}G^\dagger) \rightarrowtail [\![A^+_i, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1})$

$\Psi; \Gamma \vdash_\Sigma !N : [!\sigma_i G]$ (given)
$\Psi; \Gamma^\dagger \vdash_{Op(\Sigma)} N' : \sigma_i G^\dagger$ (by outer ind. hyp. (part 3) on N)
$\sigma_i = (\sigma_{i-1}, \overline{s_i}/\overline{x_i})$. (definition of σ_i)
$\Psi; \Gamma, [[\![A^+_i, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_i)] \vdash_{Op(\Sigma)} Sp' : \langle \bigcirc C^+ \rangle$ (by inner ind. hyp.)
$\lambda P.T :: (\Psi; \Theta^\dagger\{C^+\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Theta^\dagger\{y{:}\mathsf{retn\_a}\, s\})$ ”
$\Psi; \Gamma, [\forall\overline{x_i}.\, !(\sigma_{i-1}G) \rightarrowtail [\![A^+_i, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1})] \vdash_{Op(\Sigma)} (\overline{s_i}; !N'; Sp') : \langle \bigcirc C^+ \rangle$ (construction)

We conclude by letting $Sp = \overline{s_i}; !N'; Sp'$.

This completes the inner induction in the fourth part, and hence the completeness proof. □

Soundness

Soundness under the parallel translation requires us to prove an inversion lemma like the one that we first encountered in the proof of preservation for adequacy (Theorem 4.7). To this end, we describe two new refinements of negative propositions that capture the structure of transformed concurrent rules.

$$R ::= \forall\overline{x}.\, \mathsf{retn\_b1}\, t_1 \bullet \ldots \bullet \mathsf{retn\_bn}\, t_n \rightarrowtail S$$
$$S ::= \forall x{:}\tau.\, S \mid p^+_{pers} \rightarrowtail S \mid !A^- \rightarrowtail S \mid \bigcirc(\mathsf{eval\_b1}\, t_1 \bullet \ldots \bullet \mathsf{eval\_bn}\, t_n \bullet \downarrow R) \mid \bigcirc(\mathsf{eval\_b}\, t)$$

Every concurrent rule in a transformed signature $Op(\Sigma)$ has the form $\mathsf{r} : \forall\overline{x}.\, \mathsf{eval\_b}\, t \rightarrowtail S$.

Theorem 6.3 (Rearrangement). *If Δ contains only atomic propositions, persistent propositions of the form D, and ordered propositions of the form R, and if Γ matches $z{:}\langle \mathsf{retn\_z}\, t_z \rangle$, then*

1. *If Δ matches $\Theta\{\!\{x_1{:}\langle \mathsf{retn\_b1}\, t_1 \rangle, \ldots, x_n{:}\langle \mathsf{retn\_bn}\, t_n \rangle, y{:}(\forall\overline{x}.\, \mathsf{retn\_b1}\, s_1 \bullet \ldots \bullet \mathsf{retn\_bn}\, s_n \rightarrowtail S)\}\!\}$ and $T :: (\Psi; \Delta) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Gamma)$, then $T = \{P\} \leftarrow y \cdot (\overline{u}; (x_1 \bullet \ldots \bullet x_n); Sp); T'$ where $(\overline{u}/\overline{x})s_i = t_i$ for $1 \leq i \leq n$.*
2. *If Δ matches $\Theta\{\!\{y{:}\langle \mathsf{eval\_b}\, t \rangle\}\!\}$ and $T :: (\Psi; \Delta) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Gamma)$, then $T = \{P\} \leftarrow \mathsf{r} \cdot (\overline{u}; y; Sp); T'$ where $\mathsf{r} : \forall\overline{x}.\, \mathsf{eval\_b}\, s \rightarrowtail S \in Op(\Sigma)$.*

Proof. In both cases, the proof is by induction on the structure of T and case analysis on the first steps in T. If the first step does not proceed by focusing on y (part 1) or focusing on a rule in the signature and consuming y (part 2), then we proceed by induction on the smaller trace to move the relevant step to the front. We have to check that the first step doesn't output any variables that are input variables of the relevant step. This is immediate from the structure of R, S, and transformed signatures. □

The main soundness theorem is Theorem 6.4. The first three cases of Theorem 6.4 are straightforward transformations from deductive proofs to deductive proofs, and the last two cases are the key. In the last two cases, we take a trace that, by its type, must contain the information needed to reconstruct a deductive proof.

Theorem 6.4 (Soundness of operationalization)**.** *If all propositions in Γ have the form $x{:}D$ pers or $z{:}\langle p^+_{pers}\rangle$, and all propositions in Δ have the form $x{:}D$ pers, $z{:}\langle p^+_{pers}\rangle$, $x{:}\langle \mathsf{eval\_b}\, t\rangle$, $x{:}\langle \mathsf{retn\_b}\, t\rangle$, or $x{:}R$ ord, then*

1. *If $\Psi; \Gamma^\dagger \vdash_{Op(\Sigma)} \langle p^-_d\rangle$, then $\Psi; \Gamma \vdash_\Sigma \langle p^-_d\rangle$.*
2. *If $\Psi; \Gamma^\dagger, [D^\dagger] \vdash_{Op(\Sigma)} \langle p^-_d\rangle$, then $\Psi; \Gamma, [D] \vdash_\Sigma \langle p^-_d\rangle$.*
3. *If $\Psi; \Gamma^\dagger \vdash_{Op(\Sigma)} G^\dagger$, then $\Psi; \Gamma \vdash_\Sigma G$.*
4. *If Δ matches $\Theta^\dagger\{\!\{\Gamma^\dagger\}\!\}$, Γ_z matches $z{:}\langle \mathsf{retn\_z}\, t_z\rangle$,*
   and $(\Psi; \Theta^\dagger\{\!\{\Gamma, x{:}\langle \mathsf{eval\_a}\, t\rangle\}\!\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Gamma_z)$,
   then there exists an s s.t. $\Psi; \Gamma \vdash_\Sigma \langle \mathsf{a_c}\, t\, s\rangle$ and $(\Psi; \Theta^\dagger\{y{:}\langle \mathsf{retn\_a}\, s\rangle\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Gamma_z)$.
5. *If Δ matches $\Theta^\dagger\{\!\{\Gamma^\dagger\}\!\}$, Γ_z matches $z{:}\langle \mathsf{retn\_z}\, t_z\rangle$,*
   $\Psi; \Gamma^\dagger, [[\![A_i, \ldots, A_n]\!](\mathsf{a}, t_{n+1}, \sigma_i)] \vdash_{Op(\Sigma)} \langle \bigcirc C^+\rangle$, and $(\Psi; \Theta^\dagger\{C^+\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Gamma_z)$,
   then there exists a $\sigma \supseteq \sigma_i$ such that $\Psi; \Gamma \vdash_\Sigma [\sigma A^+_j]$ for $i \le j \le n$
   and $(\Psi; \Theta^\dagger\{y{:}\langle \mathsf{retn\_a}\, (\sigma t_{n+1})\rangle\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Delta_z)$.

Proof. By induction on the structure of the given derivation or given trace. Parts 1 and 3 exactly match the structure of the completeness proof (Theorem 6.2); the only difference in Part 2 is the case where $D = !p_c \rightarrowtail D'$. In his case, we reason as follows:

$D = !p_c \rightarrowtail D'$, where $p_c = \mathsf{a_c}\, t_1\, t_2$	(given)
$D^\dagger = !(\mathsf{eval\_a}\, t_1 \rightarrowtail \bigcirc(\mathsf{retn\_a}\, t_2)) \rightarrowtail D'^\dagger$	(definition of $D^\dagger$)
$\Psi; \Gamma, [!(\mathsf{eval\_a}\, t_1 \rightarrowtail \bigcirc(\mathsf{retn\_a}\, t_2)) \rightarrowtail D'^\dagger] \vdash_{Op(\Sigma)} Sp : \langle p^-_d\rangle$	(given)
$Sp = !(\lambda x.\, \{\mathsf{let}\, T\, \mathsf{in}\, y\}); Sp'$, where	(inversion on the type of Sp)
$\quad T :: (\Psi; \Gamma, x{:}\langle \mathsf{eval\_a}\, t_1\rangle) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Gamma_z)$,	”
$\quad \Gamma_z$ matches $y{:}\langle \mathsf{retn\_a}\, t_2\rangle$, and	”
$\quad \Psi; \Gamma, [D'^\dagger] \vdash_{Op(\Sigma)} Sp' : \langle p^-_d\rangle$	”
$\Psi; \Gamma, [D'] \vdash_\Sigma Sp'' : \langle p^-_d\rangle$	(ind. hyp. (part 2) on Sp')
$\Psi; \Gamma \vdash_\Sigma N : \langle \mathsf{a_c}\, t_1\, s\rangle$	(ind. hyp. (part 4) on T)
$T' :: (\Psi; \Gamma, y'{:}\langle \mathsf{retn\_a}\, s\rangle) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Gamma_z)$	”
$T' = \diamond$, $y' = y$, and $t_2 = s$	(case analysis on the structure of T')
$\Psi; \Gamma, [!p_c \rightarrowtail D'] \vdash_\Sigma !N; Sp' : \langle p^-_d\rangle$	(construction)

Part 4 we let $\sigma_0 = (\overline{u}/\overline{x})$.

$T :: (\Psi; \Theta^\dagger\{\!\{x{:}\langle \mathsf{eval\_a}\, t\rangle\}\!\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Gamma_z)$	(given)
$T = \{P\} \leftarrow \mathsf{r} \cdot (\overline{u}; x; Sp); T'$	(Theorem 6.3 on T)
$\quad \mathsf{r} : \forall \overline{x_0}.\, \mathsf{eval\_a}\, t_0 \rightarrowtail [\![A^+_1, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \mathsf{id}) \in Op(\Sigma)$	”
$\quad t = \sigma_0 t_0$	”
$\quad \Psi; \Gamma, [[\![A^+_1, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_0)] \vdash_{Op(\Sigma)} Sp : \langle \bigcirc C^+\rangle$	”
$\quad \lambda P.T' :: (\Psi; \Theta^\dagger\{C^+\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Gamma_z)$	”
$\Psi; \Gamma \vdash_\Sigma V_i : [\sigma A^+_i]$ for $1 \le j \le n$	(ind. hyp. (part 5) on Sp and T')
$T'' :: (\Psi; \Theta^\dagger\{y{:}\langle \mathsf{retn\_a}\, (\sigma t_{n+1})\rangle\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Gamma_z)$	”

We needed to show a derivation and a trace: the trace T'' is precisely the latter thing. For the derivation of $\mathsf{a_c}\, t\, s$ (for some s), we observe that $\mathsf{r} \in \Sigma$ has a type equivalent to $\forall \overline{x_0} \ldots \forall \overline{x_n}.\, A^+_n \rightarrowtail \ldots \rightarrowtail A^+_0 \rightarrowtail \mathsf{a_c}\, t_0\, t_{n+1}$. Therefore, by letting $s = \sigma t_{n+1}$ and using the V_i from the induction

hypothesis (part 5) above, we can construct a derivation of $\Psi; \Gamma \vdash_\Sigma \langle \mathsf{a_c}\, t\, s \rangle$ by focusing on r, which is the other thing we needed to show.

The step above where we assume that the head of r involves the predicate $\mathsf{a_c}$ is the only point in the correctness proofs where we rely on the fact that each transformed predicate eval_a is associated with exactly one original-signature predicate $\mathsf{a_c}$ – if this were not the case and eval_a were also associated with an original-signature predicate b, then we might get the "wrong" derivation back from this step. We do not have to similarly rely on retn_a being similarly uniquely associated with $\mathsf{a_c}$.

Part 5 We are given a spine $\Psi; \Gamma^\dagger, [[\![A_i, \ldots, A_n]\!](\mathsf{a}, t_{n+1}, \sigma_i)] \vdash_{Op(\Sigma)} Sp : \langle \bigcirc C^+ \rangle$ and a trace $\lambda P.T :: (\Psi; \Theta^\dagger\{C^+\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Gamma_z)$. We need to show a $\sigma \supseteq \sigma_i$, an $\Psi; \Gamma \vdash_{Op(\Sigma)} N : [\sigma A_k^+]$ for $i \leq j \leq n$, and $T' :: (\Psi; \Theta^\dagger\{y{:}\langle \mathsf{retn\_a}\,(\sigma t_{n+1})\rangle\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Delta_z)$.

We proceed by case analysis on the definition of the operationalization transformation:

* $[\![]\!](\mathsf{a}, t_{n+1}, \sigma_n) = \bigcirc(\mathsf{retn\_a}\,(\sigma_n t_{n+1}))$

  This is a base case: there are no values to construct. By inversion on the type of P we know it has the form $y :: (\Psi; \Theta^\dagger\{\mathsf{retn\_a}\,(\sigma_n t_{n+1})\}) \Longrightarrow_{Op(\Sigma)} (\Psi; \Theta^\dagger\{y{:}\langle \mathsf{retn\_a}\,(\sigma_n t_{n+1})\rangle\})$. Let $\sigma = \sigma_n$ and we are done; the trace T is the necessary trace.

  Because $(\Psi; \Theta^\dagger\{\mathsf{retn\_a}\,(\sigma_n t_{n+1})\})$ decomposes to $(\Psi; \Theta^\dagger\{y{:}\langle \mathsf{retn\_a}\,(\sigma_n t_{n+1})\rangle\})$, we let $\sigma = \sigma_n$ and we are done.

* $[\![!\mathsf{a_c}\, t_n^{in}\, t_{n+1}]\!](\mathsf{a}, t_{n+1}, \sigma_{n-1}) = \bigcirc(\mathsf{eval\_a}\,(\sigma_{n-1} t_n^{in}))$

  This is also a base case: we have one value of type $!\mathsf{a_c}\,(\sigma t_n^{in})\,(\sigma t_{n+1})$ to construct, where $\sigma \supseteq \sigma_{n-1}$. By inversion on the type of P we know that the pattern has the form $y :: (\Psi; \Theta^\dagger\{\mathsf{eval\_a}\,(\sigma_n t_n^{in})\}) \Longrightarrow_{Op(\Sigma)} (\Psi; \Theta^\dagger\{y{:}\langle \mathsf{eval\_a}\,(\sigma_n t_n^{in})\rangle\})$. This means we also have that $T :: (\Psi; \Theta^\dagger\{y{:}\langle \mathsf{eval\_a}\,(\sigma_n t_n^{in})\rangle\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Delta_z)$.

  By the induction hypothesis (part 4) on T, we have an s such that $\Psi; \Gamma \vdash_\Sigma N : \langle \mathsf{a_c}\, t\, s \rangle$ and a trace $T' :: (\Psi; \Theta^\dagger\{y{:}\langle \mathsf{retn\_a}\, s\rangle\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Gamma_z)$.

  We can only apply tail-recursion optimization when t_{n+1} is fully general, which means we can construct a $\sigma \supseteq \sigma_{n-1}$ such that $\sigma t_{n+1} = s$. The value we needed to construct is just $!N$, and the trace T' is in the form we need, so we are done.

* $[\![p^+_{pers}, A^+_{i+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1}) = \forall \overline{x_i}.\, \sigma_{i-1} p^+{}_{pers} \rightarrowtail [\![A^+_{i+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1})$

  By type inversion, the spine $Sp = \overline{u_i}; z; Sp'$. Let $\sigma_i = (\sigma_{i-1}, \overline{u_i}/\overline{x_i})$. The induction hypothesis (part 5) on Sp' and T gives $\sigma \supseteq \sigma_i$, values σA^+_j for $i < j \leq n$, and a trace $T' :: (\Psi; \Theta^\dagger\{y{:}\langle \mathsf{retn\_a}\,(\sigma t_{n+1})\rangle\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Delta_z)$. The remaining value of $\sigma A_i = \sigma p^+_{pers}$ is just z.

* $[\![!p^-_{ci}, \ldots, !p^-_{cj}, A^+_{j+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1})$

$$= \bigcirc \left(\begin{array}{l} \mathsf{eval\_bi}\,(\sigma_{i-1} t_i^{in}) \bullet \ldots \bullet \mathsf{eval\_bj}\,(\sigma_{i-1} t_j^{in}) \bullet \\ (\forall \overline{x_i} \ldots \forall \overline{x_j}.\, \mathsf{retn\_bi}\,(\sigma_{i-1} t_i^{out}) \bullet \ldots \bullet \mathsf{retn\_bj}\,(\sigma_{i-1} t_j^{out}) \\ \quad \rightarrowtail [\![A^+_{j+1}, \ldots, A^+_n]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1})) \end{array} \right)$$

  (where p^-_{ck} is $\mathsf{bk_c}\, t_k^{in}\, t_k^{out}$ and $FV(t_k^{in}) \notin (\overline{x_i} \cup \ldots \cup \overline{x_j})$ for $i \leq k \leq j$)

Let $R = \forall\overline{x_i} \ldots \forall\overline{x_j}.\, \mathsf{retn\_bi}\,(\sigma_{i-1}t_i^{out}) \bullet \ldots \bullet \mathsf{retn\_bj}\,(\sigma_{i-1}t_j^{out})$
$\rightarrowtail [\![A_{j+1}^+, \ldots, A_n^+]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1})$

By inversion on the type of P, we know it has the form $y_i, \ldots, y_j, y$, and we know that $T :: (\Psi; \Theta^\dagger\{y_i{:}\langle\mathsf{eval\_bi}\,(\sigma_{i-1}t_i^{in})\rangle, \ldots, y_j{:}\langle\mathsf{eval\_bj}\,(\sigma_{i-1}t_j^{in})\rangle, y{:}R\ ord\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi, \Delta_z)$.

By $j - i$ applications of the induction hypothesis (part 4) starting with T, we obtain for $i \leq k \leq j$ a derivation of $\Psi; \Gamma \vdash_\Sigma N_k : \langle\mathsf{bk_c}\,(\sigma_{i-1}t_k^{in})\, s_k\rangle$ and a smaller trace $T'' :: (\Psi; \Theta^\dagger\{z_i{:}\langle\mathsf{retn\_bi}\, s_i\rangle, \ldots, z_j{:}\langle\mathsf{retn\_bj}\, s_j\rangle, y{:}R\ ord\}) \rightsquigarrow^+_{Op(\Sigma)} (\Psi; \Delta_z)$.

By Theorem 6.3 we get that $T'' = \{P\} \leftarrow y \cdot (\overline{u_i}; \ldots; \overline{u_j}; (z_i \bullet \ldots \bullet z_j); Sp'); T'''$.

Let $\sigma_j = (\sigma_{i-1}, \overline{u_i}/\overline{x_i}, \ldots, \overline{u_j}/\overline{x_j})$. Then we have that $s_k = \sigma_j t_k^{out}$ for $i \leq k \leq j$ and $\Psi; \Gamma^\dagger, [[\![A_{j+1}^+, \ldots, A_n^+]\!](\mathsf{a}, t_{n+1}, \sigma_j)] \vdash_{Op(\Sigma)} Sp' : \langle\bigcirc C^+\rangle$.*

The induction hypothesis (part 5) on Sp' and T''' gives $\sigma \supseteq \sigma_j$, values σA_k^+ for $j < k \leq n$, and a trace $T' :: (\Psi; \Theta^\dagger\{y{:}\langle\mathsf{retn\_a}\,(\sigma t_{n+1})\rangle\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Delta_z)$. The remaining values of type $\sigma A_k = !(\sigma p_{ck}^-)$ for $i \leq k \leq j$ all have the form $!N_k$ (where the N_k were constructed above by invoking part 4 of the induction hypothesis).

* $[\![!G, A_{i+1}^+, \ldots, A_n^+]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1}) = \forall\overline{x_i}.\, !\sigma_{i-1}G^\dagger \rightarrowtail [\![A_i^+, \ldots, A_n^+]\!](\mathsf{a}, t_{n+1}, \sigma_{i-1})$

By type inversion, the spine $Sp = \overline{u_i}; !N; Sp'$. Let $\sigma_i = (\sigma_{i-1}, \overline{u_i}/\overline{x_i})$ The induction hypothesis (part 5) on Sp' and T gives $\sigma \supseteq \sigma_i$, values σA_j^+ for $i < j \leq n$, and a trace $T' :: (\Psi; \Theta^\dagger\{y{:}\langle\mathsf{retn\_a}\,(\sigma t_{n+1})\rangle\}) \rightsquigarrow^*_{Op(\Sigma)} (\Psi; \Delta_z)$. The remaining value of type $\sigma A_i = !(\sigma G)$ is $!N'$, where we get $\Psi; \Gamma \vdash_\Sigma N' : \sigma G$ from the induction hypothesis (part 3) on N.

This completes the proof. □

6.2 Logical transformation: defunctionalization

Defunctionalization is a procedure for turning nested SLS specifications into flat SLS specifications. The key idea is that a nested rule can always be simulated by a distinguished atomic proposition by inserting a rule into the signature that teaches the atomic proposition how to act like the nested rule. (Or, looking at it the other way around, the new atomic propositions act like triggers for the additional rules.) The correctness of defunctionalization follows from a simple lock-step bisimulation argument – if a trigger can cause a rule in the defunctionalized signature to fire, then that trigger corresponds to a nested rule in the context that can fire immediately in the non-defunctionalized process state, and vice-versa.

The defunctionalization procedure implemented in the SLS prototype is actually three transformations. The first is properly a defunctionalization transformation (Section 6.2.1), the second is an uncurrying transformation (Section 6.2.2), and the third is a refactoring that transforms a family of cont-like predicates into a single cont predicate and a family of frames (Section 6.2.3). We will explain each of these transformations in turn; one example of the full three-part defunctionalization transformation on a propositional signature is given in Figure 6.4.

$$
\begin{array}{lcl}
\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d} : \mathsf{prop\,ord}, & & \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d} : \mathsf{prop\,ord}, \\
 & & \mathsf{frame} : \mathsf{type}, \\
 & & \mathsf{cont} : \mathsf{frame} \to \mathsf{type}. \\
 & & \mathsf{frameA1} : \mathsf{frame}, \\
\mathsf{ruleA} : \mathsf{a} \rightarrowtail \{\mathsf{b} \bullet \downarrow(\mathsf{c} \rightarrowtail \{\mathsf{d}\})\}, & & \mathsf{ruleA} : \mathsf{a} \rightarrowtail \{\mathsf{b} \bullet \mathsf{cont\ frameA1}\}, \\
 & \Longrightarrow & \mathsf{ruleA1} : \mathsf{c} \bullet \mathsf{cont\ frameA1} \rightarrowtail \{\mathsf{d}\}, \\
 & & \mathsf{frameB1} : \mathsf{frame}, \\
 & & \mathsf{frameB2} : \mathsf{frame}, \\
\mathsf{ruleB} : \mathsf{c} \rightarrowtail \{\downarrow(\mathsf{d} \rightarrowtail \{\downarrow(\mathsf{a} \bullet \mathsf{a} \rightarrowtail \{\mathsf{b}\})\})\} & & \mathsf{ruleB} : \mathsf{c} \rightarrowtail \{\mathsf{cont\ frameB1}\}, \\
 & & \mathsf{ruleB1} : \mathsf{d} \bullet \mathsf{cont\ frameB1} \rightarrowtail \{\mathsf{cont\ frameB2}\}, \\
 & & \mathsf{ruleB2} : \mathsf{a} \bullet \mathsf{a} \bullet \mathsf{cont\ frameB2} \rightarrowtail \{\mathsf{b}\}
\end{array}
$$

Figure 6.4: Defunctionalization on a nested SLS signature

6.2.1 Defunctionalization

Defunctionalization is based on the following intuitions: if A^- is a closed negative proposition and we have a single-step transition $\{P\} \leftarrow y \cdot Sp :: (\Psi; \Theta\{y{:}A^-\ ord\}) \rightsquigarrow_\Sigma (\Psi; \Delta')$, then we can define an augmented signature

$$
\begin{array}{rl}
\Sigma' = & \Sigma, \\
 & \mathsf{cont} : \mathsf{prop\,ord}, \\
 & \mathsf{run\_cont} : \mathsf{cont} \rightarrowtail A^-
\end{array}
$$

and it is the case that $(\Psi; \Theta\{y{:}\langle\mathsf{cont}\rangle\}) \rightsquigarrow_{\Sigma'} (\Psi; \Delta')$ as well. Whenever the step $\{P\} \leftarrow y \cdot Sp$ is possible under Σ, the step $\{P\} \leftarrow \mathsf{run\_cont} \cdot (y; Sp)$ will be possible under Σ', and vice versa. (It would work just as well for run_cont to be $\mathsf{cont} \twoheadrightarrow A^-$ – the persistent propositions $A^+ \rightarrowtail B^-$ and $A^+ \twoheadrightarrow B^-$ are indistinguishable in ordered logic.)

Because rules in the signature must be closed negative propositions, this strategy won't work for a transition that mentions free variables from the context. However, every open proposition $\Psi \vdash_\Sigma A^-\ \mathsf{type}^-$ can be refactored as an open proposition $a_1{:}\tau_1, \ldots, a_n{:}\tau_n \vdash B^-\ \mathsf{type}^-$ and a substitution $\sigma = (t_1/a_1, \ldots, t_n/a_n)$ such that $\Psi \vdash_\Sigma \sigma : a_1{:}\tau_1, \ldots, a_n{:}\tau_n$ and $\sigma B^- = A^-$. Then, we can augment the signature in this more general form:

$$
\begin{array}{rl}
\Sigma'' = & \Sigma, \\
 & \mathsf{cont} : \Pi a_1{:}\tau_1 \ldots \Pi a_n{:}\tau_n.\ \mathsf{prop\,ord} \\
 & \mathsf{run\_cont} : \forall a_1{:}\tau_1 \ldots \forall a_n{:}\tau_n.\ \mathsf{cont}\ a_1 \ldots a_n \rightarrowtail B^-
\end{array}
$$

As before, whenever the step $\{P\} \leftarrow y \cdot Sp :: (\Psi; \Theta\{y : A^-\ ord\}) \rightsquigarrow^*_\Sigma (\Psi'; \Delta')$ is possible under Σ, the step $\{P\} \leftarrow \mathsf{run\_cont} \cdot (t_1; \ldots; t_n; y; Sp) :: (\Psi; \Theta\{y : \langle\mathsf{cont}\ t_1 \ldots t_n\rangle\}) \rightsquigarrow^*_{\Sigma'} (\Psi'; \Delta')$ will be possible under Σ', and vice versa.

Taking this a step further, if we have a signature $\Sigma_1 = \Sigma, \mathsf{rule} : \ldots \rightarrowtail \bigcirc(\ldots \bullet \downarrow B^- \bullet \ldots)$ where the variables $a_1 \ldots a_n$ are free in B^-, then any trace in this signature will be in lock-step bisimulation with a trace in this signature:

$$\begin{aligned}\Sigma_2 = \ & \Sigma, \\ & \mathsf{cont} : \Pi a_1{:}\tau_1 \ldots \Pi a_n{:}\tau_n.\, \mathsf{prop\ ord} \\ & \mathsf{rule} : \ldots \rightarrowtail \bigcirc(\ldots \bullet (\mathsf{cont}\, a_1 \ldots a_n) \bullet \ldots) \\ & \mathsf{run\_cont} : \forall a_1{:}\tau_1 \ldots \forall a_n{:}\tau_n.\, \mathsf{cont}\, a_1 \ldots a_n \rightarrowtail B^-\end{aligned}$$

Specifically, say that $(\Psi; \Delta_1)\, \mathcal{R}\, (\Psi; \Delta_2)$ when Δ_1 is Δ_2 where all variable declarations of the form $y{:}\langle \mathsf{cont}\, t_1 \ldots t_n \rangle\ ord$ in Δ_2 have been replaced by $y{:}(t_1/a_1 \ldots t_n/a_n)B^-\ ord$ in Δ_1. (Note that if $\vdash_{\Sigma_2} (\Psi; \Delta_2)\, \mathsf{state}$ holds, then $\vdash_{\Sigma_1} (\Psi; \Delta_1)\, \mathsf{state}$ holds as well.) It is the case that if $S_1 :: (\Psi; \Delta_1) \rightsquigarrow^*_{\Sigma_1} (\Psi'; \Delta_1')$ then $S_2 :: (\Psi; \Delta_2) \rightsquigarrow^*_{\Sigma_2} (\Psi'; \Delta_2')$ where $(\Psi'; \Delta_1')\, \mathcal{R}\, (\Psi'; \Delta_2')$. The opposite also holds: if $S_2 :: (\Psi; \Delta_2) \rightsquigarrow^*_{\Sigma_2} (\Psi'; \Delta_2')$ then $S_1 :: (\Psi; \Delta_1) \rightsquigarrow^*_{\Sigma_1} (\Psi'; \Delta_1')$ where $(\Psi'; \Delta_1')\, \mathcal{R}\, (\Psi'; \Delta_2')$. For transitions not involving rule or run_cont in Σ_2 this is immediate, for transitions involving rule we observe that the propositions introduced by inversion preserve the simulation, and for transitions involving run_cont we use aforementioned fact that the atomic term $y \cdot Sp$ in Σ_1 is equivalent to the atomic term $\mathsf{run\_cont} \cdot (t_1; \ldots; t_n; y; Sp)$ in Σ_2.

We can iterate this defunctionalization procedure on the nested ev/app rule from Figure 6.2:

$$\begin{aligned}\mathsf{ev/app} : \ & \forall E_1.\forall E_2.\, \mathsf{eval}\, (\mathsf{app}\, E_1\, E_2) \\ & \rightarrowtail \{\mathsf{eval}\, E_1 \bullet \\ & \quad \downarrow(\forall E.\, \mathsf{retn}\, (\mathsf{lam}\, \lambda x.E\, x) \\ & \quad\quad \rightarrowtail \{\mathsf{eval}\, E_2 \bullet \downarrow(\forall V_2.\, \mathsf{retn}\, V_2 \rightarrowtail \{\mathsf{eval}\, (E\, V_2)\})\})\}.\end{aligned}$$

The outermost nested rule only has the variable E_2 free, so the first continuation we introduce, cont_app1, has one argument.

$$\begin{aligned}& \mathsf{cont\_app1} : \mathsf{exp} \rightarrow \mathsf{prop\ ord}, \\ & \mathsf{ev/app} : \forall E_1.\forall E_2.\, \mathsf{eval}\, (\mathsf{app}\, E_1\, E_2) \rightarrowtail \{\mathsf{eval}\, E_1 \bullet \mathsf{cont\_app1}\, E_2\} \\ & \mathsf{ev/app1} : \forall E_2.\, \mathsf{cont\_app1}\, E_2 \rightarrowtail \forall E.\, \mathsf{retn}\, (\mathsf{lam}\, \lambda x.E\, x) \\ & \qquad \rightarrowtail \{\mathsf{eval}\, E_2 \bullet \downarrow(\forall V_2.\, \mathsf{retn}\, V_2 \rightarrowtail \{\mathsf{eval}\, (E\, V_2)\})\}.\end{aligned}$$

This step turns ev/app into a flat rule; we repeat defunctionalization on ev/app1 to get a completely flat specification. This introduces a new proposition cont_app2 that keeps track of the free variable E with type exp $\rightarrow$ exp.

$$\begin{aligned}& \mathsf{cont\_app1} : \mathsf{exp} \rightarrow \mathsf{prop\ ord}, \\ & \mathsf{cont\_app2} : (\mathsf{exp} \rightarrow \mathsf{exp}) \rightarrow \mathsf{prop\ ord}, \\ & \mathsf{ev/app} : \forall E_1.\forall E_2.\, \mathsf{eval}\, (\mathsf{app}\, E_1\, E_2) \rightarrowtail \{\mathsf{eval}\, E_1 \bullet \mathsf{cont\_app1}\, E_2\} \\ & \mathsf{ev/app1} : \forall E_2.\, \mathsf{cont\_app1}\, E_2 \rightarrowtail \forall E.\, \mathsf{retn}\, (\mathsf{lam}\, \lambda x.E\, x) \rightarrowtail \{\mathsf{eval}\, E_2 \bullet \mathsf{cont\_app2}\, (\lambda x.E\, x)\}. \\ & \mathsf{ev/app2} : \forall E.\, \mathsf{cont\_app2}\, (\lambda x.E\, x) \rightarrowtail \forall V_2.\, \mathsf{retn}\, V_2 \rightarrowtail \{\mathsf{eval}\, (E\, V_2)\}.\end{aligned}$$

```
eval: exp -> prop ord.
retn: exp -> prop ord.
cont_app1: exp -> prop ord.
cont_app2: (exp -> exp) -> prop ord.

ev/lam:  eval (lam \x. E x) >-> {retn (lam \x. E x)}.

ev/app:  eval (app E1 E2) >-> {eval E1 * cont_app1 E2}.

ev/app1: retn (lam \x. E x) * cont_app1 E2
          >-> {eval E2 * cont_app2 (\x. E x)}.

ev/app2: retn V2 * cont_app2 (\x. E x) >-> {eval (E V2)}.
```

Figure 6.5: Uncurried call-by-value evaluation

6.2.2 Uncurrying

The first arrow in a rule can be freely switched between left and right ordered implication: the rules $A^+ \rightarrowtail \{B^+\}$ and $A^+ \twoheadrightarrow \{B^+\}$ are equivalent, for instance. Pfenning used $A^+ \multimapdot \{B^+\}$ as a generic form of ordered implication for this reason in [Pfe04]. This observation only holds because rules act like persistent resources, however! It does not seem to be possible to treat $\multimapdot$ as a real connective in ordered logic with well-behaved left and right rules that satisfy cut admissibility, and the observation applies only to the first arrow: while the rule $\mathsf{rule1} : A^+ \rightarrowtail B^+ \rightarrowtail \{C^+\}$ is equivalent to the rule $\mathsf{rule2} : A^+ \twoheadrightarrow B^+ \rightarrowtail \{C^+\}$, these two rules are *not* equivalent to the rule $\mathsf{rule3} : A^+ \rightarrowtail B^+ \twoheadrightarrow \{C^+\}$.

Uncurrying tries to rewrite a rule so that the only arrow is the first one, taking an awkward rule like $A^+ \rightarrowtail B^+ \twoheadrightarrow C^+ \rightarrowtail D^+ \twoheadrightarrow \{E^+\}$ to the more readable flat rule $C^+ \bullet A^+ \bullet B^+ \bullet D^+ \rightarrowtail \{E^+\}$. Uncurrying can only be performed on persistent or linear propositions (or rules in the signature): there is no A^+ that makes the variable declaration $x{:}(p^+ \rightarrowtail q^+ \twoheadrightarrow \{r^+\})\ ord$, equivalent to $x{:}(A^+ \twoheadrightarrow \{r^+\})\ ord$ or $x{:}(A^+ \rightarrowtail \{r^+\})\ ord$ for any A^+.

Thus, defunctionalization and uncurrying work well together: if we replace the variable declaration $x{:}(p^+ \rightarrowtail q^+ \twoheadrightarrow \{r^+\})\ ord$ with the suspended ordered proposition $x{:}\langle \mathsf{cont} \rangle\ ord$ and add a rule $\mathsf{run\_cont} : \mathsf{cont} \rightarrowtail p^+ \rightarrowtail q^+ \twoheadrightarrow \{r^+\}$, that rule can then be uncurried to get the equivalent rule $p^+ \bullet \mathsf{cont} \bullet q^+ \rightarrowtail \{r^+\}$.

If we uncurry the defunctionalized specification for CBV evaluation from the previous section, we get the SLS specification shown in Figure 6.5. This flat specification closely and adequately represents the abstract machine semantics from the beginning of this chapter, but before proving adequacy in Section 6.3, we will make one more change to the semantics.

6.2.3 From many predicates to many frames

This last change we make appears to be largely cosmetic, but it will facilitate, in Section 6.5.4 below, the modular extension of our semantics with recoverable failure.

```
eval: exp -> prop ord.
retn: exp -> prop ord.
cont: frame -> prop ord.

app1: exp -> frame.
app2: (exp -> exp) -> frame.

ev/lam:  eval (lam \x. E x) >-> {retn (lam \x. E x)}.

ev/app:  eval (app E1 E2)  >-> {eval E1 * cont (app1 E2)}.

ev/app1: retn (lam \x. E x) * cont (app1 E2)
          >-> {eval E2 * cont (app2 \x. E x)}.

ev/app2: retn V2 * cont (app2 \x. E x) >-> {eval (E V2)}.
```

Figure 6.6: A first-order ordered abstract machine semantics for CBV evaluation

The defunctionalization procedure introduces many new atomic propositions. The two predicates introduced in the call-by-value specification were called cont_app1 and cont_app2, and a larger specification will, in general, introduce many more. The one last twist we make is to observe that, instead of introducing a new ordered atomic proposition $\mathsf{cont}\,\bar{t}$ for each iteration of the defunctionalization procedure, it is possible to introduce a single type ($\mathsf{frame} : \mathsf{type}$) and a single atomic proposition ($\mathsf{cont} : \mathsf{frame} \to \mathsf{prop\,ord}$).

With this change, each iteration of the defunctionalization procedure adds a new constant with type $\Pi y_1{:}\tau_1 \ldots \Pi y_m{:}\tau_m.\,\mathsf{frame}$ to the signature instead of a new atomic proposition with kind $\Pi y_1{:}\tau_1 \ldots \Pi y_m{:}\tau_m.\,\mathsf{prop\,ord}$. Operationally, these two approaches are equivalent, but it facilitates the addition of control features when we can modularly talk about all of the atomic propositions introduced by defunctionalization as having the form $\mathsf{cont}\,F$ for some term F of type frame.

The ordered abstract machine resulting from this version of defunctionalization and uncurrying is shown in Figure 6.6; this specification can be compared to the one in Figure 6.5.

6.3 Adequacy with abstract machines

The four-rule abstract machine specification given at the beginning of this chapter is adequately represented by the derived SLS specification in Figure 6.6. For terms and for deductive computations, adequacy is a well-understood concept: we know what it means to define an adequate encoding function $\ulcorner e \urcorner = t$ from "on-paper" terms e with (potentially) variables $x_1, \ldots, x_n$ free to LF terms t where $x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp} \vdash t : \mathsf{exp}$, and we know what it means to adequately encode the judgment $e \Downarrow v$ as a negative atomic SLS proposition $\mathsf{ev}\ulcorner e \urcorner \ulcorner v \urcorner$ and to encode derivations of this judgment to SLS terms N where $\cdot; \cdot \vdash_\Sigma N : \langle \mathsf{ev}\ulcorner e \urcorner \ulcorner v \urcorner \rangle$ [HHP93, HL07]. In Section 4.4 we discussed the methodology of adequacy and applied it to the very simple push-down automata from the introduction. In this section, we will repeat this development for Figure 6.6. The gen-

```
value: exp -> prop.
value/lam: value (lam \x. E x).

gen: prop ord.
gen/eval: gen >-> {eval E}.
gen/retn: gen * !value V >-> {retn V}.
gen/cont: gen >-> {gen * cont F}.
```

Figure 6.7: The generative signature Σ_{Gen} describing states Δ that equal $\ulcorner\!\ulcorner s \urcorner\!\urcorner$ for some s

erative signature itself has a slightly different character, but beyond that our discussion closely follows the contours of the adequacy argument from Section 4.4.

Recall the definition of states s, frames f, and stacks k from the beginning of this chapter. Our first step will be to define an interpretation function $\ulcorner\!\ulcorner s \urcorner\!\urcorner = \Delta$ from abstract machine states s to process states Δ so that, for example, the state

$$((\ldots(\mathsf{halt}; \Box\, e_1)\ldots); (\lambda x.e_n)\, \Box) \lhd v$$

is interpreted as the process state

$$y{:}\langle \mathsf{retn} \ulcorner v \urcorner\rangle,\ \ x_n{:}\langle \mathsf{cont}\,(\mathsf{app2}\,\lambda x.\ulcorner e_n \urcorner)\rangle,\ \ \ldots,\ \ x_1{:}\langle \mathsf{cont}\,(\mathsf{app1}\ulcorner e_1 \urcorner)\rangle,$$

We also define a generative signature that precisely captures the set of process states in the image of this translation. Having done so, we prove that the property that encoded abstract machine states $\ulcorner\!\ulcorner s \urcorner\!\urcorner \leadsto^*_{\Sigma_{CBV}} \Delta'$, where Σ_{CBV} stands for the signature in Figure 6.6, only when $\Delta' = \ulcorner\!\ulcorner s' \urcorner\!\urcorner$ for some abstract machine state s'. Then, the main adequacy result, that the interpretation of state s steps to the interpretation of state s' if and only if $s \mapsto s'$, follows by case analysis.

6.3.1 Encoding states

Our first goal is to describe a signature Σgen with the property that if $x{:}\langle \mathsf{gen}\rangle \leadsto^*_{\Sigma_{Gen}} \Delta$ and $\Delta \lightning_{\Sigma_{CBV}}$, then Δ encodes an abstract machine state s. A well-formed process state that represents an abstract machine state $((\ldots(\mathsf{halt}; f_1); \ldots); f_n) \rhd e$ has the form

$$y{:}\langle \mathsf{eval} \ulcorner e \urcorner\rangle,\ \ x_n{:}\langle \mathsf{cont} \ulcorner f_n \urcorner\rangle,\ \ \ldots,\ \ x_1{:}\langle \mathsf{cont} \ulcorner f_1 \urcorner\rangle$$

where $\ulcorner \Box\, e_2 \urcorner = \mathsf{app1} \ulcorner e_2 \urcorner$ and $\ulcorner (\lambda x.e)\, \Box \urcorner = \mathsf{app2}\,(\lambda x.\ulcorner e \urcorner)$. A well-formed process state representing a state $k \lhd v$ has the same form, but with $\mathsf{retn} \ulcorner v \urcorner$ instead of $\mathsf{eval} \ulcorner e \urcorner$. We also stipulate that $k \lhd v$ is only well-formed if v is actually a value – in our current specifications, the only values are functions $\ulcorner \lambda x.e \urcorner = \mathsf{lam}\, \lambda x.\ulcorner e \urcorner$.

The simplest SLS signature that encodes well-formed states has the structure of a context-free grammar like the signature that encoded well-formed PDA states in Figure 4.15. The judgment $\mathsf{value} \ulcorner e \urcorner$ captures the refinement of expressions e that are values. In addition to the four declarations above, the full signature Σgen includes all the type, proposition, and constant declarations from Figure 6.6, but none of the rules.

Note that this specification cannot reasonably be run as a logic program, because the variables E, V and F appear to be invented out of thin air. Rather than traces in these signatures being produced by some concurrent computation (such as forward chaining), they are produced and manipulated by the constructive content of theorems like the ones in this section.

Theorem 6.5 (Encoding)**.** *Up to variable renaming, there is a bijective correspondence between abstract machine states s and process states Δ such that $T :: (x{:}\langle \mathsf{gen} \rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen}} \Delta$ and $\Delta \lightning_{\Sigma_{PDA}}$.*

The proof of Theorem 6.5 follows the structure of Theorem 4.6: we first define context encoding functions $\ulcorner s \urcorner$ and $\ulcorner k \urcorner$.

* $\ulcorner k \triangleright e \urcorner = y{:}\langle \mathsf{eval} \ulcorner e \urcorner \rangle,\ \ulcorner k \urcorner$
* $\ulcorner k \triangleleft v \urcorner = y{:}\langle \mathsf{retn} \ulcorner v \urcorner \rangle,\ \ulcorner k \urcorner$
* $\ulcorner \mathsf{halt} \urcorner = \cdot$
* $\ulcorner k; f \urcorner = x_i{:}\langle \mathsf{cont} \ulcorner f \urcorner \rangle,\ \ulcorner k \urcorner$

It is simple to observe that the encoding function is injective (that $\ulcorner s \urcorner = \ulcorner s' \urcorner$ if and only if $s = s'$), so injectivity boils down to showing that every state s can be generated as a trace $\ulcorner s \urcorner = T :: (x{:}\langle gen \rangle) \rightsquigarrow^*_{\Sigma_{Gen}} \ulcorner s \urcorner$. Surjectivity requires us to do induction on the structure of T and case analysis on the first steps in T. Both steps require a lemma that the notion of value expressed by the predicate value in Σ_{Gen} matches the notion of values v used to define well-formed states $k \triangleleft v$.

6.3.2 Preservation and adequacy

Generated world preservation proceeds as in Section 4.4.3 and Theorem 4.7; this theorem has a form that we will consider further in Chapter 9.

Theorem 6.6 (Preservation)**.** *If $T_1 :: (x{:}\langle \mathsf{gen} \rangle) \rightsquigarrow^*_{\Sigma Gen} \Delta_1$, $\Delta_1 \lightning_{\Sigma_{CBV}}$, and $S :: \Delta_1 \rightsquigarrow_{\Sigma_{CBV}} \Delta_2$, then $T_2 :: (x{:}\langle \mathsf{gen} \rangle) \rightsquigarrow^*_{\Sigma_{Gen}} \Delta_2$.*

The proof proceeds by enumerating the synthetic transitions possible under Σ_{CBV}, performing inversion on the structure of the trace T_1, and using the results to construct the necessary result. This is the most interesting part of the adequacy proof, and a generalization of this preservation proof is carefully considered in Section 9.2 along with a more detailed discussion of the relevant inversion principles. With this property established, the final step is a straightforward enumeration as Theorem 4.8 in Section 4.4.4 was.

Theorem 6.7 (Adequacy of the transition system)**.** $\ulcorner s \urcorner \rightsquigarrow_{\Sigma_{CBV}} \ulcorner s' \urcorner$ *if and only if* $s \mapsto s'$.

As in Section 4.4.4, the proof is a straightforward enumeration. An immediate corollary of Theorems 6.5-6.7 is the stronger adequacy result that if $\ulcorner s \urcorner \rightsquigarrow_{\Sigma_{CBV}} \Delta$ then $\Delta = \ulcorner s' \urcorner$ for some s' such that $s \mapsto s'$.

$$\frac{[\mathsf{fix}\, x.e/x]e \Downarrow v}{\mathsf{fix}\, x.e \Downarrow v}$$

```
ev/fix:   eval (fix (\x. E x))
           >-> {eval (E (fix (\x. E x)))}.
```

$$\frac{}{\langle\rangle \Downarrow \langle\rangle}$$

```
ev/unit:  eval unit >-> {retn unit}.
```

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{\langle e_1, e_2\rangle \Downarrow \langle v_1, v_2\rangle}$$

```
ev/pair:  eval (pair E1 E2)
           >-> {eval E1 * eval E2 * cont pair1}.
ev/pair1: retn V1 * retn V2 * cont pair1
           >-> {retn (pair V1 V2)}.
```

$$\frac{e \Downarrow \langle v_1, v_2\rangle}{e.1 \Downarrow v_1}$$

```
ev/fst:   eval (fst E) >-> {eval E * cont fst1}.
ev/fst1:  retn (pair V1 V2) * cont fst1 >-> {retn V1}.
```

$$\frac{e \Downarrow \langle v_1, v_2\rangle}{e.2 \Downarrow v_2}$$

```
ev/snd:   eval (snd E) >-> {eval E * cont snd1}.
ev/snd1:  retn (pair V1 V2) * cont snd1 >-> {retn V2}.
```

$$\frac{}{\mathsf{z} \Downarrow \mathsf{z}}$$

```
ev/zero:  eval zero >-> {retn zero}.
```

$$\frac{e \Downarrow v}{\mathsf{s}\, e \Downarrow \mathsf{s}\, v}$$

```
ev/succ:  eval (succ E) >-> {eval E * cont succ1}.
ev/succ1: retn V * cont succ1 >-> {retn (succ V)}.
```

Figure 6.8: Semantics of some pure functional features

6.4 Exploring the image of operationalization

The examples given in the previous section all deal with call-by-value semantics for the untyped lambda calculus, which has the property that any expression will either evaluate forever or will eventually evaluate to a value $\lambda x.e$. We now want to discuss ordered abstract machines with traces that might get *stuck*. One way to raise the possibility of stuck states is to add values besides $\lambda x.e$. In Figure 6.8 we present an extension to Figure 6.6 with some of the features of a pure "Mini-ML" functional programming language: fixed-point recursion ($\ulcorner \mathsf{fix}\, x.e \urcorner = \mathsf{fix}\, \lambda x.\ulcorner e \urcorner$), units and pairs ($\ulcorner \langle\rangle \urcorner = \mathsf{unit}$, $\ulcorner \langle e_1, e_2\rangle \urcorner = \mathsf{pair} \ulcorner e_1 \urcorner \ulcorner e_2 \urcorner$), projections ($\ulcorner e.1 \urcorner = \mathsf{fst} \ulcorner e \urcorner$), $\ulcorner e.2 \urcorner = \mathsf{snd} \ulcorner e \urcorner$), and natural numbers ($\ulcorner \mathsf{z} \urcorner = \mathsf{zero}$, $\ulcorner \mathsf{s}\, e \urcorner = \mathsf{succ} \ulcorner e \urcorner$). The natural semantics is given on the left-hand side of that figure, and the operationalized and defunctionalized ordered abstract machine that arises from (an SLS encoding of) that natural semantics is given on the right.

Note that, facilitated by the nondeterminism inherent in operationalization, we chose parallel evaluation of pairs even though the execution of functions is sequential in Figure 6.6. Traditional abstract machine semantics are syntactic and do not handle parallel evaluation; therefore, it is not possible to show that this ordered abstract machine adequately encodes a traditional abstract machine presentation of Mini-ML. Nevertheless, the SSOS specification as a whole adapts seamlessly to the presence of this new feature.

As we discussed in Chapter 4, when we treat a natural semantics specification as an inductive definition, only the behavior of terminating computations can be observed, and it is not possible to distinguish a non-terminating term like $\mathsf{fix}\, x.x$ from a stuck term like $\mathsf{z}.1$ without relying on

$$\frac{e_1 \Downarrow v}{e_1 \,⍰\, e_2 \Downarrow v}$$

```
ev/choose1: eval (choose E1 E2)
              >-> {eval E1}.
```

$$\frac{e_2 \Downarrow v}{e_1 \,⍰\, e_2 \Downarrow v}$$

```
ev/choose2: eval (choose E1 E2)
              >-> {eval E2}.
```

Figure 6.9: Semantics of nondeterministic choice

a characterization of partial proofs. This is one of the problems that Leroy and Grall sought to overcome in their presentation of coinductive big-step operational semantics [LG09]. They defined the judgment $e \Uparrow^{\infty}$ coinductively; therefore it is easy to express the difference between non-terminating terms ($\mathsf{fix}\, x.x \Uparrow^{\infty}$) and stuck ones (there is no v such that $\mathsf{z.1} \Downarrow v$ or $\mathsf{z.1} \Uparrow^{\infty}$).

The translation of natural semantics into ordered abstract machines also allows us to distinguish $\mathsf{fix}\, x.x$ from $\mathsf{z.1}$. The former expression generates a trace that can always be extended:

$$x_1{:}\langle \mathsf{eval}\ (\mathsf{fix}\ \lambda x.x)\rangle \rightsquigarrow x_2{:}\langle \mathsf{eval}\ (\mathsf{fix}\ \lambda x.x)\rangle \rightsquigarrow x_3{:}\langle \mathsf{eval}\ (\mathsf{fix}\ \lambda x.x)\rangle \rightsquigarrow \ldots$$

whereas the latter gets stuck and can make no more transitions:

$$x_1{:}\langle \mathsf{eval}\ (\mathsf{fst}\ \mathsf{zero})\rangle \rightsquigarrow x_2{:}\langle \mathsf{eval}\ \mathsf{zero}\rangle, y{:}\langle \mathsf{cont}\ \mathsf{fst1}\rangle \rightsquigarrow x_3{:}\langle \mathsf{retn}\ \mathsf{zero}\rangle, y{:}\langle \mathsf{cont}\ \mathsf{fst1}\rangle \not\rightsquigarrow$$

Because $(x_3{:}\langle \mathsf{retn}\ \mathsf{zero}\rangle, y{:}\langle \mathsf{cont}\ \mathsf{fst1}\rangle)$ is not a final state – only states consisting of a single $\mathsf{retn}\ulcorner e \urcorner$ proposition are final – this is a stuck state and not a completed computation.

Thus, for deterministic semantics, both coinductive big-step operational semantics and the operationalization transformation represent ways of reasoning about the difference between non-termination and failure in a natural semantics. Our approach has the advantage of being automatic rather than requiring the definition of a new coinductive relation $e \Uparrow^{\infty}$, though it would presumably be possible to consider synthesizing the definition of $e \Uparrow^{\infty}$ from the definition of $e \Downarrow v$ by an analogue of our operationalization transformation.

In Section 6.4.1, we discuss the advantages that operationalization has in dealing with nondeterministic language features. These advantages do come with a cost when we consider natural semantics specifications that make deterministic choices, which we discuss in Section 6.4.2.

6.4.1 Arbitrary choice and failure

For the purposes of illustration, we will extend the language of expressions with a nondeterministic choice operator $\ulcorner e_1 \,⍰\, e_2 \urcorner = \mathsf{choose}\ulcorner e_1 \urcorner \ulcorner e_2 \urcorner$). The two natural semantics rules for this extension and their (tail-recursion optimized) operationalization are shown in Figure 6.9.

We need to think about the desired semantics of expressions like $(\lambda y.\, y) \,⍰\, \mathsf{z.1}$ – if the first subexpression $(\lambda y.\, y)$ is chosen for evaluation, then the expression evaluates to a value, but if the second subexpression $\mathsf{z.1}$ is chosen, then the evaluation gets stuck. Small-step intuitions about language safety say that this is a possibility we should be able to express, if only to exclude it with an appropriately designed type system and type safety proof. The ordered abstract machine semantics allows us to produce traces where $x{:}\langle \mathsf{eval}\ulcorner (\lambda y.\, y) \,⍰\, \mathsf{z.1}\urcorner\rangle \rightsquigarrow^{*} y{:}\langle \mathsf{eval}\ (\mathsf{lam}\ \lambda y.y)\rangle$ and

where $x{:}\langle \mathsf{eval}\ulcorner(\lambda y.\, y) \mathbin{?} \mathsf{z}.1\urcorner\rangle \rightsquigarrow^* (x'{:}\langle \mathsf{retn}\ \mathsf{zero}\rangle, y{:}\langle \mathsf{cont}\ \mathsf{fst1}\rangle) \not\rightsquigarrow$ as we would hope. Natural semantics specifications (including coinductive big step operational semantics) merely conclude that $(\lambda y.\, y\, y) \mathbin{?} \mathsf{z}.1 \Downarrow (\lambda y.\, y\, y)$. Capturing the stuck behavior in this situation would require defining an extra inductive judgment capturing all the situations where e can get stuck, which is verbose and error-prone [Har12, Section 7.3].

Our ability to reason about evaluations that go wrong is an artifact of the fact that SLS allows us to talk about traces T that represent the process of incomplete proof search in addition to talking about complete proofs. A trace that reaches a state that is not a final $\mathsf{retn}\ulcorner v\urcorner$ state but that cannot step further, like $T :: (x{:}\langle \mathsf{eval}\ulcorner(\lambda y.\, y) \mathbin{?} \mathsf{z}.1\urcorner\rangle) \rightsquigarrow^* (x'{:}\langle \mathsf{retn}\ \mathsf{zero}\rangle, y{:}\langle \mathsf{cont}\ \mathsf{fst1}\rangle)$, corresponds to a point at which backward chaining proof search must backtrack (in a backtracking interpreter) or immediately fail (in a flat resolution interpreter). The trace above corresponds semantically to a failing or going-wrong evaluation, implying that backtracking is not the correct choice. Such an evaluation *ought* to fail, and therefore faithfully capturing the semantics of nondeterministic choice $e_1 \mathbin{?} e_2$ with a natural semantics requires us to use a particular operational interpretation of the natural semantics that is based on non-backtracking backward chaining (flat resolution). The operationalization transformation allows us to concretize this particular operational strategy with traces.

6.4.2 Conditionals and factoring

It is great that we're able to reason about nondeterministic specifications in the output of the operationalization transformation! However, a complication arises if we try to encode a Mini-ML feature that was conspicuously missing from Figure 6.8: the elimination form for natural numbers $\ulcorner \mathsf{case}\ e\ \mathsf{of}\ \mathsf{z} \Rightarrow e_z \mid \mathsf{s}\, x \Rightarrow e_s\urcorner = \mathsf{case}\ulcorner e\urcorner\ulcorner e_z\urcorner(\lambda x.\ulcorner e_s\urcorner)$. The usual natural semantics for case analysis look like this:

$$\frac{e \Downarrow \mathsf{z} \quad e_z \Downarrow v}{(\mathsf{case}\ e\ \mathsf{of}\ \mathsf{z} \Rightarrow e_z \mid \mathsf{s}\, x \Rightarrow e_s) \Downarrow v}\ \mathsf{ev/casez} \qquad \frac{e \Downarrow \mathsf{s}\, v' \quad [v'/x]e_s \Downarrow v}{(\mathsf{case}\ e\ \mathsf{of}\ \mathsf{z} \Rightarrow e_z \mid \mathsf{s}\, x \Rightarrow e_s) \Downarrow v}\ \mathsf{ev/cases}$$

If we operationalize this specification directly, we get an ordered abstract machine shown in Figure 6.10 before defunctionalization and in Figure 6.11 after defunctionalization. This specification is nondeterministic much as the specification of $e_1 \mathbin{?} e_2$ was: we can evaluate a case expression either with rule ev/casez, which effectively predicts that the answer will be zero, or with rule ev/cases, which effectively predicts that the answer will be the successor of some value. But this means that it is possible to get stuck while executing an intuitively type-safe expression if we predict the wrong branch:

$$\begin{aligned} & x_1{:}\langle \mathsf{eval}\ (\mathsf{case}\ \mathsf{zero}\ e_z\ \lambda x.\, e_s)\rangle \\ \rightsquigarrow\ & x_2{:}\langle \mathsf{eval}\ \mathsf{zero}\rangle, y_2{:}\langle \mathsf{cont}\ (\mathsf{cases}\ \lambda x.\, e_s)\rangle \\ \rightsquigarrow\ & x_3{:}\langle \mathsf{retn}\ \mathsf{zero}\rangle, y_2{:}\langle \mathsf{cont}\ (\mathsf{cases}\ \lambda x.\, e_s)\rangle \\ \not\rightsquigarrow\ & (!!!) \end{aligned}$$

This is a special case of a well-known general problem: in order for us to interpret the usual natural semantics specification (rules ev/casez and ev/cases above) as an operational specification, we need backtracking. With backtracking, if we try to evaluate e using one of the rules and

```
ev/casez: eval (case E Ez (\x. Es x))
           >-> {eval E * (retn zero >-> {eval Ez})}.

ev/cases: eval (case E Ez (\x. Es x))
           >-> {eval E * (All V'. retn (succ V') >-> {eval (Es V')})}.
```

Figure 6.10: Problematic semantics of case analysis (not defunctionalized)

```
ev/casez:  eval (case E Ez (\x. Es x))
            >-> {eval E * cont (casez Ez)}.
ev/casez1: retn zero * cont (casez Ez)
            >-> {eval Ez}.

ev/cases:  eval (case E Ez (\x. Es x))
            >-> {eval E * cont (cases \x. Es x)}.
ev/cases1: retn (succ V) * cont (cases \x. Es x)
            >-> {eval (Es V)}.
```

Figure 6.11: Problematic semantics of case analysis (defunctionalized)

fail, a backtracking semantics means that we will apply the other rule, *re-evaluating* the scrutinee e to a value. Backtracking is therefore necessary for a correct interpretation of the standard rules above, even though it is incompatible with a faithful account of nondeterministic choice! Something must give: we can either give up on interpreting nondeterministic choice correctly or we can change the natural semantics for case analysis. Luckily, the second option is both possible and straightforward.

It is possible to modify the natural semantics for case analysis to avoid backtracking by a transformation called *factoring*. Factoring has been expressed by Poswolsky and Schürmann as a transformation on functional programs in a variant of the Delphin programming language [PS03]. It can also be seen as a generally-correct *logical* transformation on Prolog, λProlog, or Twelf specifications, though this appears to be a folk theorem. We factor this specification by creating a new judgment $(v', e_z, x.e_s) \Downarrow^? v$ that is mutually recursive with the definition of $e \Downarrow v$.

$$\frac{e \Downarrow v' \quad (v', e_z, x.e_s) \Downarrow^? v}{(\mathsf{case}\, e\, \mathsf{of}\, \mathsf{z} \Rightarrow e_z \mid \mathsf{s}\, x \Rightarrow e_s) \Downarrow v}\ \mathsf{ev/case}$$

$$\frac{e_z \Downarrow v}{(\mathsf{z}, e_z, x.e_s) \Downarrow^? v}\ \mathsf{casen/z} \qquad \frac{[v'/x]e_s \Downarrow v}{(\mathsf{s}\, v', e_z, x.e_s) \Downarrow^? v}\ \mathsf{casen/s}$$

This natural semantics specifications is provably equivalent to the previous one we gave when rules are interpreted as inductive definitions. Not only are the same judgments $e \Downarrow v$ derivable, the set of possible derivations are isomorphic, and it is possible to use the existing metatheoretic machinery of Twelf to verify this fact. In fact, the two specifications are also equivalent if we understand natural semantics in terms of the success or failure backtracking proof search, though the factored presentation avoids redundantly re-evaluating the scrutinee. It is only when

```
ev/case:  eval (case E Ez (\x. Es x))
           >-> {eval E *
                (All V'. retn V' >-> {casen V' Ez (\x. Es x)})}.

casen/z:  casen zero Ez (\x. Es x) >-> {eval Ez}.
casen/s:  casen (succ V') Ez (\x. Es x) >-> {eval (Es V')}.
```

Figure 6.12: Revised semantics of case analysis (not defunctionalized)

```
ev/case:  eval (case E Ez (\x. Es x))
           >-> {eval E * cont (case1 Ez (\x. Es x))}.

ev/case1: retn V' * cont (case1 Ez (\x. Es x))
           >-> {casen V' Ez (\x. Es x)}.

casen/z:  casen zero Ez (\x. Es x) >-> {eval Ez}.
casen/s:  casen (succ V') Ez (\x. Es x) >-> {eval (Es V')}.
```

Figure 6.13: Revised semantics of case analysis (defunctionalized)

we interpret the natural semantics specification through the lens of non-backtracking backward chaining (also called flat resolution) that the specifications differ.

The operationalization of these rules is shown in Figure 6.12 before defunctionalization and in Figure 6.13 after defunctionalization. In those figures, the standard evaluation judgment $e \Downarrow v$ is given the now-familiar evaluation and return predicates eval and retn. The judgment $(v', e_z, x.e_s) \Downarrow^? v$ is given the evaluation predicate casen, and *shares* the return predicate retn with the judgment $e \Downarrow v$. This is a new aspect of operationalization. It is critical for us to assign each predicate $\mathsf{a_c}$ uniquely to an evaluation predicate eval_a – without this condition, soundness (Theorem 6.4) would fail to hold. However, we never rely on $\mathsf{a_c}$ being uniquely assigned a return predicate. When return predicates that have the same type are allowed to overlap, it enables the tail-call optimization described in Section 6.1.3 to apply even when the tail call is to a different procedure. This, in turn, greatly simplifies Figures 6.12 and 6.13.

6.4.3 Operationalization and computation

When we described the semantics of nondeterministic choice $e_1 \textcircled{?} e_2$, our operational intuition was to search either for a value such that $e_1 \Downarrow v$ or a value such that $e_2 \Downarrow v$. This implies an operational interpretation of natural semantics as flat resolution as opposed to backward chaining with backtracking. Maintaining this non-backtracking intuition means that some natural semantics specifications, such as those for case analysis, need to be revised. It is a folk theorem that such revisions are always possible by factoring [PS03]; therefore, we can conclude that *in the context of natural semantics specifications* the form of deductive computation (Section 4.6.1) that we are most interested in is flat resolution.

Under the operationalization transformation, traces represent the internal structure of proof

search, and a non-extendable (and non-final) trace represents a situation in which backward chaining search backtracks and where flat resolution search gives up. If we search for a trace $(x{:}\langle \mathsf{eval} \ulcorner e \urcorner \rangle) \rightsquigarrow^* (y{:}\langle \mathsf{retn} \ulcorner v \urcorner \rangle)$ in an operationalized specification using committed-choice forward chaining, the operational behavior will coincide with the behavior of flat resolution in the original specification. Alternatively, if we take the exhaustive search interpretation of an operationalized specification and attempt to answer, one way or the other, whether a trace of the form $(x{:}\langle \mathsf{eval} \ulcorner e \urcorner \rangle) \rightsquigarrow^* (y{:}\langle \mathsf{retn} \ulcorner v \urcorner \rangle)$ can be constructed, then the operational behavior of the interpreter will coincide with the behavior of backtracking backward chaining in the original specification.

Therefore, operationalization can be said to connect backward chaining in the deductive fragment of SLS to forward chaining in the concurrent fragment of SLS. More precisely, operationalization both connects flat resolution to committed-choice forward chaining and connects backtracking backward chaining to the exhaustive search interpretation of forward chaining.

6.5 Exploring the richer fragment

Work by Danvy et al. on the functional correspondence has generally been concerned with exploring tight correspondences between different styles of specification. However, as we discussed in Section 5.3, one of the main reasons the logical correspondence in SLS is interesting is because, once we translate from a less expressive style (natural semantics) to a more expressive style (ordered abstract machine semantics), we can consider new modular extensions in the more expressive style that were not possible in the less expressive style. As we discussed in Chapter 1, extending a natural semantics with state requires us to revise every existing rule, whereas a SSOS specification can be extended in a modular fashion: we just insert new rules that deal with state. The opportunities for modular extension are part of what distinguishes the logical correspondence we have presented from the work by Hannan and Miller [HM92] and Ager [Age04]. Both of those papers translated natural semantics into a syntactic specification of abstract machines; such specifications are not modularly extensible to the degree that concurrent SLS specifications are.

The ordered abstract machine style of specification facilitates modular extension with features that involve *state* and *parallel evaluation*. We have already seem examples of the latter: the operationalization translation (as extended in Section 6.1.4) can put a natural semantics specification into logical correspondence with either a sequential ordered abstract machine semantics or a parallel ordered abstract machine semantics, and our running example evaluates pairs in parallel. In this section, we will consider some other extensions, focusing on stateful features like mutable storage (Section 6.5.1) and call-by-need evaluation (Section 6.5.2). We will also discuss the semantics of recoverable failure in Section 6.5.4. The presentation of recoverable failure will lead us to consider a point of non-modularity: if we want to extend our language flexibly with non-local control features like recoverable failure, the parallel operationalization translation will make this difficult or impossible. A more modular semantics of parallel evaluation will be presented in Section 7.2.1.

This section will present extensions to the sequential, flat abstract machine for parallel evaluation presented in Figure 6.6. We first presented most of these specifications in [PS09].

```
cell: mutable_loc -> exp -> prop lin.

ev/loc:  eval (loc L)
          >-> {retn (loc L)}.

ev/ref:  eval (ref E)
          >-> {eval E * cont ref1}.
ev/ref1: retn V * cont ref1
          >-> {Exists l. $cell l V * retn (loc l)}.

ev/get:  eval (get E)
          >-> {eval E * cont get1}.
ev/get1: retn (loc L) * cont get1 * $cell L V
          >-> {retn V * $cell L V}.

ev/set:  eval (set E1 E2)
          >-> {eval E1 * cont (set1 E2)}.
ev/set1: retn (loc L) * cont (set1 E2)
          >-> {eval E2 * cont (set2 L)}.
ev/set2: retn V2 * cont (set2 L) * $cell L _
          >-> {retn unit * $cell L V2}.
```

Figure 6.14: Semantics of mutable storage

6.5.1 Mutable storage

Classic stateful programming languages feature mutable storage, which forms the basis of imperative algorithms. We will consider ML-style references, which add four new syntax forms to the language. The first three create ($\ulcorner \mathsf{ref}\, e \urcorner = \mathsf{ref} \ulcorner e \urcorner$), dereference ($\ulcorner !e \urcorner = \mathsf{get} \ulcorner e \urcorner$), and update ($\ulcorner e_1 := e_2 \urcorner = \mathsf{set} \ulcorner e_1 \urcorner \ulcorner e_2 \urcorner$) dynamically allocated cells in the heap. The fourth, $\mathsf{loc}\, l$, is a value that represents pointers to allocated memory. The term l is of a type mutable_loc that has no constructors; locations l can only be allocated at runtime. We also introduce a new *linear* atomic proposition $\mathsf{cell}\, l\, v$ representing a piece of allocated memory (at location l) and its contents (the value v). (Recall from Section 4.5 that, in the ASCII notation for SLS, these linear propositions are written `$cell L V`, as `$` is used as the ASCII representation of the mobile modality ¡A.)

A collection of linear propositions acts much like one of Jeannin and Kozen's *capsules* [JK12], but unlike capsule formulations (and most other existing specification frameworks), we can introduce state in this way without revising any of the rules introduced in the previous section. Without mutable state or parallel computation, specifications such as the one in Figure 6.8 maintain the invariant that the process state Δ is made up of either a $\mathsf{eval}\, e$ proposition or a $\mathsf{retn}\, v$ proposition to the left of some number of $\mathsf{cont}\, f$ propositions. Once we add mutable state, the first-order (or LF) context Ψ, which has been empty in the SSOS semantics we have considered so far, becomes non-empty. We maintain the invariant that the context has one mutable location for each allocated cell:

$$(l_1{:}\mathsf{mutable\_loc}, \ldots, l_n{:}\mathsf{mutable\_loc};\ \ x_1{:}\langle \mathsf{cell}\, l_1\, v_1 \rangle, \ldots, x_n{:}\langle \mathsf{cell}\, l_n\, v_n \rangle, \Delta)$$

where Δ has the form described before. Because the $\mathsf{cell}\, l_i\, v_i$ propositions are mobile, the order of cells is irrelevant, as is the placement of these cells relative to the control structure encoded in the context Δ.

The semantics of mutable storage are presented in Figure 6.14. Rule ev/get in that figure takes an expression $\mathsf{get}\, E$ and evaluates E to a value of the form $\mathsf{loc}\, L$. After that, rule ev/get1 takes the (unique, by invariant) cell associated with the location L, reads its value, and restores the cell to the context. The synthetic transition associated with ev/get1 is as follows:

$$(\Psi;\ \ \Theta\{\!\{x{:}\langle \mathsf{retn}\,(\mathsf{loc}\, l)\rangle,\ \ y{:}\langle \mathsf{cont}\,\mathsf{get1}\rangle,\ \ z{:}\langle \mathsf{cell}\, l\, v\rangle\}\!\}) \\ \rightsquigarrow (\Psi;\ \ \Theta\{w{:}\langle \mathsf{retn}\, v\rangle\ ord,\ \ z'{:}\langle \mathsf{cell}\, l\, v\rangle\ eph\})$$

Again, it is critical, when reading this transition, to account for the fact that retn and cont are ordered predicates but cell is a mobile predicate.

The set rules are similar, except that we also evaluate a new value v_2 and restore that value to the process state instead of the value previously contained in the cell. We mention $\mathsf{cell}\, l\ \_$ in the premise of ev/set2 only to consume the old cell associated with l before we replace it with something new. If multiple parts of a process state are trying to consume the same resource in order to set the value of a cell concurrently, we have a race condition; this possibility is discussed below.

Finally, the ref rules evaluate the subexpression to a value v and then, in rule ev/ref1, allocate a new cell to hold that value. This new cell, according to our context invariant, needs to be associated with a new variable l, which we generate with existential quantification in the head of the ev/ref1 rule. The synthetic transition associated with ev/ref1 therefore has an extended first-order context after the transition:

$$(\Psi;\ \ \Theta\{\!\{x{:}\langle \mathsf{retn}\, v\rangle,\ \ y{:}\langle \mathsf{cont}\,\mathsf{ref1}\rangle\}\!\}) \\ \rightsquigarrow (\Psi, l{:}\mathsf{mutable\_loc};\ \ \Theta\{w{:}\langle \mathsf{retn}\,(\mathsf{loc}\, l)\rangle\ ord,\ \ z{:}\langle \mathsf{cell}\, l\, v\rangle\ eph\})$$

Existential angst

Our semantics of mutable storage uses existential quantification as a symbol generator to conjure up new locations. However, it is important to remember that LF variables in Ψ are defined by substitution, so if there is a step $(\Psi, l_1{:}\mathsf{loc}, l_2{:}\mathsf{loc}; \Delta) \rightsquigarrow (\Psi, l_1{:}\mathsf{loc}, l_2{:}\mathsf{loc}; \Delta')$, it must also be the case that $(\Psi, l_1{:}\mathsf{loc}; [l_1/l_2]\Delta) \rightsquigarrow (\Psi, l_1{:}\mathsf{loc}; [l_1/l_2]\Delta')$. Therefore, there can be no SLS proposition or synthetic transition that holds only if two variables are distinct, since by definition the same proposition or synthetic transition would hold when we unified the two variables. This, in turn, means our specification of Mini-ML with mutable references *cannot* be further extended to include the tests for reference equality that languages like Standard ML or OCaml have, since locations are described only as variables.

One workaround to this problem is to maintain an association between distinct variables l and distinct concrete terms (usually distinct natural numbers) in the process state. It is possible to use generative signatures to enforce that all well-formed states associate distinct variables with distinct concrete terms, as described in Section 9.4.4. In such a specification, we can use inequality of the concrete terms as a proxy for inequality of the variables. It will still be the case

that unifying distinct variables preserves transitions, but we can ensure that any process state obtained by unifying distinct variables is not well-formed according to the generative invariant.

I believe that a substructural treatment of nominal quantification could be incorporated into SLS and would allow for locations to be handled in a more satisfying way along the lines of proposals by Cheney and Harper [Che12, Har12]. This extension to the SLS framework is beyond the scope of this thesis, however. Luckily, aside from being unable to elegantly represent tests for pointer inequality or the entirety of Harper's Modernized Algol [Har12, Chapter 35], we will not miss name generation facilities much in the context of this thesis. One of the most important use cases of name generation and nominal abstraction is in reasoning *about* logical specifications within a uniform logic [GMN11], and this thesis does not consider a uniform *meta*logic for SLS specifications.

Race conditions

Because ordered abstract machine semantics allow us to add both state and parallelism to specifications, the issue of race conditions, which arise whenever there is both concurrency and state, should be briefly addressed. Fundamentally, SLS and SSOS specifications have no notion of atomicity beyond the one provided by focusing and synthetic inference rules, and so race conditions can arise.

$$
\begin{aligned}
(l : \mathsf{mutable\_loc};\ & x_1{:}\langle \mathsf{cell}\, l\, \mathsf{zero}\rangle\ \mathit{eph}, \\
& x_2{:}\langle \mathsf{retn}\ (\mathsf{succ}\, \mathsf{zero})\rangle\ \mathit{ord},\ \ x_3{:}\langle \mathsf{cont}\ (\mathsf{set2}\, l)\rangle\ \mathit{ord}, \\
& x_4{:}\langle \mathsf{retn}\ (\mathsf{succ}\ (\mathsf{succ}\, \mathsf{zero}))\rangle\ \mathit{ord},\ \ x_5{:}\langle \mathsf{cont}\ (\mathsf{set2}\, l)\rangle\ \mathit{ord}, \\
& x_6{:}\langle \mathsf{cont}\, \mathsf{pair1}\rangle\ \mathit{ord})
\end{aligned}
$$

Figure 6.15: A racy process state

A process state that starts out containing only $\mathsf{eval}\ulcorner(\lambda x.\langle \mathsf{set}\, x\, (\mathsf{s}\, \mathsf{z}), \mathsf{set}\, x\, (\mathsf{s}(\mathsf{s}\, \mathsf{z}))\rangle)(\mathsf{ref}\, \mathsf{z})\urcorner$, for example, can evaluate to the process state in Figure 6.15. Two different transitions out of this state can both manipulate the data associated with the mutable location l – this is a race condition. Reasoning about race conditions (and possibly precluding them from well-formed specifications) is not within the scope of this thesis. The applicability of generative invariants discussed in Chapter 9 to race conditions is, however, certainly an interesting topic for future work. If we set up the semantics such that a situation like the one above could nondeterministically transition to an ill-formed error state, then it would not be possible to prove the preservation of the generative invariant unless the well-formedness criteria expressed by that invariant precluded the existence of race conditions.

Let us consider what happens when we operationalize a natural semantics that uses state. Recall that the addition of an imperative counter to the natural semantics for semantics for CBV evaluation was presented as a non-modular extension, because we had to revise the existing rules

for functions and application as follows:

$$\frac{}{(\lambda x.e, \underline{n}) \Downarrow (\lambda x.e, \underline{n})} \qquad \frac{(e_1, \underline{n}) \Downarrow (\lambda x.e, \underline{n}_1) \quad (e_2, \underline{n}_1) \Downarrow (v_2, \underline{n}_2) \quad ([v_2/x]e_2, \underline{n}_2) \Downarrow (v, \underline{n}')}{(e_1\, e_2, \underline{n}) \Downarrow (v, \underline{n}')}$$

In the pure CBV specification, the two premises $e_1 \Downarrow \lambda x.e$ and $e_2 \Downarrow v_2$ could be treated as independent and could be made parallel by the operationalization transformation, but the two premises $(e_1, \underline{n}) \Downarrow (\lambda x.e, \underline{n}_1)$ and $(e_2, \underline{n}_1) \Downarrow (v_2, \underline{n}_2)$ are not: the first premise binds $\underline{n}_1$, which appears as an input argument to the second premise. Therefore, from the perspective of operationalization, parallel evaluation and state are simply incompatible: having state in the original specification will preclude parallel evaluation (and therefore race conditions) in the operationalized specification.

Ordered abstract machine semantics allow for the modular composition of mutable storage and parallel evaluation, in that the original specifications can be simply composed to give a semantically meaningful result. However, composing mutable storage and parallel evaluation leads to the possibility of race conditions (which can be represented in SLS), indicating that this composition is not always a good idea if we want to avoid race conditions. Adding state to a natural semantics specification, on the other hand, will force operationalization to produce an ordered abstract machine without parallelism.

6.5.2 Call-by-need evaluation

Mutable references were an obvious use of ambient state, and we were able to extend the ordered abstract machine obtained from the operationalization transformation by simply adding new rules for mutable references (though this did introduce the possibility of race conditions). Another completely modular extension to our (now stateful) Mini-ML language is *call-by-need* evaluation. The basic idea in call-by-need evaluation is that an expression is not evaluated eagerly; rather, instead, it is stored until the value of that expression is demanded. Once a value is needed, it is computed and the value of that computation is memoized; therefore, a suspended expression will be computed at most once.

This section considers two rather different implementations of by-need evaluation: the first, recursive suspensions, presents itself to the programmer as a different sort of fixed-point operator, and the second, lazy call-by-need, presents itself to the programmer as a different sort of function. Both approaches to lazy evaluation are based on Harper's presentation [Har12, Chapter 37].

Recursive suspensions

Recursive suspensions (Figure 6.16) replace the fixed-point operator $\mathsf{fix}\, x.e$ with a thunked expression $\ulcorner \mathsf{thunk}\, x.e \urcorner = \mathsf{thunk}\, (\lambda x. \ulcorner e \urcorner)$. Whereas the fixed-point operator returns a value (or fails to terminate), thunked expressions always immediately return a value $\mathsf{issusp}\, l$, where l is a location of type bind_loc. This location is initially associated with a linear atomic proposition $\mathsf{susp}\, l\, (\lambda x. \ulcorner e \urcorner)$ (rule ev/thunk).

When we apply the force operator to an expression that returns $\mathsf{issusp}\, l$ for the first time, the location l stops being associated with a linear atomic proposition of the form $\mathsf{susp}\, l\, (\lambda x. \ulcorner e \urcorner)$ and

```
susp: bind_loc -> (exp -> exp) -> prop lin.
blackhole: bind_loc -> prop lin.
bind: bind_loc -> exp -> prop pers.

ev/susp:    eval (issusp L) >-> {retn (issusp L)}.

ev/thunk:   eval (thunk \x. E x)
             >-> {Exists l. $susp l (\x. E x) * retn (issusp l)}.

ev/force:   eval (force E)
             >-> {eval E * cont force1}.

ev/force1a: retn (issusp L) * cont force1 * $susp L (\x. E' x)
             >-> {eval (E' (issusp L)) * cont (bind1 L) *
                  $blackhole L}.

ev/force2a: retn V * cont (bind1 L) * $blackhole L
             >-> {retn V * !bind L V}.

#| STUCK -  retn (issusp L) * cont force1 * $blackhole L >-> ??? |#

ev/force1b: retn (issusp L) * cont force1 * !bind L V
             >-> {retn V}.
```

Figure 6.16: Semantics of call-by-need recursive suspensions

becomes associated with a linear atomic proposition of the form blackhole l (rule ev/force1a). This blackhole l proposition can be used to detect when an expression tries to directly reference its own value in the process of computing to a value. In this example, such a computation will end up stuck, but the comparison with fix $x.x$ suggests that the semantically correct option is failing to terminate instead. A rule with the premise retn (issusp l) • cont force1 • blackhole l could instead be used to loop endlessly or signal failure. This possibility is represented in Figure 6.16 by the commented-out rule fragment – `#| this is comment syntax |#`.

Once a suspended expression has been fully evaluated (rule ev/force2b), the black hole is removed and the location l is persistently associated with the value v; future attempts to force the same suspended expression will trigger rule ev/force1b instead of ev/force1a and will immediately return the memoized value.

The last four rules in Figure 6.16 (and the one commented-out rule fragment) are all part of one multi-stage protocol. It may be enlightening to consider the *re*functionalization of Figure 6.16 presented in Figure 6.17. This rule has a conjunctive continuation (using the additive conjunction connective $A^- \,\&\, B^-$) with one conjunct for two of the three atomic propositions a bind_loc can be associated with: the linear proposition susp l $(\lambda x.e)$, the linear proposition blackhole l (which cannot be handled by the continuation and so will result in a stuck state), and the persistent proposition bind l v.

```
ev/force: eval (force E)
           >-> {eval E *
                ((All L. All E'.
                  retn (issusp L) * $susp L (\x. E' x)
                   >-> {eval (E' (issusp L)) * $blackhole L *
                        (All V. retn V * $blackhole L
                          >-> {retn V * !bind L V})})
     #| STUCK -  & (All L. retn (issusp L) * $blackhole L >-> ???) |#
                & (All L. All V.
                   retn (issusp L) * !bind L V >-> {retn V}))}.
```

Figure 6.17: Semantics of call-by-need recursive suspensions, refunctionalized

```
susp': exp -> exp -> prop lin.
blackhole': exp -> prop lin.
bind': exp -> exp -> prop pers.

ev/lazylam: eval (lazylam \x. E x) >-> {retn (lazylam \x. E x)}.

ev/applazy: retn (lazylam \x. E x) * cont (app1 E2)
             >-> {Exists x:exp. eval (E x) * $susp' x E2}.

ev/susp':   eval X * susp' X E
             >-> {$blackhole' X * eval E * cont (bind1' E)}.

ev/susp1':  retn V * cont (bind1' X) * $blackhole' X
             >-> {retn V * !bind' X V}.

ev/bind':   eval X * !bind' X V >-> {retn V}.
```

Figure 6.18: Semantics of lazy call-by-need functions

Lazy evaluation

Recursive suspensions must be forced explicitly; an alternative to recursive suspensions, which uses very similar specification machinery but that presents a different interface to the programmer, is lazy call-by-need function evaluation. Lazy evaluation better matches the semantics of popular call-by-need languages like Haskell. For this semantics, we will not create a new abstract location type like mutable_loc or bind_loc; instead, we will associate suspended expressions (and black holes and memoized values) with free expression variables of type exp.

We can treat lazy call-by-need functions $(\mathsf{lazylam}\ \lambda x.e)$ as an extension to the language that already includes call-by-value functions $(\mathsf{lam}\ \lambda x.e)$ and application; this extension is described in Figure 6.18. Lazy functions are values (rule ev/lazylam), but when a lazy function is returned to a frame $\ulcorner \square\, e_2 \urcorner = \mathsf{app1} \ulcorner e_2 \urcorner$, we do not evaluate $\ulcorner e_2 \urcorner$ to a value immediately. Instead, we create a free variable x of type exp and substitute that into the lazy function.

Free variables x are now part of the language of expressions that get evaluated, though they

```
ev/envlam:  eval (envlam \x. E x) >-> {retn (envlam \x. E x)}.

ev/appenv1: retn (envlam \x. E x) * cont (app1 E2)
             >-> {Exists x. eval E2 * cont (app2' x (E x))}.

ev/appenv2: retn V2 * cont (app2' X E)
             >-> {eval E * !bind' X V2}.
```

Figure 6.19: Environment semantics for call-by-value functions

are not in the language of values that get returned. Therefore, we need some way of evaluating free variables. This is handled by the three rules ev/susp′, ev/susp1′, and ev/bind′ as before. Each free expression variable x is either associated with a unique linear atomic proposition $\mathsf{susp}'\, x\, e_2$, a black hole $\mathsf{blackhole}'\, x$, or a persistent binding $\mathsf{bind}'\, x\, v$.

As a final note, call-by-need evaluation is semantically equivalent to call-by-name evaluation in a language that does not otherwise use state. Unevaluated suspensions can therefore be ignored if they are not needed. For this reason, if SLS were extended with an affine modality, it would be quite reasonable to view susp and susp′ as affine atomic propositions instead of linear atomic propositions. However, as long as we restrict ourselves to considering traces rather than complete derivations, the differences between affine and linear propositions are irrelevant.

6.5.3 Environment semantics

Toninho, Caires, and Pfenning have observed that call-by-need and call-by-value can both be seen in a larger family of *sharing* evaluation strategies (if and when the argument to $\lambda x.e$ is evaluated, the work of evaluating that argument to a value is shared across all occurrences of x). Call-by-name, in contrast, is called a *copying* evaluation strategy, since the unevaluated argument of $\lambda x.e$ is copied to all occurrences of x [TCP12]. This relationship between the lazy call-by-need semantics from Figure 6.18 and call-by-value is better presented by giving a variant of what we called an *environment semantics* in [PS09].

As with the lazy call-by-name semantics, we introduce the environment semantics by creating a new function value $\mathsf{envlam}\,(\lambda x.\, e)$ instead of reinterpreting the existing function expression $\mathsf{lam}\,(\lambda x.\, e)$. When a value of the form $\mathsf{lazylam}\,(\lambda x.\, e)$ was returned to a frame $\mathsf{app1}\, e_2$ in rule ev/applazy from Figure 6.18, we immediately created a new expression variable x, suspended the argument e_2, and scheduled the function body for evaluation. When a value of the form $\mathsf{envlam}\,(\lambda x.\, e)$ is returned to a frame $\mathsf{app1}\, e_2$ frame in rule ev/appenv1 in Figure 6.19, we likewise create the new expression variable x, but we suspend the *function body* in a frame $\mathsf{app2}'\, x\, e$ that also records the new expression variable x and schedule the *argument* for evaluation. Immediately evaluating the argument is, of course, exactly how call-by-value evaluation is performed; this is what makes environment semantics equivalent to call-by-value semantics. Then, when the evaluated function argument v_2 is returned to that frame (rule ev/appenv2), we create the same persistent binding $\mathsf{bind}\, x\, v_2$ that was generated by rule ev/susp1′ in Figure 6.18 and proceed to evaluate the function body. Upon encountering the free variable x in the course of evaluation, the same rule ev/bind′ from Figure 6.18 will return the right value.

```
error: prop ord.
handle: exp -> prop ord.

ev/fail:   eval fail >-> {error}.
ev/error:  error * cont F >-> {error}.

ev/catch:  eval (catch E1 E2) >-> {eval E1 * handle E2}.
ev/catcha: retn V * handle _ >-> {retn V}.
ev/catchb: error * handle E2 >-> {eval E2}.
```

Figure 6.20: Semantics of recoverable failure

This presentation of the environment semantics is designed to look like call-by-need, and so it creates the free variable x early, in rule ev/appenv1. It would be equally reasonable to create the free variable x later, in rule ev/appenv2, which would result in a specification that resembles Figure 6.6 more closely. This is what was done in [PS09] and [SP11a], and we will use a similar specification (Figure 8.3) as the basis for deriving a control flow analysis in Section 8.4.

6.5.4 Recoverable failure

In a standard abstract machine presentation, recoverable failure can be introduced by adding a new state $s = k\blacktriangleleft$ to the existing two ($s = k \triangleright e$ and $s = k \triangleleft v$) [Har12, Chapter 28]. Whereas $k \triangleleft v$ represents a value being returned to the stack, $k\blacktriangleleft$ represents *failure* being returned to the stack; failure is signaled by the expression fail[7] and can be handled by the expression $\ulcorner \mathsf{try}\, e_1\, \mathsf{ow}\, e_2 \urcorner = \mathsf{catch} \ulcorner e_1 \urcorner \ulcorner e_2 \urcorner$.

We can extend *sequential* ordered abstract machines with exceptions in a modular way, as shown in Figure 6.20. Recall that a sequential ordered abstract machine specification is one where there is only one ordered $\mathsf{eval}\, e$ or $\mathsf{retn}\, v$ proposition in the process state to the right of a series of ordered $\mathsf{cont}\, f$ propositions – process states with $\mathsf{eval} \ulcorner e \urcorner$ correspond to states $k \triangleright e$ and process states with $\mathsf{retn} \ulcorner v \urcorner$ correspond to states $k \triangleleft v$.

We introduce two new ordered atomic propositions. The first, error, is introduced by rule ev/fail. A state with an error proposition corresponds to a state $k\blacktriangleleft$ in traditional ordered abstract machine specifications. Errors eat away at any $\mathsf{cont}\, f$ propositions to their right (rule ev/error). The only thing that stops the inexorable march of an error is the special ordered atomic proposition $\mathsf{handle}\, e$ that is introduced in rule ev/catch when we evaluate the handler.

This is one case where the use of defunctionalized specifications – and, in particular, our decision to defunctionalize with a single $\mathsf{cont}\, f$ proposition instead of inventing a new ordered atomic proposition at every step (Section 6.2.3) – gives us a lot of expressive power. If we wanted to add exceptions to the higher-order specification of Mini-ML, we would have to include the possibility of an exceptional outcome in every individual rule. For instance, this would be the rule for evaluating $\ulcorner \mathsf{s}\, e \urcorner = \mathsf{succ} \ulcorner e \urcorner$:

[7]Failure could also be introduced by actions like as dividing by zero or encountering the black hole in the commented-out case of Figure 6.16.

```
ev/succ: eval (succ E)
          >-> {eval E *
               ((All V. retn V >-> {retn (succ V)})
                & (error >-> {error}))}.
```

Instead, this case is handled generically by rule ev/error in Figure 6.20, though the above specification is what we would get if we were to refunctionalize the defunctionalized specification of core Mini-ML from Figure 6.8 extended with rule ev/error.

Failures and the parallel translation

Our semantics of recoverable failure composes reasonably well with stateful features, though arguably in call-by-need evaluation it is undesirable that forcing a thunk can lead to errors being raised in a non-local fashion. However, recoverable failure does not compose well with parallel semantics as we have described them.

We assume, in rule ev/error, that we can blindly eliminate cont f frames with the ev/error rule. If we eliminate the cont pair1 frame from Figure 6.8 in this way, it breaks the invariant that the ordered propositions represent a branching tree written down in postfix. Recall that, without exceptions, a piece of process state in the process of evaluating $\ulcorner\langle e_1, e_2\rangle\urcorner = \mathsf{pair}\ulcorner e_1\urcorner\ulcorner e_2\urcorner$ has the following form:

$$(\text{subgoal: evaluating } e_1), (\text{subgoal: evaluating } e_2), y{:}\langle \mathsf{cont}\ \mathsf{pair1}\rangle$$

If the second subgoal evaluating e_2 signals an error, that error will immediately propagate to the right, orphaning the first subgoal. Conversely, if the first subgoal signals an error, that error will have to wait until the first subgoal completes: SLS specifications are local, and there is no local way for the first subgoal to talk about its continuation $y{:}\langle \mathsf{cont}\ \mathsf{pair1}\rangle$ because an arbitrary amount of stuff (the representation of the second subgoal) is in the way. This seems to force us into treating parallel evaluations asymmetrically: if e_{raise} signals failure and e_{loop} loops forever, then the two Mini-ML pair expressions $\langle e_{raise}, e_{loop}\rangle$ and $\langle e_{loop}, e_{raise}\rangle$ are observably different. That is arguably bad, though if we switch the relative position of e_1 and e_2 in the context, it can also be seen as an implementation of the sequential semantics for exceptions followed by Manticore [FRR08].

A fix is to modify defunctionalization to group similar propositions rather than grouping all propositions into the single proposition cont. Specifically, we defunctionalize *sequential* propositions of the form $\forall\overline{x}.\,\mathsf{retn}\ v \rightarrowtail \{\ldots\}$ using one ordered atomic proposition cont f and defunctionalize *parallel* propositions of the form $\forall\overline{x}.\,\mathsf{retn}\ v_1 \bullet \mathsf{retn}\ v_2 \rightarrowtail \{\ldots\}$ using a different ordered atomic proposition cont2 f. This lets us write rules that treat parallel continuations generically and that only return errors when *both* sub-computations have completed and at least one has signaled an error:

```
ev/errerr:  error * error * cont2 _ >-> {error}.
ev/errretn: error * retn _ * cont2 _ >-> {error}.
ev/retnerr: retn _ * error * cont2 _ >-> {error}.
```

This is a big improvement, because parallel pairs are again treated symmetrically. But it's not the way we necessarily wanted to restore symmetry: the evaluation $\langle e_{raise}, e_{loop}\rangle$ and $\langle e_{loop}, e_{raise}\rangle$

will both loop forever, but we might wish for both of them to signal failure. The latter alternative is not expressible in an ordered abstract machine specification.

Part of the problem is that recoverable failure is fundamentally a *control* feature and not a stateful or parallel programming language feature. As a result, it is not easy to handle at the level of ordered abstract machines, because ordered abstract machines do not give the specification author enough access to the control structure. The destination-passing style we consider in the next chapter, on the other hand, will give us sufficient access to control structure.

6.5.5 Looking back at natural semantics

Mutable storage, call-by-need evaluation, and the environment semantics are all modular extensions to the call-by-value specification in Figure 6.6. The extensions are modular because they make essential use of the ambient context available to concurrent SLS specifications, introducing new linear and persistent ordered atomic propositions that can be added to the context (and, in the linear case, removed as well).

For extensions to sequential ordered abstract machines that are only based on extending the state, we can consider what it would mean to reverse-engineer a natural semantics formalism that is as extensible as the resulting ordered abstract machine. The primary judgment of such a specification is not $e \Downarrow v$ as before; rather, the primary judgment becomes $\{e \| \mu\}_\Psi \Downarrow \{v \| \mu\}_{\Psi'}$.[8] The variable contexts Ψ and Ψ' are the same variable contexts that appear in our process states $(\Psi; \Delta)$ and our specifications are expected to maintain the invariant that $\Psi \subseteq \Psi'$. The objects e and v remain syntactic objects adequately encodable in LF, as before, whereas μ is an extensible bag of judgments $\mu = J_1 \otimes \ldots \otimes J_n$ that correspond to the propositions in our linear and persistent context; we treat $\otimes$ as an associative and commutative operator (just like conjunction of linear logic contexts). A new judgment in an SLS specification can be treated as a new member of the syntactic class J. For instance, lazy call-by-need functions as defined in Figure 6.18 use three judgments: $x \hookrightarrow e$ (corresponding to $\mathsf{susp}'\, x\, e$), $x \hookrightarrow \bullet$ (corresponding to $\mathsf{blackhole}'\, x$), and $x \rightarrowtriangle v$ (corresponding to $\mathsf{bind}'\, x\, v$). We can give a statefully-modular natural semantics for call-by-need lazy functions as follows:

$$\frac{}{\{\lambda x.e \| \mu\}_\Psi \Downarrow \{\lambda x.e \| \mu\}_\Psi}$$

$$\frac{\{e_1 \| \mu\}_\Psi \Downarrow \{\lambda x.e \| \mu'\}_{\Psi'} \quad \{e \| x \hookrightarrow e_2 \otimes \mu'\}_{\Psi',x} \Downarrow \{v \| \mu''\}_{\Psi''}}{\{e_1\, e_2 \| \mu\}_\Psi \Downarrow \{v \| \mu''\}_{\Psi''}}$$

$$\frac{\{e \| x \hookrightarrow \bullet \otimes \mu\}_{\Psi,x} \Downarrow \{v \| x \hookrightarrow \bullet \otimes \mu'\}_{\Psi',x}}{\{x \| x \hookrightarrow e \otimes \mu\}_{\Psi,x} \Downarrow \{v \| x \rightarrowtriangle v \otimes \mu'\}_{\Psi',x}}$$

$$\frac{}{\{x \| x \rightarrowtriangle v \otimes \mu\}_{\Psi,x} \Downarrow \{v \| x \rightarrowtriangle v \otimes \mu\}_{\Psi,x}}$$

While the definition of $e \Downarrow v$ could be directly encoded as a deductive SLS specification, the definition of $\{e \| \mu\}_\Psi \Downarrow \{v \| \mu'\}_{\Psi'}$ cannot. Nevertheless, the example above suggests that

[8] The notation $\{e \| \mu\}_\Psi$ is intended to evoke Harper's notation $\nu\Sigma\{e \| \mu\}$, which is used to describe mutable references and lazy evaluation in Chapters 36 and 37 of [Har12]. The critical semantic distinction is that our Ψ contains variables whereas Σ contains proper symbols that are not available in SLS, as discussed in Section 6.5.1.

```
#mode inc + -.
inc: nat -> nat -> prop.

inc/eps: inc eps (c eps b1).
inc/b0: inc (c N b0) (c N b1).
inc/b1: inc (c N b1) (c R b0) <- inc N R.

#mode plus + + -.
plus: nat -> nat -> nat -> prop.
plus/eN: plus eps N N.
plus/Ne: plus N eps N.
plus/b00: plus (c M b0) (c N b0) (c R b0) <- plus M N R.
plus/b01: plus (c M b0) (c N b1) (c R b1) <- plus M N R.
plus/b10: plus (c M b1) (c N b0) (c R b1) <- plus M N R.
plus/b11: plus (c M b1) (c N b1) (c R b0) <- plus M N K <- inc K R.
```

Figure 6.21: Backward-chaining logic program for binary addition

a carefully-defined formalism for statefully-modular natural semantics specifications could be similarly compiled into (or defined in terms of) the operationalization of specifications into SLS.

There is a great deal of work on special-purpose formalisms for the specification and modular extension of operational semantics. The specification above follows Harper's development in [Har12], and Mosses's Modular Structural Operational Semantics (MSOS) is a similar development [Mos04]. Previous work is primarily interested in the modular extension of small-step structural operational semantics specifications rather than big-step natural semantics, though Mosses does discuss the latter. The operationalization transformation applies to SOS specifications (as discussed below in Section 6.6.2), but the result is something besides an ordered abstract machine semantics.

The functional correspondence connects structural operational semantics and abstract machines [Dan08]. A logical correspondence between SOS specifications and ordered abstract machines in SLS might give us insight into a modular formalism for SOS that is defined in terms of concurrent SLS specifications, but this is left for future work.

6.6 Other applications of transformation

Thus far, we have only discussed the application of the operationalization and defunctionalization transformations to natural semantics specifications. However, both transformations are general and can be applied to many different specifications.

In this section, we will consider the meaning of operationalization on three other types of deductive SLS specifications: an algorithmic specification of addition by Pfenning, small-step structural operational semantics specifications, and the natural semantics of Davies' staged computation language $\lambda^{\bigcirc}$. The last two transformations explore the use of *partial* operationalization in which we use the generality of operationalization to transform only some of the predicates in a program.

```
inc: nat -> prop ord.
plus: nat -> nat -> prop ord.
retn: nat -> prop ord.
cont: frame -> prop ord.

inc/eps:    inc eps >-> {retn (c eps b1)}.
inc/b0:     inc (c N b0) >-> {retn (c N b1)}.
inc/b1:     inc (c N b1) >-> {inc N * cont append0}.

plus/eN:    plus eps N >-> {retn N}.
plus/Ne:    plus N eps >-> {retn N}.

plus/b00:   plus (c M b0) (c N b0) >-> {plus M N * cont append0}.
plus/b01:   plus (c M b0) (c N b1) >-> {plus M N * cont append1}.
plus/b10:   plus (c M b1) (c N b0) >-> {plus M N * cont append1}.
plus/b11:   plus (c M b1) (c N b1) >-> {plus M N * cont carry}.
plus/carry: retn K * cont carry >-> {inc K * cont append0}.

cont/0:     retn R * cont append0 >-> {retn (c R b0)}.
cont/1:     retn R * cont append1 >-> {retn (c R b1)}.
```

Figure 6.22: Forward-chaining logic program for binary addition

6.6.1 Binary addition

In the notes for his Spring 2012 course on Linear Logic, Pfenning gave two algorithmic specifications of binary addition as logic programs; in both cases, binary numbers are represented as lists of bits, either $\ulcorner\epsilon\urcorner = \mathsf{eps}$, $\ulcorner n0\urcorner = \mathsf{c}\ulcorner n\urcorner\mathsf{b0}$, or $\ulcorner n1\urcorner = \mathsf{c}\ulcorner n\urcorner\mathsf{b1}$. The first specification was given as a forward-chaining ordered abstract machine [Pfe12d], and the second specification was given as a backward chaining logic program [Pfe12b]. Because these operational specifications were developed independently of operationalization, this provides an interesting and relatively simple test-case for operationalization, defunctionalization, and their implementation in the SLS prototype.

The backward-chaining logic program for binary addition is presented in Figure 6.21; the three-place relation plus depends on a two-place relation inc that handles carry bits. We operationalize this specification by giving plus and inc the evaluation predicates plus and inc, respectively, and giving them the same return predicate, retn.

In the direct operationalization of Figure 6.21, we can observe that there are three separate continuation frames (associated with the rules inc/b1, plus/b00, and plus/b11) that do the exact same thing: cause the bit 0 to be appended to the end of the returned number. With this observation, we can consolidate these three frames and the rules associated with them into one frame append0 and one rule cont/0 in Figure 6.22. Similarly, continuation frames associated with rules plus/b01 and plus/b10 both append the bit 1, and can be consolidated into the frame append1 and the rule cont/1 in Figure 6.22. (The only remaining frame is associated with the rule plus/b11 and invokes the increment procedure inc to handle the carry bit.) With the exception of the cre-

```
#mode value +.
value: exp -> prop.
value/lam: value (lam \x. E x).

#mode step + -.
step: exp -> exp -> prop.

step/app1:   step (app E1 E2) (app E1' E2)
              <- step E1 E1'.

step/app2:   step (app E1 E2) (app E1 E2') <- value E1
              <- step E2 E2'.

step/appred: step (app (lam \x. E x) V) (E V) <- value V.

#mode evsos + -.
evsos: exp -> exp -> prop.
evsos/steps: evsos E V <- step E E' <- evsos E' V.
evsos/value: evsos V V <- value V.
```

Figure 6.23: SOS evaluation

ation of redundant continuations, which could certainly be addressed by giving a more robust implementation of defunctionalization, Figure 6.22 can be seen as a direct operationalization of the deductive procedure in Figure 6.21.

Unfortunately, Figure 6.22 is not quite the same as Pfenning's ordered abstract machine for addition in [Pfe12d], but the difference is rather minor. In Pfenning's version of addition, the rule we call plus/carry in Figure 6.22 does not generate the conclusion cont append0. Instead, that frame is generated earlier by the rule we called plus/b11, which in Pfenning's formulation is $\forall M.\, \forall N.\, \mathsf{plus}\,(\mathsf{c}\, M\, \mathsf{b1})\,(\mathsf{c}\, N\, \mathsf{b1}) \rightarrowtail \{\mathsf{plus}\, M\, N \bullet \mathsf{cont}\, \mathsf{carry} \bullet \mathsf{cont}\, \mathsf{append0}\}$.

Relating specifications that differ only in the order with which certain continuation frames are generated seems to be a pervasive pattern. For example, Ian Zerny observed a very similar phenomenon when using operationalization to replay the correspondence between natural semantics and abstract machines presented in [DMMZ12]. Characterizing this observation more precisely is left for future work.

6.6.2 Operationalizing SOS specifications

We have thus far considered big-step operational semantics and abstract machines, mostly neglecting another great tradition in operational semantics, *structural operational semantics* (SOS) specifications [Plo04], though we did define the small-step judgment $\ulcorner e \mapsto e' \urcorner = \mathsf{step} \ulcorner e \urcorner \ulcorner e' \urcorner$ for call-by-value evaluation in the beginning of this chapter. The SOS specification from that discussion is encoded as an SLS specification in Figure 6.23. The figure also defines the judgment $\ulcorner e_1 \mapsto^* v \urcorner = \mathsf{evsos} \ulcorner e \urcorner \ulcorner v \urcorner$ that implements big-step evaluation in terms of the small-step SOS specification.

```
eval_sos: exp -> prop ord.
retn_sos: exp -> prop ord.
evsos/steps: eval_sos E * !step E E'  >-> {eval_sos E'}.
evsos/value: eval_sos V * !value V >-> {retn_sos V}.
```

Figure 6.24: The operationalization of evsos from Figure 6.23

There are several ways that we can contemplate operationalizing the SOS specification in Figure 6.23. If we operationalize only the evsos predicate, making the evaluation predicate eval_sos and the return predicate retn_sos, then we get what may be the most boring possible substructural operational semantics specification, shown in Figure 6.24. The specification is fully tail-recursive and there are no continuation frames, just an expression transitioning according to the rules of the small-step evaluation relation for an indefinite number of steps as we extend the trace. While the specification is almost trivial, it still captures something of the essence of an SSOS specification – atomic transitions are interpreted as steps (by way the inductively-defined relation $\mathsf{step}\ulcorner e\urcorner\ulcorner e'\urcorner$) and potentially nonterminating or failing computations are interpreted as traces. This specification is also the first case where we have performed operationalization on only part of a specification. In the terminology of Section 6.1.1, Figure 6.24 implies that the rules value/lam, step/app1, step/app2, and step/appred have been assigned to the category D of rules that remain in the deductive fragment while the rules evsos/steps and evsos/value were assigned to the category C of rules that end up being transformed.

In the other direction, we can consider operationalizing only the predicate step, which implies that the rules value/lam, and evsos/steps and evsos/value to the category D and placing step/app1, step/app2, and step/appred in the category C of rules that end up being transformed into the concurrent fragment. The result of this transformation is shown in Figure 6.25. The first subgoal of ev/steps, the proposition $!(\mathsf{decomp}\ E \rightarrowtail \{\mathsf{plug}\ E'\})$, is the first time we have actually encountered the effect of the $D^\dagger$ operation discussed in Section 6.1.2.

Instead of eval, we have decomp in Figure 6.25, since the relevant action is to decompose the expression looking for an applicable β-reduction, and instead of retn we have plug, since the relevant action is to plug the reduced expression back into the larger term. When we operationalized natural semantics, the structure of the suspended cont f propositions was analogous to the control stacks k of abstract machine specifications. In our operationalized SOS specification, the structure of the cont f propositions is analogous to *evaluation contexts*, often written as $E[]$.

$$E[] ::= E[]\ e \mid v\ E[] \mid []$$

The names *decomp* and *plug* are taken from the treatment of evaluation contexts in the functional correspondence [Dan08].

As we foreshadowed in Section 6.4.3, the right computational interpretation of Figure 6.25 is *not* committed-choice forward chaining; the concurrent rules we generate can get stuck without states being stuck, and factoring does not seem to provide a way out. Consider terms of type $\mathsf{eval}\ulcorner(\lambda x.x)\ e\urcorner \rightarrowtail \{\mathsf{retn}\ulcorner(\lambda x.x)\ e'\urcorner\}$ where $e \mapsto e'$ and consequently $\Theta\{x{:}\langle\mathsf{eval}\ulcorner e\urcorner\rangle\} \rightsquigarrow^* \Theta\{y{:}\langle\mathsf{retn}\ulcorner e'\urcorner\rangle\}$. It is entirely possible to use rule step/app1 to derive the following:

$$(x_1{:}\langle\mathsf{eval}\ulcorner(\lambda x.x)\ e\urcorner\rangle) \ \rightsquigarrow\ (x_2{:}\langle\mathsf{eval}\ulcorner\lambda x.x\urcorner\rangle,\ \ y_2{:}\langle\mathsf{cont}\,(\mathsf{ap1}\ulcorner e\urcorner)\rangle) \ \not\rightsquigarrow$$

```
decomp: exp -> prop ord.
plug: exp -> prop ord.

#| Reduction rules |#
step/appred: decomp (app (lam \x. E x) V) * !value V
              >-> {plug (E V)}.

#| Decomposing a term into an evaluation context |#
step/app1:   decomp (app E1 E2)
              >-> {decomp E1 * cont (ap1 E2)}.
step/app2:   decomp (app V1 E2) * !value V1
              >-> {decomp E2 * cont (ap2 V1)}.

#| Reconstituting a term from an evaluation context |#
step/app1/1: plug E1' * cont (ap1 E2)
              >-> {plug (app E1' E2)}.
step/app2/1: plug E2' * cont (ap2 E1)
              >-> {plug (app E1 E2')}.

#mode evsos + -.
evsos: exp -> exp -> prop.

evsos/steps: evsos E V
              <- (decomp E >-> {plug E'})
              <- evsos E' V.

evsos/value: evsos V V <- value V.
```

Figure 6.25: This transformation of Figure 6.23 evokes an evaluation context semantics

While stuck states in abstract machines raised alarm bells about language safety, the stuck state above is not a concern – we merely should have applied rule step/app2 to x_1 instead of rule step/app1. This corresponds to the fact that small-step SOS specifications and specifications that use evaluation contexts map most naturally to the backtracking search behavior generally associated with backward chaining.

6.6.3 Partial evaluation in $\lambda^{\bigcirc}$

As a final example, we present two SLS specifications of Davies' $\lambda^{\bigcirc}$, a logically-motivated type system and natural semantics for partial evaluation [Dav96]. Partial evaluation is not a modular language extension, either on paper or in SLS. On paper, we have to generalize the judgment $e \Downarrow v$ to have free variables; we write $e \Downarrow_{\Psi} v$ where Ψ contains free expression variables.

$$\frac{}{\lambda x.e \Downarrow_{\Psi} \lambda x.e} \qquad \frac{e_1 \Downarrow_{\Psi} \lambda x.e \quad e_2 \Downarrow_{\Psi} v_2 \quad [v_2/x]e \Downarrow_{\Psi} v}{e_1\, e_2 \Downarrow_{\Psi} v}$$

```
#mode fvar -.
fvar: exp -> prop.

#mode evn + + -.
evn: nat -> exp -> exp -> prop.

evn/var:  evn N X X <- fvar X.
evn/lam:  evn N (lam \x. E x) (lam \x. E' x)
           <- (All x. fvar x -> evn N (E x) (E' x)).
evn/app:  evn N (app E1 E2) (app E1' E2')
           <- evn N E1 E1'
           <- evn N E2 E2'.
```

Figure 6.26: Semantics of partial evaluation for $\lambda^{\bigcirc}$ (lambda calculus fragment)

In SLS, this does not actually require us to change the judgment $\mathsf{ev}\, e\, v$ from Figure 6.1, since the specification itself does not specify the context of LF. However, $\lambda^{\bigcirc}$ also requires a separate judgment $e \Downarrow^n_\Psi e'$ for partially evaluating expressions that will be fully evaluated not now but n partial evaluation stages in the future. On the fragment of the logic that deals with functions and applications, this judgment does nothing but induct over the structure of expressions:

$$\frac{x \in \Psi}{x \Downarrow^n_\Psi x} \qquad \frac{e \Downarrow^n_{\Psi,x} e'}{\lambda x.e \Downarrow^n_\Psi \lambda x.e'} \qquad \frac{e_1 \Downarrow^n_\Psi e_1' \quad e_2 \Downarrow^n_\Psi e_2'}{e_1\, e_2 \Downarrow^n_\Psi e_1'\, e_2'}$$

Note that the partial evaluation rule for $\lambda x.e$ extends the variable context Ψ. The SLS encoding of the judgment $e \Downarrow^n_\Psi e'$ is given in Figure 6.26, which also introduces an auxiliary fvar judgment that tracks all the variables in Ψ.

The evaluation judgment $e \Downarrow_\Psi v$ and the partial evaluation judgment $e \Downarrow^n_\Psi e'$ only interact in $\lambda^{\bigcirc}$ through the temporal fragment, which mediates between the two judgments by way of two expressions. The first, $\mathsf{next}\, e$, says that the enclosed expression e should be evaluated one time step later than the surrounding expression. The second, $\mathsf{prev}\, e$, says that the enclosed expression should be evaluated one time step *before* the surrounding expression. When we evaluate $\mathsf{prev}\, e$ at time 1 it is necessary that e evaluates to $\mathsf{next}\, e'$, as $\mathsf{prev}\,(\mathsf{next}\, e')$ at time step 1 will reduce to e'.

$$\frac{e \Downarrow^1_\Psi e'}{\mathsf{next}\, e \Downarrow_\Psi \mathsf{next}\, e'} \qquad \frac{e \Downarrow^{n+1}_\Psi e'}{\mathsf{next}\, e \Downarrow^n_\Psi \mathsf{next}\, e'} \qquad \frac{e \Downarrow_\Psi \mathsf{next}\, e'}{\mathsf{prev}\, e \Downarrow^1_\Psi e'} \qquad \frac{e \Downarrow^{n+1}_\Psi e'}{\mathsf{prev}\, e \Downarrow^{n+2}_\Psi \mathsf{prev}\, e'}$$

This natural semantics specification is represented on the left-hand side of Figure 6.27. Due to the structure of the evn/lam rule, we *cannot* operationalize the evn predicate: it does not have the structure of a C proposition as described in Section 6.1.1. Rule evn/lam does, however, have the structure of a D proposition if we assign evn to the class of predicates that are not operationalized. Therefore, it is possible to operationalize the ev predicate without operationalizing the evn predicate. This leaves the rules in Figure 6.26 completely unchanged; the right-hand side of Figure 6.27 contains the transformed temporal fragment, where evn/next and evn/prev rules are similarly unchanged. The ev/next rule, however, contains a subgoal !evn $(\mathsf{s}\,\mathsf{z})\, E\, V$ which uses a deductive derivation to build a concurrent step. Conversely, the ev/prev rule contains a subgoal

```
ev/next:
ev (next E) (next E')
 <- evn (s z) E E'.

evn/next:
evn N (next E) (next E')
 <- evn (s N) E E'.

ev/prev:
evn (s z) (prev E) E'
 <- ev E (next E').

evn/prev:
evn (s (s N)) (prev E) (prev E')
 <- evn (s N) E E'.
```

```
ev/next:
eval (next E) * !evn (s z) E E'
 >-> {retn (next E')}.

evn/next:
evn N (next E) (next E')
 <- evn (s N) E E'.

ev/prev:
evn (s z) (prev E) E'
 <- (eval E >-> {retn (next E')}).

evn/prev:
evn (s (s N)) (prev E) (prev E')
 <- evn (s N) E E'.
```

Figure 6.27: Semantics for $\lambda^{\bigcirc}$ (temporal fragment)

of $\mathsf{eval}\, E \rightarrowtail \{\mathsf{retn}\,(\mathsf{next}\, V)\}$ that uses a concurrent derivation to create a deductive derivation. This makes the right-hand side of Figure 6.27 the only SLS specification in this dissertation that exhibits an arbitrarily nested dependency between concurrent and deductive reasoning.

The natural semantics of $\lambda^{\bigcirc}$ are not, on a superficial level, significantly more complex than other natural semantics. It turns out, though, that the usual set of techniques for adding state to a operational semantics break down for $\lambda^{\bigcirc}$. Discussing a $\lambda^{\bigcirc}$-like logic with state remained a challenge for many years, though a full solution has recently been given by Kameyama et al. using delimited control operators [KKS11]. Our discussion of operationalization gives a perspective on why this task is difficult, as the specification is far outside of the image of the extended natural semantics we considered in Section 6.5.5. We normally add state to ordered abstract machine specifications by manipulating and extending the set of ambient linear and persistent resources. If we tried to add state to $\lambda^{\bigcirc}$ the same way we added it in Section 6.5.1, the entire store would effectively leave scope whenever computation considered the subterm e of $\mathsf{next}\, e$.

I conjecture that the nominal generalization of ordered linear lax logic alluded to in the discussion of locations and existential name generation (Section 6.5.1) could support operationalizing predicates like $\mathsf{evn}\, n\, e\, e'$. This might, in turn, make it possible to add state to an SSOS specification of $\lambda^{\bigcirc}$, but that is left for future work.

Chapter 7

Destination-passing

The natural notion of ordering provided by ordered linear logic is quite convenient for encoding evolving systems that perform local manipulations to a stack-like structure. This was demonstrated by the push-down automaton for generic bracket matching discussed in the introduction. We can now present that specification in Figure 7.1 as an SLS specification.

```
hd: prop ord.
left: tok -> prop ord.
right: tok -> prop ord.
stack: tok -> prop ord.

push: hd * left X >-> {stack X * hd}.
pop:  stack X * hd * right X >-> {hd}.
```

Figure 7.1: Ordered SLS specification of a PDA for parenthesis matching

Tree structures were reasonably straightforward to encode in the ordered context as well, as we saw from the SSOS specification for parallel pairs in Chapter 6.

At some point, however, the simple data structures that can be naturally encoded in an ordered context become too limiting. When we reach this point, we turn to *destinations*, which allow us to glue control flow together in much more flexible ways. Destinations (terms of type dest) are a bit like the locations l introduced in the specification of mutable storage in Section 6.5.1. They have no constructors: they are only introduced as variables by existential quantification, which means they can freely be subject to unification when the conclusion of a rule declares them to be equal (as described in Section 4.2). Destinations allow us to encode very expressive structures in the linear context of SLS. Instead of using order to capture the local relationships between different propositions, we use destinations.

Linear logic alone is able to express any (flat, concurrent) specifications that can be expressed using ordered atomic propositions. In other words, we did not ever *need* order, it was just a more pleasant way to capture simple control structures. We will demonstrate that fact in this chapter by describing a transformation, *destination-adding*, from specifications with ordered atomic propositions to specifications that only include linear and persistent atomic propositions. This destination-adding transformation, which we originally presented in [SP11a], turns all ordered

atomic propositions into linear atomic propositions and tags them with two new arguments (the destinations of the destination-adding transformation). These extra destinations serve as a link between a formerly-ordered atomic proposition and its two former neighbors in the ordered context. When we perform the destination-adding transformation on the specification in Figure 7.1, we get the specification in Figure 7.2.

```
hd: dest -> dest -> prop lin.
left: tok -> dest -> dest -> prop lin.
right: tok -> dest -> dest -> prop lin.
stack: tok -> dest -> dest -> prop lin.

push: hd L M * left X M R >-> {Exists m. stack X L m * hd m R}.
pop:  stack X L M1 * hd M1 M2 * right X M2 R >-> {hd L R}.
```

Figure 7.2: Linear SLS specification of a PDA for parenthesis matching

The specification in Figure 7.2, like every other specification that results from destination-adding, has no occurrences of $\downarrow A^-$ (the transformation has not been adapted to nested rules) and no ordered atomic propositions (these are specifically removed by the transformation). As a result, we write `hd L M` instead of `$hd L M`, omitting the optional linearity indicator `$` on the linear atomic propositions as discussed in Section 4.5. Additionally, by the discussion in Section 3.7, we would be justified in viewing this specification as a linear logical specification (or a CLF specification) instead of a ordered logical specification in SLS. This would not impact the structure of the derivations significantly; essentially, it just means that we would write $A_1^+ \multimap \{A_2^+\}$ instead of $A_1^+ \rightarrowtail \{A_2^+\}$. This reinterpretation was used in [SP11a], but we will stick with the notation of ordered logic for consistency, while recognizing that there is nothing ordered about specifications like the one in Figure 7.2.

When the destination-adding translation is applied to ordered abstract machine SSOS specifications, the result is a style of SSOS specification called *destination-passing*. Destination-passing specifications were the original style of SSOS specification proposed in the CLF technical reports [CPWW02]. Whereas the operationalization transformation exposed the structure of natural semantics proofs so that they could be modularly extended with stateful features, the destination-adding translation exposes the control structure of specifications, allowing the language to be modularly extended with control effects and effects like synchronization.

7.1 Logical transformation: destination-adding

The translation we define operates on rules the form $\forall\overline{x}.S_1 \rightarrowtail \{S_2\}$, where S_1 must contain at least one ordered atomic proposition. The syntactic category S is a refinement of the positive types A^+ defined by the following grammar:

$$S ::= p^+_{pers} \mid p^+_{eph} \mid p^+ \mid \mathbf{1} \mid t \doteq s \mid S_1 \bullet S_2 \mid \exists x{:}\tau.S$$

The translation of a rule $\forall\overline{x}.S_1 \rightarrowtail \{S_2\}$ is then $\forall\overline{x}.\,\forall d_L{:}\mathsf{dest}.\,\forall d_R{:}\mathsf{dest}.\,[\![S_1]\!]^{d_L}_{d_R} \rightarrowtail \{[\![S_2]\!]^{d_L}_{d_R}\}$, where $[\![S]\!]^{d_L}_{d_R}$ is defined in Figure 7.3. It is also necessary to transform all ordered predicates with

$$
\begin{aligned}
[\![p^+]\!]^{d_L}_{d_R} &= \mathsf{a}\, t_1 \ldots t_n\, d_L\, d_R \quad (\text{where } p^+ = \mathsf{a}\, t_1 \ldots t_n) \\
[\![p^+_{eph}]\!]^{d_L}_{d_R} &= p^+_{eph} \bullet d_L \doteq d_R \\
[\![p^+_{pers}]\!]^{d_L}_{d_R} &= p^+_{pers} \bullet d_L \doteq d_R \\
[\![\mathbf{1}]\!]^{d_L}_{d_R} &= d_L \doteq d_R \\
[\![t \doteq s]\!]^{d_L}_{d_R} &= t \doteq s \bullet d_L \doteq d_R \\
[\![S_1 \bullet S_2]\!]^{d_L}_{d_R} &= \exists d_M{:}\mathsf{dest}.\ [\![S_1]\!]^{d_L}_{d_M} \bullet [\![S_2]\!]^{d_M}_{d_R} \\
[\![\exists x{:}\tau.\ S]\!]^{d_L}_{d_R} &:= \exists x{:}\tau.\ [\![S]\!]^{d_L}_{d_R}
\end{aligned}
$$

Figure 7.3: Destination-adding transformation

kind $\Pi.x_1{:}\tau_1 \ldots \Pi.x_n{:}\tau_n.\ \mathsf{prop\,ord}$ that are declared in the signature into predicates with kind $\Pi.x_1{:}\tau_1 \ldots \Pi.x_n{:}\tau_n.\ \mathsf{dest} \to \mathsf{dest} \to \mathsf{prop\,ord}$ in order for the translation of an ordered atomic proposition p^+ to remain well-formed in the transformed signature.

The destination-adding translation presented here is the same as the one presented in [SP11a], except that the transformation operated on rules of the form $\forall\overline{x}.S_1 \twoheadrightarrow \{S_2\}$ and ours will operate over rules of the form $\forall\overline{x}.S_1 \rightarrowtail \{S_2\}$.[1] As discussed in Section 6.2.2, the difference between $\twoheadrightarrow$ and $\rightarrowtail$ is irrelevant in this situation. The restriction to flat specifications, on the other hand, is an actual limitation. We conjecture that the translation presented here, and the correctness proof presented in [SP11a], would extend to nested SLS specifications. However, the detailed correctness proofs in that work are already quite tedious (though our explicit notation for patterns as partial derivations can simplify the proof somewhat) and the limited transformation described by Figure 7.3 is sufficient for our purposes. Therefore, we will rely on the existing result, leaving the correctness of a more general development for future work.

According to Figure 7.3, the rule pop in Figure 7.2 should actually be written as follows:

$$
\begin{aligned}
&\mathsf{pop} : \forall x{:}\mathsf{tok}.\ \forall l{:}\mathsf{dest}.\ \forall r{:}\mathsf{dest}. \\
&\qquad (\exists m_1{:}\mathsf{dest}.\ \mathsf{stack}\, x\, l\, m_1 \bullet (\exists m_2{:}\mathsf{dest}.\ \mathsf{hd}\, m_1\, m_2 \bullet \mathsf{right}\, x\, m_2\, r)) \\
&\qquad\quad \rightarrowtail \{\mathsf{hd}\, l\, r\}
\end{aligned}
$$

The destination-adding transformation as implemented produces rules that are equivalent to the specification in Figure 7.3 but that avoid unnecessary equalities and push existential quantifiers as far out as possible (which includes turning existential quantifiers $(\exists x.\ A^+) \rightarrowtail B^-$ into universal quantifiers $\forall x.\ A^+ \rightarrowtail B^-$). The result is a specification, equivalent at the level of synthetic transitions, that looks like the one in Figure 7.2. We write the result of the destination-adding transformation on the signature Σ as $Dest(\Sigma)$.

We can consider a further simplification: is it necessary to generate a new destination m by existential quantification in the head $\exists m.\ \mathsf{stack}\, x\, l\, m \bullet \mathsf{hd}\, m\, r$ of push in Figure 7.2? There is

[1] The monad $\{S_2\}$ did not actually appear in [SP11a], and the presentation took polarity into account but was not explicitly polarized. We are justified in reading the lax modality back in by the erasure arguments discussed in Section 3.7.

already a destination m mentioned in the head that will be unused in the conclusion. It would, in fact, be possible to avoid generating new destinations in the transformation of rules $\forall\overline{x}.\, S_1 \rightarrowtail \{S_2\}$ where the head S_2 contains no more ordered atomic propositions than the premise S_1.

We don't perform this simplification for a number of reasons. First and foremost, the transformation described in Figure 7.3 more closely follows the previous work by Morrill, Moot, Piazza, and van Benthem discussed in Section 5.2, and using the transformation as given simplifies the correctness proof (Theorem 7.1). Pragmatically, the additional existential quantifiers also give us more structure to work with when considering program abstraction in Chapter 8. Finally, if we apply both the transformation in Figure 7.3 and a transformation that reuses destinations to an ordered abstract machine SSOS specification, the former transformation produces results that are more in line with existing destination-passing SSOS specifications.

To prove the correctness of destination-adding, we must describe a translation $[\![\Psi; \Delta]\!]$ from process states with ordered, linear, and persistent atomic propositions to ones with only linear and persistent atomic propositions:

$$
\begin{aligned}
[\![\Psi; \cdot]\!] &= (\Psi, d_L{:}\mathsf{dest}; \cdot) \\
[\![\Psi; \Delta, x{:}\langle \mathsf{a}\, t_1 \ldots t_n\rangle\ ord]\!] &= (\Psi', d_L{:}\mathsf{dest}, d_R{:}\mathsf{dest}; \Delta', x{:}\langle \mathsf{a}\, t_1 \ldots t_n\, d_L\, d_R\rangle) \\
&\qquad \text{(where } \mathsf{a} \text{ is ordered and } [\![\Psi; \Delta]\!] = (\Psi', d_L{:}\mathsf{dest}; \Delta')) \\
[\![\Psi; \Delta, x{:}S\ ord]\!] &= (\Psi', d_L{:}\mathsf{dest}, d_R{:}\mathsf{dest}; \Delta', x{:}[\![S]\!]^{d_L}_{d_R}\ ord) \\
&\qquad \text{(where } \mathsf{a} \text{ is ordered and } [\![\Psi; \Delta]\!] = (\Psi', d_L{:}\mathsf{dest}; \Delta')) \\
[\![\Psi; \Delta, x{:}\langle p^+_{eph}\rangle\ eph]\!] &= (\Psi'; \Delta', x{:}\langle p^+_{eph}\rangle) \\
&\qquad \text{(where } [\![\Psi; \Delta]\!] = (\Psi'; \Delta')) \\
[\![\Psi; \Delta, x{:}\langle p^+_{pers}\rangle\ pers]\!] &= (\Psi'; \Delta', x{:}\langle p^+_{pers}\rangle) \\
&\qquad \text{(where } [\![\Psi; \Delta]\!] = (\Psi'; \Delta'))
\end{aligned}
$$

Theorem 7.1 (Correctness of destination-adding)**.**
$[\![\Psi; \Delta]\!] \rightsquigarrow_{Dest(\Sigma)} (\Psi_l; \Delta_l)$ *if and only if* $(\Psi; \Delta) \rightsquigarrow_{\Sigma} (\Psi_o; \Delta_o)$ *and* $(\Psi_l; \Delta_l) = [\![\Psi_o, \Psi''; \Delta_o]\!]$ *for some variable context* Ψ'' *containing destinations free in the translation of* Δ *but not in the translation of* Δ_o.

Proof. This proof is given in detail in [SP11a, Appendix A]. It involves a great deal of tedious tracking of destinations, but the intuition behind that tedious development is reasonably straightforward.

First, we need to prove that a right-focused proof of $\Psi; \Delta \vdash_{\Sigma} [S]$ implies that there is an analogous proof of $[\![\Psi; \Delta]\!] \vdash_{Dest(\Sigma)} [[\![S]\!]^{d_L}_{d_R}]$. Conversely, if we can prove $\Psi; \Delta \vdash_{Dest(\Sigma)} [[\![S]\!]^{d_L}_{d_R}]$ in right focus under then linear translation, then it is possible to reconstruct an ordered context $\Psi'; \Delta'$ such that $[\![\Psi'; \Delta']\!] = \Psi; \Delta$ and $\Psi'; \Delta' \vdash_{\Sigma} [S]$ by threading together the destinations from d_L to d_R in Δ. Both directions are established by structural induction on the given derivation. The critical property is that it is possible to reconstruct the ordered context from the context of any right-focus sequent that arises during translation. Proving that property is where the flat structure of rules is particularly helpful; the use of positive atomic propositions comes in handy too [SP11a, Lemma 1].

```
eval: exp -> dest -> dest -> prop lin.
retn: exp -> dest -> dest -> prop lin.
cont: frame -> dest -> dest -> prop lin.

ev/lam:  eval (lam \x. E x) D' D >-> {retn (lam \x. E x) D' D}.

ev/app:  eval (app E1 E2) D' D
          >-> {Exists d1. eval E1 D' d1 * cont (app1 E2) d1 D}.

ev/app1: retn (lam \x. E x) D' D1 * cont (app1 E2) D1 D
          >-> {Exists d2. eval E2 D' d2 * cont (app2 \x. E x) d2 D}.

ev/app2: retn V2 D' D2 * cont (app2 \x. E x) D2 D
          >-> {eval (E V2) D' D}.
```

Figure 7.4: Translation of Figure 6.6 with vestigial destinations

Second, we need to prove that patterns can be translated in both directions: that if $(\Psi; \Delta) \Longrightarrow (\Psi'; \Delta_o)$ under the original signature then $P :: [\![\Psi; \Delta]\!] \Longrightarrow [\![\Psi'; \Delta_o]\!]$ under the translated signature [SP11a, Lemma 4], and that if $P :: [\![\Psi; \Delta]\!] \Longrightarrow (\Psi'; \Delta_l)$ then there exists Δ_o such that $(\Psi'; \Delta_o) = [\![\Psi'; \Delta'']\!]$ [SP11a, Lemma 5]. Both directions are again by induction over the structure of the given pattern.

The theorem then follows directly from these two lemmas. There is a trivial induction on spines to handle the sequence of quantifiers, but the core of a flat rule is a proposition $S_1 \rightarrowtail \{S_2\}$ – we reconstruct the ordered context from the value used to prove S_1, and then begin inverting with the positive proposition S_2 in the context. □

If we leave off explicitly mentioning the variable context Ψ, then the trace that represents successfully processing the string [( )] with the transformed push-down automaton specification in Figure 7.2 is as follows (we again underline hd for emphasis):

$$
\begin{aligned}
& y_0{:}\langle \underline{\mathsf{hd}}\ d_0\ d_1\rangle, x_1{:}\langle \mathsf{left\ sq}\ d_1\ d_2\rangle, x_2{:}\langle \mathsf{left\ pa}\ d_2\ d_3\rangle, x_3{:}\langle \mathsf{right\ pa}\ d_3\ d_4\rangle, x_4{:}\langle \mathsf{right\ sq}\ d_4\ d_5\rangle \\
\rightsquigarrow\ & z_1{:}\langle \mathsf{stack\ sq}\ d_0\ d_6\rangle, y_1{:}\langle \underline{\mathsf{hd}}\ d_6\ d_2\rangle, x_2{:}\langle \mathsf{left\ pa}\ d_2\ d_3\rangle, x_3{:}\langle \mathsf{right\ pa}\ d_3\ d_4\rangle, x_4{:}\langle \mathsf{right\ sq}\ d_4\ d_5\rangle \\
\rightsquigarrow\ & z_1{:}\langle \mathsf{stack\ sq}\ d_0\ d_6\rangle, z_2{:}\langle \mathsf{stack\ pa}\ d_6\ d_7\rangle, y_2{:}\langle \underline{\mathsf{hd}}\ d_7\ d_3\rangle, x_3{:}\langle \mathsf{right\ pa}\ d_3\ d_4\rangle, x_4{:}\langle \mathsf{right\ sq}\ d_4\ d_5\rangle \\
\rightsquigarrow\ & z_1{:}\langle \mathsf{stack\ sq}\ d_0\ d_6\rangle, y_3{:}\langle \underline{\mathsf{hd}}\ d_6\ d_4\rangle, x_4{:}\langle \mathsf{right\ sq}\ d_4\ d_5\rangle \\
\rightsquigarrow\ & y_4{:}\langle \underline{\mathsf{hd}}\ d_0\ d_5\rangle
\end{aligned}
$$

One reason for leaving off the variable context Ψ in this example is that by the end it contains the LF variables d_1, d_2, d_3, d_4, d_5, d_6, and d_7, none of which are actually present in the substructural context $y_4{:}\langle \underline{\mathsf{hd}}\ d_0\ d_5\rangle$. We can informally think of these destinations as having been "garbage collected," but this notion is not supported by the formal system we described in Chapter 4.

```
eval: exp -> dest -> prop lin.
retn: exp -> dest -> prop lin.
cont: frame -> dest -> dest -> prop lin.

ev/lam:  eval (lam \x. E x) D >-> {retn (lam \x. E x) D}.

ev/app:  eval (app E1 E2) D
          >-> {Exists d1. eval E1 d1 * cont (app1 E2) d1 D}.

ev/app1: retn (lam \x. E x) D1 * cont (app1 E2) D1 D
          >-> {Exists d2. eval E2 d2 * cont (app2 \x. E x) d2 D}.

ev/app2: retn V2 D2 * cont (app2 \x. E x) D2 D
          >-> {eval (E V2) D}.
```

Figure 7.5: Translation of Figure 6.6 without vestigial destinations

7.1.1 Vestigial destinations

When we apply the translation of expressions to the call-by-value lambda calculus specification from Figure 6.6, we get the specification in Figure 7.4. Because eval and retn are always unique and always appear at the leftmost end of this substructural context, this specification has a quirk: the second argument to eval and retn is always d', and the destination never changes; it is essentially a vestige of the destination-adding transformation. As long as we are transforming a sequential ordered abstract machine, we can eliminate this vestigial destination, giving us the specification in Figure 7.5. This extra destination is *not* vestigial when we translate a parallel specification, but as we discuss in Section 7.2.1, we don't necessarily want to apply destination-adding to parallel ordered abstract machines anyway.

7.1.2 Persistent destination passing

When we translate our PDA specification, it is actually not necessary to translate hd, left, right and stack as linear atomic propositions. If we translate hd as a linear predicate but translate the other predicates as persistent predicates, it will still be the case that there is always exactly one linear atomic proposition $\mathsf{hd}\, d_L\, d_R$ in the context, at most one $\mathsf{stack}\, x\, d\, d_L$ proposition with the same destination d_L, and at most one $\mathsf{right}\, x\, d_R\, d$ or $\mathsf{left}\, x\, d_R\, d$ with the same destination d_R. This means it is still the case that the PDA accepts the string if and only if there is the following series of transitions:

$$(x{:}\langle \mathsf{hd}\, d_0\, d_1\rangle, y_1{:}\langle \mathsf{left}\, x_1\, d_1\, d_2\rangle, \ldots, y_n{:}\langle \mathsf{right}\, x_n\, d_n\, d_{n+1}\rangle) \;\leadsto^*\; (\Gamma, z{:}\langle \mathsf{hd}\, d_0\, d_{n+1}\rangle)$$

Unlike the entirely-linear PDA specification, the final state may include some additional persistent propositions, represented by Γ. Specifically, the final state contains all the original $\mathsf{left}\, x\, d_i\, d_{i+1}$ and $\mathsf{right}\, x\, d_i\, d_{i+1}$ propositions along with all the $\mathsf{stack}\, x\, d\, d'$ propositions that were created during the course of evaluation.

I originally conjectured that a version of Theorem 7.1 would hold in any specification that turned some ordered atomic propositions linear and others persistent just as long as at least one atomic proposition in the premise of every rule remained linear after transformation. This would have given a generic justification for turning left, right and stack persistent in Figure 7.2 and to turning cont persistent in Figure 7.5. However, that condition is not strong enough. To see why, consider a signature with one rule, $\mathsf{a} \bullet \mathsf{b} \bullet \mathsf{a} \rightarrowtail \{\mathsf{b}\}$, where a and b are ordered atomic propositions. We can construct the following trace:

$$(x_1{:}\langle\mathsf{a}\rangle, x_2{:}\langle\mathsf{b}\rangle, x_3{:}\langle\mathsf{a}\rangle, x_4{:}\langle\mathsf{b}\rangle, x_5{:}\langle\mathsf{a}\rangle) \rightsquigarrow (x{:}\langle\mathsf{b}\rangle, x_4{:}\langle\mathsf{b}\rangle, x_5{:}\langle\mathsf{a}\rangle) \not\rightsquigarrow$$

From the same starting point, exactly one other trace is possible:

$$(x_1{:}\langle\mathsf{a}\rangle, x_2{:}\langle\mathsf{b}\rangle, x_3{:}\langle\mathsf{a}\rangle, x_4{:}\langle\mathsf{b}\rangle, x_5{:}\langle\mathsf{a}\rangle) \rightsquigarrow (x_1{:}\langle\mathsf{a}\rangle, x_2{:}\langle\mathsf{b}\rangle, x{:}\langle\mathsf{b}\rangle) \not\rightsquigarrow$$

However, if we perform the destination-passing transformation, letting $\mathsf{a}\, d\, d'$ be a persistent atomic proposition and letting $\mathsf{b}\, d\, d'$ be a linear atomic proposition, then we have a series of transitions in the transformed specification that can reuse the atomic proposition $\mathsf{a}\, d_2\, d_3$ in a way that doesn't correspond to any series of transitions in ordered logic:

$$\begin{aligned}
& x_1{:}\langle\mathsf{a}\, d_0\, d_1\rangle\ \mathit{pers}, x_2{:}\langle\mathsf{b}\, d_1\, d_2\rangle\ \mathit{eph}, x_3{:}\langle\mathsf{a}\, d_2\, d_3\rangle\ \mathit{pers}, x_4{:}\langle\mathsf{b}\, d_3\, d_4\rangle\ \mathit{eph}, x_5{:}\langle\mathsf{a}\, d_4\, d_5\rangle\ \mathit{pers} \\
\rightsquigarrow\ & x_1{:}\langle\mathsf{a}\, d_0\, d_1\rangle\ \mathit{pers}, \underline{x{:}\langle\mathsf{b}\, d_0\, d_3\rangle\ \mathit{eph}}, x_3{:}\langle\mathsf{a}\, d_2\, d_3\rangle\ \mathit{pers}, x_4{:}\langle\mathsf{b}\, d_3\, d_4\rangle\ \mathit{eph}, x_5{:}\langle\mathsf{a}\, d_4\, d_5\rangle\ \mathit{pers} \\
\rightsquigarrow\ & x_1{:}\langle\mathsf{a}\, d_0\, d_1\rangle\ \mathit{pers}, x{:}\langle\mathsf{b}\, d_0\, d_3\rangle\ \mathit{eph}, x_3{:}\langle\mathsf{a}\, d_2\, d_3\rangle\ \mathit{pers}, \underline{x'{:}\langle\mathsf{b}\, d_2\, d_5\rangle\ \mathit{eph}}, x_5{:}\langle\mathsf{a}\, d_4\, d_5\rangle\ \mathit{pers}
\end{aligned}$$

In the first process state, there is a path $d_0, d_1, d_2, d_3, d_4, d_5$ through the context that reconstructs the ordering in the original ordered context. In the second process state, there is still a path d_0, d_3, d_4, d_5 that allows us to reconstruct the ordered context $(x{:}\langle\mathsf{b}\rangle, x_4{:}\langle\mathsf{b}\rangle, x_5{:}\langle\mathsf{a}\rangle)$ by ignoring the persistent propositions associated with x_1 and x_3. However, in the third process state above, no path exists, so the final state cannot be reconstructed as any ordered context.

It would be good to identify a condition that allowed us to selectively turn some ordered propositions persistent when destination-adding without violating (a version of) Theorem 7.1. In the absence of such a generic condition, it is still straightforward to see that performing destination-passing and then turning some propositions persistent is an *abstraction*: if the original system can make a series of transitions, the transformed system can simulate those transitions, but the reverse may not be true. In any case, we can observe that, for many of systems we are interested in, a partially-persistent destination-passing specification can only make transitions that were possible in the ordered specification. The push-down automata with persistent stack, left, and right is one example of this, and we can similarly make the cont predicate persistent in SSOS specifications without introducing any new transitions. Turing the cont predicate persistent will be necessary for the discussion of first-class continuations in Section 7.2.4.

7.2 Exploring the richer fragment

In [SP11a], we were interested in exact logical correspondence between ordered abstract machine SSOS specifications and destination-passing SSOS specifications. (Destination-adding is

```
cont2: frame -> dest -> dest -> dest -> prop lin.

ev/pair:  eval (pair E1 E2) D
           >-> {Exists d1. Exists d2.
                eval E1 d1 * eval E2 d2 * cont2 pair1 d1 d2 D}.

ev/pair1: retn V1 D1 * retn V2 D2 * cont2 pair1 D1 D2 D
           >-> {retn (pair V1 V2) D}.
```

Figure 7.6: Destination-passing semantics for parallel evaluation of pairs

useful in that context because it exposes information about the control structure of computations; this control structure can be harnessed by the program abstraction methodology described in Chapter 8 to derive program analyses.) In keeping with our broader use of the logical correspondence, this section will cover programming language features that are not easily expressible with ordered abstract machine SSOS specifications but that can be easily expressed with destination-passing SSOS specifications. Consequently, these are features that can be modularly added to (sequential) ordered abstract machine specifications that have undergone the destination-adding transformation.

The semantics of parallelism and failure presented in Section 7.2.1 are new. The semantics of futures (Section 7.2.3) and synchronization (Section 7.2.2) are based on the specifications first presented in the CLF tech report [CPWW02]. The semantics of first-class continuations (Section 7.2.4) were presented previously in [Pfe04, PS09]. In destination-passing semantics, when we are dealing with fine-grained issues of control flow, the interaction of programming language features becomes more delicate. Parallel evaluation, recoverable failure, and synchronization are compatible features, as are synchronization and futures. Failure and first-class continuations are also compatible. We will not handle other interactions, though it would be interesting to explore the adaptation of Moreau and Ribbens' abstract machine for Scheme with parallel evaluation and callcc as a substructural operational semantics [MR96].

7.2.1 Alternative semantics for parallelism and failure

In Section 6.5.4, we discussed how parallel evaluation and recoverable failure can be combined in an ordered abstract machine SSOS specification. Due to the fact that the two parts of a parallel ordered abstract machine are separated by an arbitrary amount of ordered context, some potentially desirable ways of integrating parallelism and failure were difficult or impossible to express, however.

Once we transition to destination-passing SSOS specifications, it is possible to give a more direct semantics to parallel evaluation that better facilitates talking about failure. Instead of having the stack frame associated with parallel pairs be cont pair1 (as in Figure 6.8) or cont2 pair1 (as discussed in Section 6.5.4), we create a continuation cont2 pair1 d_1 d_2 d with *three* destinations; d_1 and d_2 represent the return destinations for the two subcomputations, whereas d represents the destination to which the evaluated pair is to be returned. This strategy applied to the parallel evaluation of pairs is shown in Figure 7.6.

```
error: dest -> prop lin.
handle: exp -> dest -> dest -> prop lin.
terminate: dest -> prop lin.

ev/fail:   eval fail D >-> {error D}.
ev/error:  error D' * cont F D' D >-> {error D}.
ev/errorL: error D1 * cont2 F D1 D2 D >-> {error D * terminate D2}.
ev/errorR: error D2 * cont2 F D1 D2 D >-> {error D * terminate D1}.

term/retn: retn V D * terminate D >-> {one}. ; Returning in vain
term/err:  error D * terminate D >-> {one}.  ; Failing redundantly

ev/catch:  eval (catch E1 E2) D
            >-> {Exists d'. eval E1 d' * handle E2 d' D}.
ev/catcha: retn V D' * handle _ D' D >-> {retn V D}.
ev/catchb: error D' * handle E2 D' D >-> {eval E2 D}.
```

Figure 7.7: Integration of parallelism and exceptions; signals failure as soon as possible

In ordered specifications, an ordered atomic proposition can be directly connected to at most two other ordered propositions: the proposition immediately to the left in the ordered context, and the proposition immediately to the right in the ordered context. What Figure 7.6 demonstrates is that, with destinations, a linear proposition can be locally connected to *any fixed finite number* of other propositions. (If we encode lists of destinations, this need not even be fixed!) Whereas in ordered abstract machine specifications the parallel structure of a computation had to be reconstructed by parsing the context in postfix, a destination-passing specification uses destinations to thread together the treelike dependencies in the context. It would presumably be possible to consider a different version of parallel operationalization that targeted this desirable form of parallel destination-passing specification specifically, but we will not present such a transformation in this thesis.

Using destination-based parallel continuations, we give, in Figure 7.7, a semantics for recoverable failure that eagerly returns errors from either branch of a parallel computation. The rules ev/errorL and ev/errorR immediately pass on errors returned to a frame where the computation forked. Those two rules also leave behind a linear proposition $\mathsf{terminate}\ d$ that will abort the other branch of computation if it returns successfully (rule term/retn) or with an error (rule term/err). It would also be possible to add rules like $\forall d.\, \forall d'.\, \mathsf{cont}\ d'\ d \bullet \mathsf{terminate}\ d \rightarrowtail \{\mathsf{terminate}\ d'\}$ that actively abort the useless branch instead of passively waiting for it to finish. (In a language with state, this can make an observable difference in the results of computation.)

7.2.2 Synchronization

The CLF tech report gives a destination-passing presentation of nearly the full set of Concurrent ML primitives, omitting only negative acknowledgements [CPWW02]. We will present an SLS version of that Concurrent ML specification as a part of the hybrid specification in Appendix B. In Figure 7.8, rather than reprising that specification, we present an extremely simple

```
ev/chan:  eval (chan \c. E c) D >-> {Exists c. eval (E c) D}.

ev/send:  eval (send C E) Dsend
           >-> {Exists d'. eval E d' * cont (send1 C) d' Dsend}.

ev/send1: retn V D' * cont (send1 C) D' Dsend * eval (recv C) Drecv
           >-> {retn unit Dsend * retn V Drecv}.
```

Figure 7.8: Semantics of simple synchronization

form of synchronous communication.

New channels are created by evaluating $\ulcorner \mathsf{chan}\, c.e \urcorner = \mathsf{chan}\, \lambda c. \ulcorner e \urcorner$, which introduces a new channel (an LF term of the type channel that has no constructors) and substitutes it for the bound variable c in e. Synchronization happens when there is both a send $\mathsf{send}\, c\, e$ being evaluated in one part of the process state and a receive $\mathsf{recv}\, c$ with the same channel being evaluated in a different part of the process state. The expression e will first evaluate to a value v (rule ev/send). Communication is driven by rule ev/send1, which allows computation to continue in both the sender and the receiver.

Synchronous communication introduces the possibility of deadlocks. Without synchronous communication, the presence of a suspended atomic proposition $\mathsf{eval}\, e\, d$ always indicates the possibility of some transition, and the combination of a proposition $\mathsf{retn}\, v\, d$ and a continuation $\mathsf{cont}\, f\, d\, d'$ can either immediately transition or else are permanently in a stuck state. In [PS09], this observation motivated a classification of atomic propositions as *active* propositions like $\mathsf{eval}\, e\, d$ that independently drive computation, *passive* propositions like $\mathsf{cont}\, f\, d'\, d$ that do not drive computation, and *latent* propositions like $\mathsf{retn}\, f\, d$ that may or may not drive computation based on the ambient environment of passive propositions.

The specification in Figure 7.8 does not respect this classification because a proposition of the form $\mathsf{eval}\, (\mathsf{recv}\, c)\, d$ cannot immediately transition. We could restore this classification by having a rule $\forall c. \forall d. \mathsf{eval}\, (\mathsf{recv}\, c)\, d \rightarrowtail \{\mathsf{await}\, c\, d\}$ for some new passive linear predicate await and then replacing the premise $\mathsf{eval}\, (\mathsf{recv}\, C)\, D$ in ev/send1 with $\mathsf{await}\, C\, D$.

Labeled transitions

Substructural operational semantics specifications retain much of the flavor of abstract machines, in that we are usually manipulating expressions along with their continuations. In ordered specifications, continuations are connected to evaluating expressions and returning values only by their relative positions in the ordered context; in destination-passing specifications, expressions and values are connected to continuations by the threading of destinations.

Abstract machines are not always the most natural way to express a semantics. This observation is part of what motivated our discussion of the operationalization transformation from natural semantics (motto: "natural" is our first name!) and our informal discussion of statefully-modular natural semantics in Section 6.5.5. In Chapter 6, we showed that the continuation-focused perspective of SSOS allowed us to expose computation to the ambient state. With the example of synchronization above, we see that destination-passing SSOS specifications also expose compu-

```
bind: exp -> exp -> prop pers.    ; Future is complete
promise: dest -> exp -> prop lin. ; Future is waiting on a value

ev/bind:    eval X D * !bind X V >-> {retn V D}.
#| WAITING: eval X D * promise Dfuture X >-> ??? |#
ev/promise: retn V D * promise D X >-> {!bind X V}.

ev/flam:    eval (flam \x. E x) D >-> {retn (flam \x. E x) D}.

ev/fapp1:   retn (flam \x. E x) D1 * cont (app1 E2) D1 D
             >-> {Exists x. eval (E x) D *
                  Exists dfuture. eval E2 dfuture *
                  promise dfuture x}.
```

Figure 7.9: Semantics of call-by-future functions

tations in the process state to *other computations*, which is what allows the synchronization in rule ev/send1 to take place.

In small-step operational semantics, *labeled deduction* is used to describe specifications like the one above. At a high level, in a labeled transition system we inductively define a small step judgment $e \xmapsto{lab} e'$ with the property that

* $e \xmapsto{c!v} e'$ if e steps to e' by reducing some subterm $\mathsf{send}\, c\, v$, to $\langle\rangle$,
* $e \xmapsto{c?v} e'$ if e steps to e' by reducing some subterm $\mathsf{recv}\, c$ to v, and
* e_1 in parallel with e_2 (and possibly also in parallel with some other e_3, e_4, etc.) can step to e_1' in parallel with e_2' (and in parallel with an unchanged e_3, e_4, etc.) if $e_1 \xmapsto{c!v} e_1'$ and $e_2 \xmapsto{c?v} e_2'$.

Labels essentially serve to pass messages up through the inductive structure of a proposition. In destination-passing SSOS semantics, on the other hand, the internal structure of e is spread out as a series of frames throughout the context, and so the innermost redexes of terms can be directly connected. It would be interesting (but probably quite nontrivial) to consider a translation from labeled deduction systems to destination-passing SSOS specifications along the lines of the operationalization transformation.

7.2.3 Futures

Futures can be seen as a parallel version of call-by-value, and the presentation in Figure 7.9 can be compared to the environment semantics for call-by-value in Figure 6.19. We introduce future-functions as a new kind of function $\mathsf{flam}\ \lambda x.e$ comparable to plain-vanilla call-by-value functions $\mathsf{lam}\ \lambda x.e$, lazy call-by-need functions $\mathsf{lazylam}\ \lambda x.e$, and environment-semantics functions $\mathsf{envlam}\ \lambda x.e$. As in the environment semantics specification, when a call-by-future function returns to a frame $\ulcorner \Box\, e_2 \urcorner = \mathsf{app1} \ulcorner e_2 \urcorner$, we create a new expression x by existential quantification. However, instead of suspending the function body on the stack as we did in Figure 6.19,

$x_1{:}\langle \mathsf{eval} \ulcorner (\lambda x.\mathsf{s}\, x)\, (\mathsf{s}\, \mathsf{z}) \urcorner\, d_1 \rangle$

$x_2{:}\langle \mathsf{eval} \ulcorner \lambda x.\mathsf{s}\, x \urcorner\, d_2 \rangle,\ \ x_3{:}\langle \mathsf{cont} \ulcorner \square\, (\mathsf{s}\, \mathsf{z}) \urcorner\, d_2\, d_1 \rangle$

$x_3{:}\langle \mathsf{retn} \ulcorner \lambda x.\mathsf{s}\, x \urcorner\, d_2 \rangle,\ \ x_3{:}\langle \mathsf{cont} \ulcorner \square\, (\mathsf{s}\, \mathsf{z}) \urcorner\, d_2\, d_1 \rangle$

In this phase, the two computations proceed in parallel

$x_4{:}\langle \mathsf{eval}\, (\mathsf{succ}\, x)\, d_1 \rangle,\ \ x_5{:}\langle \mathsf{eval} \ulcorner \mathsf{s}\, \mathsf{z} \urcorner\, d_3 \rangle,\ \ x_6{:}\langle \mathsf{promise}\, d_3\, x \rangle$

$x_4{:}\langle \mathsf{eval}\, (\mathsf{succ}\, x)\, d_1 \rangle,\ \ x_7{:}\langle \mathsf{eval} \ulcorner \mathsf{z} \urcorner\, d_4 \rangle,\ \ x_8{:}\langle \mathsf{cont}\, \mathsf{succ1}\, d_4\, d_3 \rangle,\ \ x_6{:}\langle \mathsf{promise}\, d_3\, x \rangle$

$x_4{:}\langle \mathsf{eval}\, x\, d_5 \rangle,\ \ x_9{:}\langle \mathsf{cont}\, \mathsf{succ1}\, d_5\, d_1 \rangle,\ \ x_7{:}\langle \mathsf{eval} \ulcorner \mathsf{z} \urcorner\, d_4 \rangle,\ \ x_8{:}\langle \mathsf{cont}\, \mathsf{succ1}\, d_4\, d_3 \rangle,\ \ x_6{:}\langle \mathsf{promise}\, d_3\, x \rangle$

$x_4{:}\langle \mathsf{eval}\, x\, d_5 \rangle,\ \ x_9{:}\langle \mathsf{cont}\, \mathsf{succ1}\, d_5\, d_1 \rangle,\ \ x_{10}{:}\langle \mathsf{retn} \ulcorner \mathsf{z} \urcorner\, d_4 \rangle,\ \ x_8{:}\langle \mathsf{cont}\, \mathsf{succ1}\, d_4\, d_3 \rangle,\ \ x_6{:}\langle \mathsf{promise}\, d_3\, x \rangle$

$x_4{:}\langle \mathsf{eval}\, x\, d_5 \rangle,\ \ x_9{:}\langle \mathsf{cont}\, \mathsf{succ1}\, d_5\, d_1 \rangle,\ \ x_{11}{:}\langle \mathsf{retn} \ulcorner \mathsf{s}\, \mathsf{z} \urcorner\, d_3 \rangle,\ \ x_6{:}\langle \mathsf{promise}\, d_3\, x \rangle$

$x_4{:}\langle \mathsf{eval}\, x\, d_5 \rangle,\ \ x_9{:}\langle \mathsf{cont}\, \mathsf{succ1}\, d_5\, d_1 \rangle,\ \ x_{12}{:}\langle \mathsf{bind}\, x \ulcorner \mathsf{s}\, \mathsf{z} \urcorner \rangle\ \mathit{pers}$

$x_{13}{:}\langle \mathsf{retn} \ulcorner \mathsf{s}\, \mathsf{z} \urcorner\, d_5 \rangle,\ \ x_9{:}\langle \mathsf{cont}\, \mathsf{succ1}\, d_5\, d_1 \rangle,\ \ x_{12}{:}\langle \mathsf{bind}\, x \ulcorner \mathsf{s}\, \mathsf{z} \urcorner \rangle\ \mathit{pers}$

$x_{14}{:}\langle \mathsf{retn} \ulcorner \mathsf{s}\, (\mathsf{s}\, \mathsf{z}) \urcorner\, d_1 \rangle,\ \ x_{12}{:}\langle \mathsf{bind}\, x \ulcorner \mathsf{s}\, \mathsf{z} \urcorner \rangle\ \mathit{pers}$

In this phase, the primary computation is stuck while it waits on the promise

Remember: bind *is persistent, all other propositions are linear*

Figure 7.10: Series of process states in an example call-by-future evaluation

in Figure 7.9 we create a new destination $\mathit{dfuture}$ and start evaluating the function argument towards that destination (rule ev/fapp1). We also create a linear proposition – promise $\mathit{dfuture}\ x$ – that will take any value returned to $\mathit{dfuture}$ and permanently bind it to x (rule ev/promise). As a proposition that only exists during the course of evaluating the argument, promise is analogous to the black hole in our specification of lazy call-by-need.

Futures use destinations to create new and potentially disconnected threads of computation, which can be seen in the example evaluation of $(\lambda x.\mathsf{s}\, x)\, (\mathsf{s}\, \mathsf{z})$ – where $\lambda x.e$ is interpreted as a future function flam instead of lam as before – given in Figure 7.10. That figure illustrates how spawning a future splits the destination structure of the ordered context into two disconnected threads of computation. This was not possible in the ordered framework where every computation had to be somewhere specific in the ordered context relative to the current computation – either to the left, or to the right. These threads are connected not by destinations but by the variable x, which the primary computation needs the future to return before it can proceed.

Note the similarity between the commented-out rule fragment in Figure 7.9 and the commented out rule fragments in the specifications of call-by-need evaluation (Section 6.5.2). In the call-by-need specifications, needing an unavailable value was immediately fatal. With specifications, needing an unavailable value is not immediately fatal: the main thread of computation is stuck, but only until the future's promise is fulfilled. (This again violates the classification of eval as active; as before, this could again be fixed by adding a new latent proposition.)

The destination-passing semantics of futures interact seamlessly with the semantics of synchronization and parallelism, but not with the semantics of recoverable failure: we would have to make some choice about what to do when a future signals failure.

```
eval: exp -> dest -> prop lin.
retn: exp -> dest -> prop lin.
cont: frame -> dest -> dest -> prop pers.

ev/letcc:  eval (letcc \x. E x) D >-> {eval (E (contn D)) D}.
ev/throw2: retn (contn DK) D2 * !cont (throw2 V1) D2 D
            >-> {retn V1 DK}.
```

Figure 7.11: Semantics of first-class continuations (with letcc)

7.2.4 First-class continuations

First-class continuations are a sophisticated control feature. *Continuations* are another name for the stacks k in abstract machine semantics with states $k \triangleright e$ and $k \triangleleft v$ (and also, potentially, $k\blacktriangleleft$ if we want to be able to return errors, as discussed in Section 6.5.4). First-class continuations introduce a new value, $\mathsf{contn}\, k$, to the language. Programmers cannot write continuations k directly, just as they cannot write locations l directly; rather, the expression $\ulcorner \mathsf{letcc}\, x.e \urcorner = \mathsf{letcc}\, \lambda x.\ulcorner e \urcorner$ captures the current expression as a continuation:

$$k \triangleright \mathsf{letcc}\, x.e \;\mapsto\; k \triangleright [\mathsf{contn}\, k/x]e$$

There is a third construct, $\ulcorner \mathsf{throw}\, e_1\, \mathsf{to}\, e_2 \urcorner = \mathsf{throw} \ulcorner e_1 \urcorner \ulcorner e_2 \urcorner$ that evaluates e_1 to a value v_1, evaluates e_2 to a continuation value $\mathsf{cont}\, k'$, and then throws away the current continuation in favor of returning v_1 to k':

$$\begin{aligned} k \triangleright \mathsf{throw}\, e_1\, \mathsf{to}\, e_2 &\mapsto (k; \mathsf{throw}\, \square\, \mathsf{to}\, e_2) \triangleright e_1 \\ (k; \mathsf{throw}\, \square\, \mathsf{to}\, e_2) \triangleleft v_1 &\mapsto (k; \mathsf{throw}\, v_1\, \mathsf{to}\, \square) \triangleright e_2 \\ (k; \mathsf{throw}\, v_1\, \mathsf{to}\, \square) \triangleleft \mathsf{contn}\, k' &\mapsto k' \triangleleft v_1 \end{aligned}$$

When handled in a typed setting, a programming language with first-class continuations can be seen as a Curry-Howard interpretation of classical logic.

In destination-passing SSOS specifications, we never represent continuations or control stacks k directly. However, we showed in Section 6.3 that a control stack k is encoded in the context as a series of cont frames. In a destination-passing specification, it is therefore reasonable to associate a k continuation with the destination d that points to the topmost frame $\mathsf{cont}\, f\, d\, d'$ in the stack k encoded in the process state. Destinations stand for continuations in much the same way that introduced variables x in the environment semantics stand for the values v they are bound to through persistent $\mathsf{bind}\, x\, v$ propositions. In Figure 7.11, the rule ev/letcc captures the current continuation d as an expression $\mathsf{cont}\, d$ that is substituted into the subexpression. In rule ev/throw2, the destination dk held by the value $\mathsf{contn}\, dk$ gets the value v_1 returned to it; the previous continuation, represented by the destination d, is abandoned.

Just as it is critical for the bind predicate in the environment semantics to be persistent, it is necessary, when dealing with first-class-continuations, to have the cont predicate be persistent. As discussed in Section 7.1.2, it does not change the behavior of any SSOS specifications we have discussed if linear cont predicates are turned into persistent cont predicates.

Turning cont into a persistent predicate does not, on its own, influence the transitions that are possible, so in a sense we have not changed our SSOS semantics very much in order to add first-class continuations. However, the implicit representation of stacks in the context does complicate adequacy arguments for the semantics in Figure 7.11 relative to the transition rules given above. We will return to this point in Section 9.6 when we discuss generative invariants that apply to specifications using first-class continuations.

Chapter 8

Linear logical approximation

A general recipe for constructing a sound program analysis is to (1) specify the operational semantics of the underlying programming language via an interpreter, and (2) specify a terminating approximation of the interpreter itself. This is the basic idea behind *abstract interpretation* [CC77], which provides techniques for constructing approximations (for example, by exhibiting a Galois connection between concrete and abstract domains). The correctness proof must establish the appropriate relationship between the concrete and abstract computations and show that the abstract computation terminates. We need to vary both the specification of the operational semantics and the form of the approximation in order to obtain various kinds of program analyses, sometimes with considerable ingenuity.

In this chapter, which is mostly derived from [SP11a], we consider a new class of instances in the general schema of abstract interpretation that is based on the approximation of SSOS specifications in SLS. We apply logically justified techniques for manipulating and approximating SSOS specifications to yield approximations that are correct by construction. The resulting persistent logical specifications can be interpreted and executed as saturating logic programs, which means that derived specifications are executable program analyses.

The process described in this chapter does not claim to capture or derive all possible interesting program analyses. The methodology we describe only derives over-approximations (or *may*- analyses) that ensure all possible behaviors will be reported by the analysis. There is a whole separate class of under-approximations (or *must*- analyses) which ensure that if a behavior is reported by the analysis it is possible; we will not consider under-approximations here [GNRT10]. Instead, we argue for the utility of our methodology by deriving two fundamental and rather different over-approximation-based analyses: a context-insensitive control flow analysis (Section 8.4) and an alias analysis (Section 8.5). Might and Van Horn's closely related "abstracting abstract machines" methodology, described in Section 8.6 along with other related work, suggests many more examples.

8.1 Saturating logic programming

Concurrent SLS specifications where all positive atomic propositions are persistent (and where all inclusions of negative propositions in positive propositions – if there are any – have the form

$!A^-$, not $\downarrow A^-$ or $¡A^-$) have a distinct logical and operational character. Logically, by the discussion in Section 3.7 we are justified in reading such specifications as specifications in persistent intuitionistic logic or persistent lax logic. Operationally, while persistent specifications have an interpretation as transition systems, that interpretation is not very useful. This is because if we can take a transition once – for instance, using the rule $\mathsf{a} \rightarrowtail \{\mathsf{b}\}$ to derive the persistent atomic proposition b from the persistent atomic proposition a – none of the facts that enabled that transition can be consumed, as all facts are persistent. Therefore, we can continue to make the same transition indefinitely; in the above-mentioned example, such transitions will derive multiple redundant copies of b.

The way we will understand the meaning of persistent and concurrent SLS specifications is in terms of *saturation*. A process state $(\Psi; \Delta)$ is saturated relative to the signature Σ if, for any step $(\Psi; \Delta) \rightsquigarrow_\Sigma (\Psi'; \Delta')$, it is the case that Ψ and Ψ' are the same (the step unified no distinct variables and introduced no new variables), $x{:}\langle p^+_{pers}\rangle \in \Delta'$ implies $x{:}\langle p^+_{pers}\rangle \in \Delta$, and $x{:}A^-\ pers \in \Delta'$ implies $x{:}A^-\ pers \in \Delta$. This means that a signature with a rule that produces new variables by existential quantification, like $\mathsf{a} \rightarrowtail \{\exists x.\mathsf{b}(x)\}$ has no saturated process states where a is present. We will cope with rules of this form by turning them into rules of the form $\mathsf{a} \rightarrowtail \{\exists x.\mathsf{b}(x) \bullet x \doteq t\}$ for some t, neutralizing the free existential variable as a notational definition. Notions of saturation that can cope with free existentially generated variables in other ways are interesting, but are beyond the scope of this dissertation.

A *minimal* saturated process state is one with no duplicated propositions; we can compute a minimal process state from any saturated process state by removing duplicates. For purely persistent specifications and process states, minimal saturated process states are unique when they exist: if $(\Psi; \Delta) \rightsquigarrow^*_\Sigma (\Psi_1; \Delta_1)$ and $(\Psi; \Delta) \rightsquigarrow^*_\Sigma (\Psi_2; \Delta_2)$ and both $(\Psi_1; \Delta_1)$ and $(\Psi_2; \Delta_2)$ are saturated, then $(\Psi_1; \Delta_1)$ and $(\Psi_2; \Delta_2)$ have minimal process states that differ only in the names of variables.

Furthermore, if a saturated process state exists for a given initial process state, the minimal saturated process state can be computed by the usual forward-chaining semantics where only transitions that derive new persistent atomic propositions or equalities $t \doteq s$ are allowed. This forward-chaining logic programming interpretation of persistent logic is extremely common, usually associated with the logic programming language Datalog. A generalization of Datalog formed the basis of McAllester and Ganzinger's meta-complexity results: they gave a cost semantics to their logic programming language, and then they used that cost semantics to argue that many program analyses could be *efficiently* implemented as logic programs [McA02, GM02]. Persistent SLS specifications can be seen as an extension of McAllester and Ganzinger's language (and, transitively, as a generalization of Datalog). We will not deal with cost semantics or efficiency, however, as our use of higher-order abstract syntax appears to complicate McAllester and Ganzinger's cost semantics.

Just as the term *persistent logic* was introduced in Chapter 2 to distinguish what is traditionally referred to as intuitionistic logic from intuitionistic ordered and linear logic, we will use the term *saturating logic programming* to distinguish what is traditionally referred to as forward-chaining logic programming from the forward-chaining logic programming interpretation that makes sense for ordered and linear logical specifications. There is a useful variant of substructural forward chaining, forward chaining with *quiescence* [LPPW05], that acts like saturating logic programming on purely-persistent specifications and like simple committed-choice

```
hd: dest -> dest -> prop pers.
left: tok -> dest -> dest -> prop pers.
right: tok -> dest -> dest -> prop pers.
stack: tok -> dest -> dest -> prop pers.

push: hd L M * left X M R
       >-> {Exists m. stack X L m * hd m R * m == fm X L M R}.

pop:  stack X L M1 * hd M1 M2 * right X M2 R >-> {hd L R}.
```

Figure 8.1: Skolemized approximate version of the PDA specification from Figure 7.2

forward chaining on specifications with no persistent propositions. We refined this interpretation and gave it a cost semantics in [SP08], but this more sophisticated interpretation is not relevant to the examples in this dissertation.

8.2 Using approximation

The meta-approximation theorem that we present in the next section gives us a way of building abstractions from specifications and initial process states: we interpret the approximate version of the program as a saturating logic program over that initial state. If we can obtain a saturated process state using the logic programming interpretation, it is an abstraction of the initial process state. It is not always possible to obtain a saturated process state using the logic programming interpretation, however: rules like $\forall x.\, \mathsf{a}(x) \rightarrowtail \{\mathsf{a}(\mathsf{s}(x))\}$ and $\forall x.\, \mathsf{a}(x) \rightarrowtail \{\exists y.\mathsf{a}(y)\}$ lead to non-termination when interpreted as saturating logic programs. Important classes of programs are known to terminate in all cases, such as those in the Datalog fragment where the only terms in the program are variables and constants. Structured terms (like expressions encoded in the LF type exp) fall outside the Datalog fragment.

Consider the destination-passing PDA specification from Figure 7.2. If we simply turn all linear predicates persistent, the first step in the approximation methodology, then the push rule will lead to non-termination because the head $\exists m.\mathsf{stack}\, x\, l\, m \bullet \mathsf{hd}\, m\, r$ introduces a new existential parameter m. We can cope by adding a new conclusion $m \doteq t$; adding new conclusions is the second step in the approximation methodology. This, however, means we have to pick a t. The most general starting point for selecting a t is to apply Skolemization to the rule. By moving the existential quantifier for m in front of the implicitly quantified X, L, M, and R, we get a Skolem function $\mathsf{fm}\, X\, L\, M\, R$ that takes four arguments. Letting $t = \mathsf{fm}\, X\, L\, M\, R$ results in the SLS specification/logic program shown in Figure 8.1. (Remember that, because the specification in Figure 8.1 is purely persistent, we will omit the optional `!` annotation described in Section 4.5, writing `hd L M` instead of `!hd L M` and so on.)

Notice that we have effectively taken a specification that freely introduces existential quantification (and that therefore *definitely* will not terminate when interpreted as a saturating logic program) and produced a specification that uses structured terms $\mathsf{fm}\, X\, L\, R\, M$. But the introduction of structured terms takes us outside the Datalog fragment, which may also lead to non-

termination! This is not as bad as it may seem: when we want to treat a specification with structured terms as a saturating logic program, it is simply necessary to reason explicitly about termination. Giving any finite upper bound on the number of derivable facts is a simple and sufficient criteria for showing that a saturating logic program terminates.

Skolem functions provide a natural starting point for approximations, even though the Skolem constant that arises directly from Skolemization is usually more precise than we want. From the starting point in Figure 8.1, however, we can define approximations simply by instantiating the Skolem constant. For instance, we can equate the existentially generated destination in the conclusion with the one given in the premise (letting $\mathsf{fm} = \lambda X.\lambda L.\lambda M.\lambda R.\, M$). The result is equivalent to this specification:

```
push: hd L M * left X M R >-> {stack X L M * hd M R}.
pop:  stack X L M1 * hd M1 M2 * right X M2 R >-> {hd L R}.
```

This substitution yields a precise approximation that exactly captures the behavior of the original PDA as a saturating logic program.

To be concrete about what this means, let us recall how the PDA works and what it means for it to accept a string. To use the linear PDA specification in Figure 7.2, we encode a string as a sequence of linear atomic propositions $ptok_1 \ldots ptok_n$, where each $ptok_i$ either has the form $\mathsf{left}\, tok_i\, \mathsf{d_i}\, \mathsf{d_{i+1}}$ or the form $\mathsf{right}\, tok_i\, \mathsf{d_i}\, \mathsf{d_{i+1}}$. The term tok_i that indicates whether we're talking about a left/right parenthesis, curly brace, square brace, etc., and $\mathsf{d_0} \ldots \mathsf{d_{n+1}}$ are $n+2$ constants of type *dest*.[1] Let $\Delta = (h{:}\langle \mathsf{hd}\, \mathsf{d_0}\, \mathsf{d_1}\rangle\, eph, x_1{:}\langle ptok_1\rangle\, eph, \ldots, x_n{:}\langle ptok_n\rangle\, eph)$. The PDA accepts the string encoded as $ptok_1 \ldots ptok_n$ if and only if there is a trace under the signature in Figure 7.2 where $\Delta \rightsquigarrow^* (\Psi'; x{:}\langle \mathsf{hd}\, \mathsf{d_0}\, \mathsf{d_{n+1}}\rangle\, eph)$.

Now, say that we turn the predicates persistent and run the program described by the push and pop rules above as a saturating logic program, obtaining a saturated process state Δ_{sat} from the initial process state $(h{:}\langle \mathsf{hd}\, d_0\, d_1\rangle\, pers, x_1{:}\langle ptok_1\rangle\, pers, \ldots, x_n{:}\langle ptok_n\rangle\, pers)$. (We can see from the structure of the program that LF context will remain empty.) The meta-approximation theorem *ensures* that, if the original PDA accepted, then the proposition $\mathsf{hd}\, \mathsf{d_0}\, \mathsf{d_{n+1}}$ is in Δ_{sat}. It just so happens to be the case that the converse is also true – if $\mathsf{hd}\, \mathsf{d_0}\, \mathsf{d_{n+1}}$ is in Δ_{sat}, the original PDA specification accepts the string. That is why we say we have a precise approximation.

On the other hand, if we set m equal to l (letting $\mathsf{fm} = \lambda X.\lambda L.\lambda M.\lambda R.\, L$), the result is equivalent to this specification:

```
push: hd L M * left X M R >-> {stack X L L * hd L R}.
pop:  stack X L M1 * hd M1 M2 * right X M2 R >-> {hd L R}.
```

If the initial process state contains a single atomic proposition $\mathsf{hd}\, \mathsf{d_0}\, \mathsf{d_1}$ in addition to all the left and right facts, then the two rules above maintain the invariant that, as new facts are derived, the first argument of hd and the second and third arguments of stack will *always* be $\mathsf{d_0}$. These arguments are therefore vestigial, like the extra arguments to eval and retn discussed in Section 7.1.1, and we can remove them from the approximate specification, resulting in the specification in Figure 8.2.

[1] We previously saw destinations as only inhabited by parameters, but the guarantees given by the meta-approximation theorem are clearer when the initial state contains destinations that are constants.

```
hd: dest -> prop pers.
left: tok -> dest -> dest -> prop pers.
right: tok -> dest -> dest -> prop pers.
stack: tok -> prop pers.

push: hd M * left X M R >-> {stack X * hd R}.
pop:  stack X * hd M2 * right X M2 R >-> {hd R}.
```

Figure 8.2: Approximated PDA specification

This logical approximation of the original PDA accepts if we run saturating logic programming from the initial process state $(h{:}\langle \mathsf{hd}\ \mathsf{d}_1\rangle\ pers, x_1{:}\langle ptok_1\rangle\ pers, \ldots, x_n{:}\langle ptok_n\rangle\ pers)$ and $\mathsf{hd}\ \mathsf{d}_{n+1}$ appears in the saturated process state. Again, the meta-approximation theorem ensures that any string accepted by the original PDA will also be accepted by any approximation. This approximation will additionally accept every string where, for every form of bracket tok, at least one left tok appears before any of the right tok. The string [ ]]] (( ) would be accepted by this approximated PDA, but the string ( )] [[ ] would not, as the first right square bracket appears before any left square bracket.

8.3 Logical transformation: approximation

The approximation strategy demonstrated in the previous section is quite simple: a signature in an ordered or linear logical specification can be approximated by making all atomic propositions persistent, and a flat rule $\forall\overline{x}.\, A^+ \rightarrowtail \{B^+\}$ containing only persistent atomic propositions can be further approximated by removing premises from A^+ and adding conclusions to B^+. Of particular practical importance are added conclusions that neutralize an existential quantification with a notational definition. The approximation procedure doesn't force us to neutralize all such variables in this way. However, as we explained above, failing to do so almost ensures that the specification cannot be run as a saturating logic program, and being able to interpret specifications as saturating logic programs is a prerequisite for applying the meta-approximation theorem (Theorem 8.4).

First, we define what it means for a specification to be an approximate version of another specification:

Definition 8.1. *A flat, concurrent, and persistent specification Σ_a is an* approximate version *of another specification Σ if every predicate* $\mathsf{a} : \Pi x_1{:}\tau_1 \ldots \Pi x_n{:}\tau_n.\ \mathsf{prop}\ \mathsf{lvl}$ *declared in Σ has a corresponding predicate* $\mathsf{a} : \Pi x_1{:}\tau_1 \ldots \Pi x_n{:}\tau_n.\ \mathsf{prop}\ \mathsf{pers}$ *in Σ_a and if for every rule* $\mathsf{r} : \forall\overline{x}.\, A_1^+ \rightarrowtail \{A_2^+\}$ *in Σ there is a corresponding rule* $\mathsf{r} : \forall\overline{x}.\, B_1^+ \rightarrowtail \{B_2^+\}$ *in Σ_a such that:*

- *The existential quantifiers in A_1^+ and A_2^+ are identical to the existential quantifiers in B_1^+ and B_2^+ (respectively),*
- *For each premise (p_{pers}^+ or $t \doteq s$) in B_1^+, the same premise appears in A_1^+, and*
- *For each conclusion (p_{lvl}^+ or $t \doteq s$) in A_2^+, the same premise appears in B_2^+.*

While approximation is a program transformation, it is not a deterministic one: Definition 8.1 describes a whole family of potential approximations. Even the nondeterministic operationalization transformation was just a bit nondeterministic, giving several options for operationalizing any given deductive rule. The approximation transformation, in contrast, needs explicit information from the user: which premises should be removed, and what new conclusions should be introduced? While there is value in actually implementing the operationalization, defunctionalization, and destination-adding transformations, applying approximation requires intelligence. Borrowing a phrase from Danvy, approximation is a candidate for "mechanization by graduate student" rather than mechanization by computer.

Next, we give a definition of what it means for a state to be an approximate version (we use the word "generalization") of another state or a family of states.

Definition 8.2. *The persistent process state* $(\Psi_g; \Delta_g)$ *is a* generalization *of the process state* $(\Psi; \Delta)$ *if there is a substitution* $\Psi_g \vdash \sigma : \Psi$ *such that, for all atomic propositions* $p^+_{lvl} = \mathsf{a}\, t_1 \ldots t_n$ *in* Δ*, there exists a persistent proposition* $p^+_{pers} = \mathsf{a}\, (\sigma t_1) \ldots (\sigma t_n)$ *in* Δ_g*.*

One thing we might prove about the relationship between process states and their generalizations is that generalizations can *simulate* the process states they generalize: that is, if $(\Psi_g; \Delta_g)$ is a generalization of $(\Psi; \Delta)$ and $(\Psi; \Delta) \rightsquigarrow_\Sigma (\Psi'; \Delta')$ then $(\Psi_g; \Delta_g) \rightsquigarrow_{\Sigma_a} (\Psi'_g; \Delta'_g)$ where $(\Psi'_g; \Delta'_g)$ is a generalization of $(\Psi'; \Delta')$. This property, *one-step simulation*, is true [SP11a, Lemma 6], and we will prove it as a corollary on the way to the proof of Theorem 8.4. However, we are not interested in generalization per se; rather, we're interested in a stronger property, *abstraction*, that is defined in terms of generalization:

Definition 8.3. *A process state* $(\Psi_a; \Delta_a)$ *is an* abstraction *of* $(\Psi_0; \Delta_0)$ *under the signature* Σ *if, for any trace* $(\Psi_0; \Delta_0) \rightsquigarrow^*_\Sigma (\Psi_n; \Delta_n)$*,* $(\Psi_a; \Delta_a)$ *is a generalization of* $(\Psi_n; \Delta_n)$*.*

An abstraction of the process state $(\Psi_0; \Delta_0)$ is therefore a single process state that captures *all possible future behaviors* of the state $(\Psi_0; \Delta_0)$ because, for any atomic proposition $p^+_{lvl} = \mathsf{a}\, t_1 \ldots t_n$ that may be derived by evolving $(\Psi_0; \Delta_0)$, there is a substitution σ such that $\mathsf{a}\, (\sigma t_1) \ldots (\sigma t_n)$ is already present in the abstraction. The meta-approximation theorem relates this definition of abstraction to the concept of approximate versions of programs as specified by Definition 8.1.

Theorem 8.4 (Meta-approximation)**.** *If* Σ_a *is an approximate version of* Σ*, and if there is a state* $(\Psi_0; \Delta_0)$ *well-formed according to* Σ*, and if for some* $\Psi'_0 \vdash \sigma : \Psi_0$ *there is a trace* $(\Psi'_0; \sigma\Delta_0) \rightsquigarrow^*_{\Sigma_a} (\Psi_a; \Delta_a)$ *such that* $(\Psi_a; \Delta_a)$ *is a saturated process state, then* $(\Psi_a; \Delta_a)$ *is an abstraction of* $(\Psi_0; \Delta_0)$*.*

Proof. The central lemma is one-step simulation, mentioned above, which is established by induction on the structure of the step. A multi-step *simulation* lemma immediately follows by induction on traces: If Σ_a is an approximate version of Σ, $(\Psi_g; \Delta_g)$ is a generalization of $(\Psi; \Delta)$ and $(\Psi; \Delta) \rightsquigarrow^*_\Sigma (\Psi'; \Delta')$ then $(\Psi_g; \Delta_g) \rightsquigarrow^*_{\Sigma_a} (\Psi'_g; \Delta'_g)$ where $(\Psi'_g; \Delta'_g)$ is a generalization of $(\Psi'; \Delta')$ [SP11a, Lemma 7].

The *monotonicity* lemma establishes that transitions in a purely-persistent specification only increase the generality of a process state: if $(\Psi; \Delta) \rightsquigarrow^*_\Sigma (\Psi'; \Delta')$ and Σ defines no ordered or mobile predicates, then $(\Psi'; \Delta')$ is a generalization of $(\Psi; \Delta)$ [SP11a, Lemma 8].

We use the monotonicity lemma to prove the *saturation* lemma: if $(\Psi; \Delta) \rightsquigarrow^*_\Sigma (\Psi_s; \Delta_s)$, Σ defines no ordered or mobile predicates, and $(\Psi_s; \Delta_s)$ is saturated, then whenever $(\Psi; \Delta) \rightsquigarrow^*_\Sigma (\Psi'; \Delta')$ $(\Psi_s; \Delta_s)$ is a generalization of $(\Psi'; \Delta')$. The proof proceeds by induction on the last steps of the trace witnessing $(\Psi; \Delta) \rightsquigarrow^*_\Sigma (\Psi'; \Delta')$.

* In the base case, $(\Psi; \Delta) = (\Psi'; \Delta')$ and we appeal to monotonicity.
* In the inductive case, we have $(\Psi; \Delta) \rightsquigarrow^*_\Sigma (\Psi''; \Delta'') \rightsquigarrow_\Sigma (\Psi'; \Delta')$. By the induction hypothesis we have that $(\Psi_s; \Delta_s)$ is a generalization of $(\Psi''; \Delta'')$, and by one-step simulation $(\Psi_s; \Delta_s) \rightsquigarrow_\Sigma (\Psi'_s; \Delta'_s)$ such that $(\Psi'_s; \Delta'_s)$ is a generalization of $(\Psi'; \Delta')$. But saturation means that $\Psi_s = \Psi'_s$ and that all the propositions in Δ'_s already appear in Δ_s, so $(\Psi_s; \Delta_s)$ must be a generalization of $(\Psi'; \Delta')$ as well. [SP11a, Lemma 9]

Finally, we prove meta-approximation. Consider a trace $(\Psi_0; \Delta_0) \rightsquigarrow^*_\Sigma (\Psi_n; \Delta_n)$ of the original program. By the simulation lemma, there is a trace $(\Psi_0; \sigma\Delta_0) \rightsquigarrow^*_{\Sigma_a} (\Psi'_n; \Delta'_n)$ where $(\Psi'_n; \Delta'_n)$ is a generalization of $(\Psi_n; \Delta_n)$. By the saturation lemma, $(\Psi_a; \Delta_a)$ is a generalization of $(\Psi'_n; \Delta'_n)$, and so because generalization is transitive, $(\Psi_a; \Delta_a)$ is a generalization of $(\Psi_0; \Delta_0)$, which is what we needed to show [SP11a, Theorem 3]. □

The meaning of the meta-approximation theorem is that if (1) we can approximate a specification and an initial state and (2) we can obtain a saturated process state from that approximate specification and approximate initial state, then the saturated process state captures all possible future behaviors of the (non-approximate) initial state.

8.4 Control flow analysis

The initial process state for destination-passing SSOS specifications generally has the form $(d{:}\mathsf{dest}; x{:}\langle \mathsf{eval}\, t\, d\rangle)$ for some program represented by the LF term $t = \ulcorner e \urcorner$. This means that we can use the meta-approximation result to derive abstractions from initial expressions e using the saturating logic programming interpretation of approximated SSOS specifications.

A control flow analysis is a fundamental analysis on functional programs, attributed to Shivers [Shi88]. It is used for taking an expression and "determining for each subexpression a hopefully small number of functions that it may evaluate to; thereby it will determine where the flow of control may be transferred to in the case where the subexpression is the operator of a function application" [NNH05, p. 142]. That is, we want to take a program and find, for every subexpression e of that unevaluated program, all the values v that the subexpression may evaluate to over the course of evaluating the program to a value. Because we are talking about subexpressions of the unevaluated program, the answer might not be unique. Consider the evaluation of $(\lambda f \ldots (f\,(\lambda y \ldots)) \ldots (f\,(\lambda z \ldots)) \ldots)\,(\lambda x.x)$. The function $\lambda x.x$ gets bound to f and therefore may get called twice, once with the argument $(\lambda y \ldots)$ and once with the argument $(\lambda z \ldots)$. The subexpression x of $\lambda x.x$ can therefore evaluate to $(\lambda y \ldots)$ in the context of the call $f\,(\lambda y \ldots)$

and to $(\lambda z \ldots)$ in the context of the call $f\,(\lambda z \ldots)$. As a *may*-analysis, the output of a control flow analysis is required to report both of these possibilities.[2]

When we *use* a control flow analysis, it is relevant that the calculation of which subexpressions evaluate to which values is done in service of a different goal: namely, determining which functions may be called from which calling sites. However, the ultimate goal of control flow analysis is irrelevant to our discussion of deriving control flow analyses from SSOS specifications, so we will concentrate on the question of which subexpressions evaluate to which values. Before we begin, however, we will address the issue of what it even means to be a (closed) subterm of an expression e that has been encoded with higher-order abstract syntax into the canonical forms of LF.

8.4.1 Subexpressions in higher-order abstract syntax

When given a term $\mathsf{a}\,(\mathsf{b}\,\mathsf{c}\,\mathsf{c})$, it is clear that there are three distinct subterms: the entire term, $\mathsf{b}\,\mathsf{c}\,\mathsf{c}$, and c. Therefore, it is meaningful to bound the size of a saturated process state using some function that depends on the number of subterms of the original term. But what are the subterms of $\mathsf{lam}\,(\lambda x.\,\mathsf{app}\,x\,x)$, and how can we write a saturating logic program that derives all those subterms? The rule for application is easy:

$$\mathsf{sub/app}: \quad \forall e_1{:}\mathsf{exp}.\,\forall e_2{:}\mathsf{exp}.\,\mathsf{subterms}(\mathsf{app}\,e_1\,e_2) \rightarrowtail \{\mathsf{subterms}\,e_1 \bullet \mathsf{subterms}\,e_2\}$$

What about the rule for lambda abstractions? Experience with LF says that, when we open up a binder, we should substitute a fresh variable into that binder. This would correspond to the following rule:

$$\mathsf{sub/lam/ohno}: \quad \forall e{:}\mathsf{exp} \rightarrow \mathsf{exp}.\,\mathsf{subterms}(\mathsf{lam}(\lambda x.e\,x)) \rightarrowtail \{\exists x.\,\mathsf{subterms}(e\,x)\}$$

The rule sub/lam/ohno will, as we have discussed, lead to nontermination when we interpret the rules as a saturating logic program. The solution is to apply Skolemization as described in Section 8.2, which introduces a new constant we will call var. The rule sub/lam/ohono can then be approximated as a terminating rule:

$$\mathsf{sub/lam}: \quad \forall e{:}\mathsf{exp} \rightarrow \mathsf{exp}.\,\mathsf{subterms}(\mathsf{lam}(\lambda x.e\,x)) \rightarrowtail \{\mathsf{subterms}(e(\mathsf{var}(\lambda x.e\,x)))\}$$

The subterms of any closed term e of LF type exp can then be enumerated by running this saturating logic program starting with the fact $\mathsf{subterms}(e)$, where subterms is a persistent positive proposition. We start counting subterms from the outside, and stop when we reach a variable represented by a term $\mathsf{var}(\lambda x.e)$. The logic program and discussion above imply that there are three distinct subterms of $\mathsf{lam}\,(\lambda x.\,\mathsf{app}\,x\,x)$: the entire term, $\mathsf{app}\,(\mathsf{var}(\lambda x.\,\mathsf{app}\,x\,x))\,(\mathsf{var}(\lambda x.\,\mathsf{app}\,x\,x))$, and $\mathsf{var}(\lambda x.\,\mathsf{app}\,x\,x)$.

Another solution, discussed in the next section, is to uniquely tag the lambda expression with a label. This has the same effect of allowing us to associate the variable x with a different concrete term, the tag, that represents the site where x was bound.

[2]This statement assumes that both of the calling sites $f\,(\lambda y \ldots)$ and $f\,(\lambda z \ldots)$ are reachable: the control flow analysis we derive performs some dead-code analysis, and it may not report that x evaluates to $(\lambda y \ldots)$, for instance, if the call $f\,(\lambda y \ldots)$ is certain to never occur.

```
bind: exp -> exp -> prop pers.
eval: exp -> dest -> prop lin.
retn: exp -> dest -> prop lin.
cont: frame -> dest -> dest -> prop lin.

ev/bind: eval X D * !bind X V >-> {retn V D}.

ev/lam:  eval (lam \x. E x) D >-> {retn (lam \x. E x) D}.

ev/app:  eval (app E1 E2) D
          >-> {Exists d1. eval E1 d1 * cont (app1 E2) d1 D}.

ev/app1: retn (lam \x. E x) D1 * cont (app1 E2) D1 D
          >-> {Exists d2. eval E2 d2 * cont (app2 \x. E x) d2 D}.

ev/app2: retn V2 D2 * cont (app2 \x. E x) D2 D
          >-> {Exists x. !bind x V *
               Exists d3. eval (E x) d3 * cont app3 d3 D}.

ev/app3: retn V D3 * cont app3 D3 D >-> {retn V D}.
```

Figure 8.3: Alternative environment semantics for CBV evaluation

8.4.2 Environment semantics

The starting point for deriving a control flow analysis is the environment semantics for call-by-value shown in Figure 8.3. It differs from the environment semantics shown in Figure 6.19 in three ways. First and foremost, it is a destination-passing specification instead of an ordered abstract machine specification, but that difference is accounted for by the destination-adding transformation in Chapter 7. A second difference is that the existentially generated parameter x associated with the persistent proposition $\mathsf{bind}\, x\, v$ is introduced as late as possible in the multi-stage protocol for evaluating an application (rule ev/app2 in Figure 8.3), not as early as possible (rule ev/appenv1 in Figure 6.19). The third difference is that there is an extra frame app3 and an extra rule ev/app3 that consumes such frames. The app3 frame is an important part of the control flow analysis we derive, but in [SP11a] the addition of these frames was otherwise unmotivated. Based on our discussion of the logical correspondence in Chapters 5 and 6, we now have a principled account for this extra frame and rule: it is precisely the pattern we get from operationalizing a natural semantics *without* tail-recursion optimization and then applying defunctionalization and destination-adding.

8.4.3 Approximation to 0CFA

In order for us to approximate Figure 8.3 to derive a finite control flow analysis, we turn all linear atomic propositions persistent and then must deal with the variables introduced by existential quantification. The variable x introduced in ev/app2 will be equated with $\mathsf{var}(\lambda x.E\, x)$, which is

```
bind: exp -> exp -> prop pers.
eval: exp -> exp -> prop pers.
retn: exp -> exp -> prop pers.
cont: frame -> exp -> exp -> prop pers.

ev/bind: eval X D * bind X V >-> {retn V D}.

ev/lam:  eval (lam \x. E x) D >-> {retn (lam \x. E x) D}.

ev/app:  eval (app E1 E2) D
          >-> {Exists d1. eval E1 d1 * cont (app1 E2) d1 D *
               d1 == E1}.

ev/app1: retn (lam \x. E x) D1 * cont (app1 E2) D1 D
          >-> {Exists d2. eval E2 d2 * cont (app2 \x. E x) d2 D *
               d2 == E2}.

ev/app2: retn V2 D2 * cont (app2 \x. E x) D2 D
          >-> {Exists x. bind x V *
               Exists d3. eval (E x) d3 * cont app3 d3 D *
               x == var (\x. E x) *
               d3 == E x}.

ev/app3: retn V D3 * cont app3 D3 D >-> {retn V D}.
```

Figure 8.4: A control-flow analysis derived from Figure 8.3

consistent with making $E\,x$ – which is now equal to $E(\mathsf{var}(\lambda x.E\,x))$ – a subterm of $\mathsf{lam}(\lambda x.\,E\,x)$. The new constructor var is also a simplified Skolem function for x that only mentions the LF term E; the most general Skolem function in this setting would have also been dependent on V, D, and D_2. The existentially generated variable x was also the first argument to bind, so bind, as a relation, will now associate binding sites and values instead of unique variables and values.

The discussion above pertains to the existentially generated variable x in rule ev/app2, but we still need some method for handling destinations d_1, d_2, and d_3 in ev/app, ev/app1, and ev/app2 (respectively). To this end, we need recall the question that we intend to answer with control flow analysis: what values may a given subexpression evaluate to? A destination passing specification attempts to return a value to a destination: we will instead return *to an expression* by equating destinations d with the expressions they represent. One way to do this would be to introduce a new constructor $\mathsf{d} : \mathsf{exp} \to \mathsf{dest}$, but we can equivalently conflate the two types exp and dest to get the specification in Figure 8.4.

The specification in Figure 8.4 has a point of redundancy along the lines of the redundancy in our second PDA approximation: the rules maintain the invariants that the two arguments to eval $e\,d$ are always the same. Therefore, the second argument to eval can be treated as vestigial; by removing that argument, we get a specification equivalent to Figure 8.5. That figure includes another simplifications as well: instead of introducing expressions d_1, d_2, and d_3 by existential

```
bind: exp -> exp -> prop pers.
eval: exp -> prop pers.
retn: exp -> exp -> prop pers.
cont: frame -> exp -> exp -> prop pers.

ev/bind: eval X * bind X V >-> {retn V X}.

ev/lam:  eval (lam \x. E x) >-> {retn (lam \x. E x) (lam \x. E x)}.

ev/app:  eval (app E1 E2) * E == app E1 E2
          >-> {eval E1 * cont (app1 E2) E1 E}.

ev/app1: retn (lam \x. E0 x) E1 * cont (app1 E2) E1 E
          >-> {eval E2 * cont (app2 \x. E0 x) E2 E}.

ev/app2: retn V2 E2 * cont (app2 \x. E0 x) E2 E
          >-> {Exists x. bind x V *
               eval (E0 x) * cont app3 (E0 x) E *
               x == var (\x. E0 x)}.

ev/app3: retn V E3 * cont app3 E3 E >-> {retn V E}.
```

Figure 8.5: Simplification of Figure 8.4 that eliminates the vestigial argument to eval

quantification just to equate them with expressions e_1, e_2, and e, we substitute in the equated expressions where the respective destinations appeared in Figure 8.4; this modification does not change anything at the level of synthetic inference rules.

Let's consider the termination of specification in Figure 8.5 interpreted as a saturating logic program. Fundamentally, the terms in the heads of rules are all subterms (in the generalized sense of Section 8.4.1), which is a sufficient condition for the termination of a saturating logic program. More specifically, consider that we start the database with a single fact $\mathsf{eval}\ulcorner e\urcorner$, where $\ulcorner e\urcorner$ has n subterms by the analysis in Section 8.4.1. We can only ever derive n new facts $\mathsf{eval}\, e$ – one for every subterm. If we deduced that every subexpression was a value that could be returned at every subexpression, there would still be only n^2 facts $\mathsf{retn}\, e\, e'$, and the same analysis holds for facts of the form $\mathsf{cont}\, \mathsf{app3}\, e\, e'$. A fact of the form $\mathsf{cont}\, (\mathsf{app1}\, e_2)\, e_1\, e$ will only be derived when $e = \mathsf{app}\, e_1\, e_2$, so there are at most n of these facts. A fact of the form $\mathsf{cont}\, (\mathsf{app2}\, \lambda x.\, e_0\, x)\, e_2\, e$ will only be derived when $e = \mathsf{app}\, e_1\, e_2$ for some e_1 that is also a subterm, so there are most n^2 of these facts too. This means that we can derive no more than $2n + 3n^2$ facts starting from a database containing $\mathsf{eval}\ulcorner e\urcorner$, where e has n subterms. We could give a much more precise analysis than this, but this imprecise analysis certainly bounds the size of the database, ensuring termination, which was our goal.

There is one important caveat to the control flow analysis we have derived. If for some value v we consider the program $\ulcorner((\lambda x.x)\,(\lambda y.y))\,v\urcorner$, we might expect a reasonable control flow analysis to notice that only $\ulcorner\lambda y.y\urcorner$ is passed to the function $\ulcorner\lambda x.x\urcorner$ and that only v is passed to the function $\ulcorner\lambda y.y\urcorner$. Because of our use of higher-order abstract syntax, however, $\ulcorner\lambda y.y\urcorner$ and

$\ulcorner \lambda x.x \urcorner$ are α-equivalent and therefore equal in the eyes of the logic programming interpreter. This is not a problem with correctness, but it means that our analysis may be less precise than expected, because the analysis distinguishes only subterms, not *subterm occurrences*. One solution would be to add distinct labels to terms, marking the α-equivalent $\lambda x.x$ and $\lambda y.y$ with their distinct positions in the overall term. Adding a label on the inside of every lambda-abstraction would seem to suffice, and in any real example labels would already be present in the form of source-code positions or line numbers. The alias analysis presented in the next section demonstrates the use of such labels.

8.4.4 Correctness

The termination analysis for the derived specification in Figure 8.5, together with the meta-approximation theorem (Theorem 8.4), ensures that we have derived some sort of program analysis. How do we know that it is a control flow analysis?

The easy option is to simply inspect the analysis and compare it to the behavior of the SSOS semantics whose behavior the analysis is approximating. Note that the third argument e to $\mathsf{cont}\, f\, e'\, e$ is always a term $\mathsf{app}\, e_1\, e_2$ – that is, a call site. The rule ev/app2 starts evaluating the function $\mathsf{lam}(\lambda x.e_0\, x)$ and generates the fact $\mathsf{cont}\, \mathsf{app3}\, (e(\mathsf{var}(\lambda x.e_0\, x)))\, e$. This means that, in the course of evaluating some initial expression e_{init}, the function $\mathsf{lam}(\lambda x.e_0\, x)$ may be called from the call site e only if $\mathsf{cont}\, \mathsf{app3}\, (e_0(\mathsf{var}(\lambda x.e_0\, x)))\, e$ appears in a saturated process state that includes the persistent atomic proposition $\mathsf{eval}(e_{init})$.

The analysis above is a bit informal, however. Following Nielson et al., an *acceptable control flow analysis* takes the form of two functions. The first, $\widehat{\mathsf{C}}$, is a function from expressions e to sets of values $\{v_1, \ldots, v_n\}$, and the second, $\widehat{\rho}$, is a function from variables x to sets of values $\{v_1, \ldots, v_n\}$. $\widehat{\mathsf{C}}$ and $\widehat{\rho}$ are said to represent an acceptable control flow analysis for the expression e if a coinductively defined judgment $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e$ holds.

We would like to interpret a saturated program state Δ as a (potentially acceptable) control flow analysis as follows (keeping in mind that, given our current interpretation of subterms, $\ulcorner x \urcorner = \mathsf{var}(\lambda x.\, E\, x)$ for some E):

* $\widehat{\mathsf{C}}(e) = \{v \mid \mathsf{retn} \ulcorner v \urcorner \ulcorner e \urcorner\}$, and
* $\widehat{\rho}(x) = \{v \mid \mathsf{bind} \ulcorner x \urcorner \ulcorner v \urcorner\}$.

Directly adapting Nielson et al.'s definition of an acceptable control flow analysis from [NNH05, Table 3.1] turns out not to work. The control flow analysis we derived in Figure 8.5 is rather sensitive to non-termination: if we let $\omega = (\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$, then our derived control flow analysis will not analyze the argument e_2 in an expression $\omega\, e_2$, nor will it analyze the function body e in an expression $(\lambda x.e)\, \omega$. Nielson et al.'s definition, on the other hand, demands that both e_2 in $\omega\, e_2$ and e in $(\lambda x.e)\, \omega$ be analyzed. In Exercise 3.4, of their book, Nielson et al. point out that a modified analysis, which takes order of evaluation into account, is possible.

We can carry out Nielson et al.'s Exercise 3.4 to get the definition of an acceptable control flow analysis given in Figure 8.6. Relative to this definition, it is possible to prove that the abstractions computed by the derived SLS specification in Figure 8.5 are acceptable control flow analyses.

$$
\begin{array}{ll}
[var] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models x \ \text{ iff } \ \widehat{\rho}(x) \subseteq \widehat{\mathsf{C}}(x) \\
[lam] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models \lambda x.e \ \text{ iff } \ \{(\lambda x.e)\} \subseteq \widehat{\mathsf{C}}(\lambda x.e) \\
[app] & (\widehat{\mathsf{C}}, \widehat{\rho}) \models e_1\, e_2 \ \text{ iff} \\
& \qquad (\widehat{\mathsf{C}}, \widehat{\rho}) \models e_1 \wedge \\
& \qquad (\forall (\lambda x.e_0) \in \widehat{\mathsf{C}}(e_1) : \\
& \qquad\quad (\widehat{\mathsf{C}}, \widehat{\rho}) \models e_2 \wedge \\
& \qquad\quad (\widehat{\mathsf{C}}(e_2) \subseteq \widehat{\rho}(x)) \wedge \\
& \qquad\quad (\forall (v) \in \widehat{\mathsf{C}}(e_2) : \\
& \qquad\qquad (\widehat{\mathsf{C}}, \widehat{\rho}) \models e_0 \wedge \\
& \qquad\qquad (\widehat{\mathsf{C}}(e_0) \subseteq \widehat{\mathsf{C}}(e_1\, e_2))))
\end{array}
$$

Figure 8.6: Coinductive definition of an acceptable control flow analysis

Theorem 8.5. *If Δ is a saturated process state that is well-formed according to the signature in Figure 8.5, and if $\widehat{\mathsf{C}}$ and $\widehat{\rho}$ are defined in terms of Δ as described above, then* $\mathsf{eval}\ulcorner e \urcorner \in \Delta$ *implies that* $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e$.

Proof. By coinduction on the definition of acceptability in Figure 8.6, and case analysis on the form of e.

- $e = x$, so $\ulcorner e \urcorner = \ulcorner x \urcorner = \mathsf{var}(\lambda x.\, E_0\, x)$
  We have to show $\widehat{\rho}(x) \subseteq \widehat{\mathsf{C}}(x)$. In other words, if $\mathsf{bind}\ulcorner x \urcorner \ulcorner v \urcorner \in \Delta$, then $\mathsf{retn}\ulcorner v \urcorner \ulcorner x \urcorner \in \Delta$. Because $\mathsf{eval}\ulcorner e \urcorner \in \Delta$, this follows by the presence of rule ev/bind – if $\mathsf{eval}\ulcorner e \urcorner \in \Delta$ and $\mathsf{bind}\ulcorner x \urcorner \ulcorner v \urcorner \in \Delta$, then $\mathsf{retn}\ulcorner v \urcorner \ulcorner x \urcorner \in \Delta$ as well; if it were not, the process state would not be saturated!

- $e = \lambda x.e$, so $\ulcorner e \urcorner = \ulcorner \lambda x.e_0 \urcorner = \mathsf{lam}(\lambda x.\, E_0\, x)$
  We have to show $\{(\lambda x.e)\} \subseteq \widehat{\mathsf{C}}(\lambda x.e)$. In other words, $\mathsf{retn}\ulcorner \lambda x.e \urcorner \ulcorner \lambda x.e \urcorner \in \Delta$. This follows by rule ev/lam by the same reasoning given above.

- $e = e_1\, e_2$, so $\ulcorner e \urcorner = \ulcorner e_1\, e_2 \urcorner = \mathsf{app}\, E_1\, E_2$
  We have to show several things. The first, that $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e_1$, follows from the coinduction hypothesis – by rule ev/app, $\mathsf{eval}\ulcorner e_1 \urcorner \in \Delta$. That rule also allows us to conclude that $\mathsf{cont}\ \mathsf{app1}\ulcorner e_2 \urcorner \ulcorner e_1 \urcorner \ulcorner e_1\, e_2 \urcorner \in \Delta$.

  Second, given a $(\lambda x.e_0) \in \widehat{\mathsf{C}}(e_1)$ (meaning $\mathsf{retn}\ulcorner \lambda x.e_0 \urcorner \ulcorner e_1 \urcorner \in \Delta$) we have to show that $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e_2$. This follows from the coinduction hypothesis: by rule ev/app1, because $\mathsf{retn}\ulcorner \lambda x.e_0 \urcorner \ulcorner e_1 \urcorner \in \Delta$ and $\mathsf{cont}\ (\mathsf{app1}\ulcorner e_2 \urcorner)\ulcorner e_1 \urcorner \ulcorner e_1\, e_2 \urcorner \in \Delta$, $\mathsf{eval}\ulcorner e_2 \urcorner \in \Delta$. This same reasoning allows us to conclude that $\mathsf{cont}\ (\mathsf{app2}\,(\lambda x.\ulcorner e_0 \urcorner))\ulcorner e_2 \urcorner \ulcorner e_1\, e_2 \urcorner \in \Delta$ given that $(\lambda x.e_0) \in \widehat{\mathsf{C}}(e_1)$.

Third, given a $(\lambda x.e_0) \in \widehat{\mathsf{C}}(e_1)$, we have to show that $(\widehat{\mathsf{C}}(e_2) \subseteq \widehat{\rho}(x))$: in other words, that $\mathsf{retn}\ulcorner v_2 \urcorner \ulcorner e_2 \urcorner \in \Delta$ implies $\mathsf{bind}\,(\mathsf{var}(\lambda x.\ulcorner e_0 \urcorner))\ulcorner v_2 \urcorner \in \Delta$. Because we know by the reasoning above that $\mathsf{cont}\,(\mathsf{app2}\,(\lambda x.\ulcorner e_0 \urcorner))\ulcorner e_2 \urcorner \ulcorner e_1\, e_2 \urcorner \in \Delta$, this follows by rule ev/app2.

The same reasoning from ev/app2 allows us to conclude that both $(\lambda x.e_0) \in \widehat{\mathsf{C}}(e_1)$ and $\mathsf{retn}\ulcorner v_2 \urcorner \ulcorner e_2 \urcorner \in \Delta$ together imply $\mathsf{eval}\ulcorner e_0 \urcorner \in \Delta$ (and therefore that $(\widehat{\mathsf{C}}, \widehat{\rho}) \models e_0$ by the coinduction hypothesis, the fourth thing we needed to prove) in addition to implying $\mathsf{cont}\,\mathsf{app3}\ulcorner e_0 \urcorner \ulcorner e_1\, e_2 \urcorner \in \Delta$ (which with ev/app3 implies $\widehat{\mathsf{C}}(e_0) \subseteq \widehat{\mathsf{C}}(e_1\, e_2)$, the last thing we needed to prove).

This completes the proof. □

We claim that, if we had started with an analysis that incorporated both parallel evaluation of functions and arguments (in the style of Figure 7.6 from Section 7.2.1) and the call-by-future functions discussed in Figure 7.9 from Section 7.2.3, then the derived analysis would have satisfied a faithful representation of Nielson et al.'s acceptability relation. The proof, in this case, should proceed along the same lines as the proof of Theorem 8.5.

8.5 Alias analysis

The control flow analysis above was derived from the SSOS specification of a language that looked much like the Mini-ML-like languages considered in Chapters 6 and 7, and we described how to justify such an analysis in terms of coinductive specifications of what comprises a well-designed control flow analysis.

In this section, we work in the other direction: the starting point for this specification was the interprocedural object-oriented alias analysis presented as a saturating logic program in [ALSU07, Chapter 12.4]. We then worked backwards to get a SSOS semantics that allowed us to derive Aho et al.'s logic program as closely as possible. The result is a monadic SSOS semantics. There should not be any obstacle to deriving an alias analysis from a semantics that looks more like the specifications elsewhere in this dissertation.

8.5.1 Monadic language

The language we consider differentiates atomic actions, which we will call *expressions* (and encode in the LF type exp) and procedures or *commands* (which we encode in the LF type cmd). There are only two commands m in our monadic language. The first command, $\mathsf{ret}\,x$, is a command that returns the value bound to the variable x (rule ev/ret in Figure 8.7). The second command, $\ulcorner \mathsf{bnd}^l\, x \leftarrow e\,\mathsf{in}\,m \urcorner = \mathsf{bnd}\, l \ulcorner e \urcorner\, \lambda x.\ulcorner m \urcorner$, evaluates e to a value, binds that value to the variable x, and then evaluates m. Note the presence of l in the bind syntax; we will call it a *label*, and we can think of it as a line number or source-code position from the original program.

In the previous languages we have considered, values v were a syntactic refinement of the expressions e. In contrast, our monadic language will differentiate the two: there are five expression forms and three values that we will consider. An expression $\ulcorner \lambda x.m_0 \urcorner = \mathsf{fun}\,\lambda x.\ulcorner m_0 \urcorner$

```
bind: variable -> value -> prop pers.
eval: cmd -> dest -> prop lin.
retn: value -> dest -> prop lin.
cont: frame -> dest -> dest -> prop lin.

ev/ret:   eval (ret X) D * !bind X V >-> {retn V D}.

ev/fun:   eval (bnd L (fun \x. M0 x) (\x. M x)) D
           >-> {Exists y. eval (M y) D * !bind y (lam L \x. M0 x)}.

ev/call:  eval (bnd L (call F X) (\x. M x)) D *
          !bind F (lam L0 (\x. M0 x)) *
          !bind X V
           >-> {Exists d0. Exists y.
                eval (M0 y) d0 * cont (call1 L (\x. M x)) d0 D *
                !bind y V}.

ev/call1: retn V D0 * cont (call1 L (\x. M x)) D0 D
           >-> {Exists y. eval (M y) D * !bind y V}.
```

Figure 8.7: Semantics of functions in the simple monadic language

evaluates to a value $\ulcorner \lambda^l x.m_0 \urcorner = \mathsf{lam}\, l\, \lambda x.\ulcorner m_0 \urcorner$, where the label l represents the source code position where the function was bound. (A function value is a command m_0 with one free variable.) When we evaluate the command $\ulcorner \mathsf{bnd}^l\, y \leftarrow \lambda x.m_0\, \mathsf{in}\, m \urcorner$, the value $\ulcorner \lambda^l x.m_0 \urcorner$ gets bound to y in the body of the command m (rule ev/fun in Figure 8.7).

The second expression form is a function call: $\ulcorner f\, x \urcorner = \mathsf{app}\, f\, x$. To evaluate a function call, we expect a function value to be bound to the variable f; we then store the rest of the current command on the stack and evaluate the command m_0 to a value. Note that the rule ev/call in Figure 8.7 also stores the call site's source-code location l on the stack frame. The reason for storing a label here is that we need it for the alias analysis. However, it is possible to independently motivate adding these source-code positions to the operational semantics: for instance, it would allow us to model the process of giving a stack trace when an exception is raised. When the function we have called returns (rule ev/call1 in Figure 8.7), we continue evaluating the command that was stored on the control stack.

The rules for mutable pairs are given in Figure 8.8. Evaluating the expression newpair allocates a tuple with two fields fst and snd and yields a value $\mathsf{loc}\, l$ referring to the tuple; both fields in the tuple are initialized to the value null, and each field is represented by a separate linear cell resource (rule ev/new). The expressions $\ulcorner x.\mathsf{fst} \urcorner = \mathsf{proj}\, x\, \mathsf{fst}$ and $\ulcorner x.\mathsf{snd} \urcorner = \mathsf{proj}\, x\, \mathsf{snd}$ expect a pair location to be bound to x, and yield the value stored in the appropriate field of the mutable pair (rule ev/proj). The expressions $\ulcorner x.\mathsf{fst} := y \urcorner = \mathsf{set}\, x\, \mathsf{fst}\, y$ and $\ulcorner x.\mathsf{snd} := y \urcorner = \mathsf{set}\, x\, \mathsf{snd}\, y$ work much the same way. The difference is that the former expressions do not change the accessed field's contents, whereas the latter expressions replace the accessed field's contents with the value bound to y (rule ev/set).

This language specification bears some similarity to Harper's Modernized Algol with free

```
cell: locvar -> field -> value -> prop lin.

ev/new:   eval (bnd L newpair (\x. M x)) D
           >-> {Exists y. Exists l'. eval (M y) D *
                cell l' fst null * cell l' snd null *
                !bind y (loc l')}.

ev/proj:  eval (bnd L (proj X Fld) (\x. M x)) D *
          !bind X (loc L') *
          cell L' Fld V
           >-> {Exists y. eval (M y) D * cell L' Fld V *
                !bind y V}.

ev/set:   eval (bnd L (set X Fld Y) (\x. M x)) D *
          !bind X (loc L') *
          !bind Y V *
          cell L' Fld V'
           >-> {Exists y. eval (M y) D *
                cell L' Fld V *
                !bind y V'}.
```

Figure 8.8: Semantics of mutable pairs in the simple monadic language

assignables [Har12, Chapter 36]. The *free assignables* addendum is critical: SSOS specifications do not have a mechanism for enforcing the stack discipline of Algol-like languages.[3]

8.5.2 Approximation and alias analysis

To approximate the semantics of our monadic language, we can follow the methodology from before and turn the specification persistent. A further approximation is to remove the last premise from ev/set, as the meta-approximation theorem allows – the only purpose of this premise in Figure 8.8 was to consume the ephemeral proposition cell l' fld v, and this is unnecessary if cell is not an ephemeral predicate. Having made these two moves (turning all propositions persistent, and removing a premise from ev/set), we are left with three types of existentially-generated variables that must be equated with concrete terms in order for our semantics to be interpreted as a saturating logic program:

- Variables y, introduced by every rule except for ev/ret,
- Mutable locations l, introduced by rule ev/new, and
- Destinations d; the only place where a destination is created by the destination-adding transformation is in rule ev/call.

[3]It is, however, possible to represent Algol-like languages that maintain a stack discipline even though the machinery of SLS does not enforce that stack discipline. This is analogous to the situation with pointer equality discussed in Section 6.5.1, as a stack discipline is an invariant that can be maintained in SLS even though the framework's proof theory does not enforce the invariant.

```
bind: label -> value -> prop pers.
eval: cmd -> label -> prop pers.
retn: value -> label -> prop pers.
cont: frame -> label -> label -> prop pers.
cell: label -> field -> value -> prop pers.

ev/ret:   eval (ret X) D * bind X V >-> {retn V D}.

ev/fun:   eval (bnd L (fun \x. M0 x) (\x. M x)) D
           >-> {eval (M L) D * bind L (lam L \x. M0 x)}.

ev/call:  eval (bnd L (call F X) (\x. M x)) D *
          bind F (lam L0 \x. M0 x) *
          bind X V
           >-> {eval (M0 L0) L0 * cont (call1 L (\x. M x)) L0 D}.

ev/call1: retn V D0 * cont (call1 L (\x. M x)) D0 D
           >-> {eval (M L) D * bind L V}.

ev/new:   eval (bnd L newpair (\x. M x)) D
           >-> {eval (M L) D *
                cell L fst null * cell L snd null *
                bind L (loc L)}.

ev/proj:  eval (bnd L (proj X Fld) (\x. M x)) D *
          bind X (loc L') *
          cell L' Fld V
           >-> {eval (M L) D * cell L' Fld V *
                bind L V}.

ev/set:   eval (bnd L (set X Fld Y) (\x. M x)) D *
          bind X (loc L') *
          bind Y V
           >-> {eval (M L) D * cell L' Fld V *
                bind L V'}.
```

Figure 8.9: Alias analysis for the simple monadic language

Variables y are generated to be substituted into the body of some command, so we could equate them with the Skolemized function body as we did when deriving a control flow analysis example. Another option comes from noting that, for any initial source program, every command is associated with a particular source code location, so a simpler alternative is just to equate the variable with that source code location. This is why we stored labels on the stack: if we had not done so, then the label l associated with m in the command $\ulcorner \mathsf{bnd}^l\, x \leftarrow \lambda x.m_0 \,\mathsf{in}\, m \urcorner$ would no longer be available when we needed it in rule ev/call1.

We deal with mutable locations l in a similar manner: we equate them with the label l representing the line where that cell was generated.

There are multiple ways to deal with the destination d_0 generated in rule ev/call. We want our analysis, like Aho et al.'s, to be insensitive to control flow, so we will equate d_0 with the label l_0 associated with the function we are calling. If we instead equated d_0 with the label l associated with the call-site or with the pair of the call site and the called function, the result would be an analysis that is more sensitive to control flow.

The choices described above are reflected in Figure 8.9, which takes the additional step of inlining uses of equality in the conclusions of rules. We can invoke this specification as a program analysis by packaging a program as a single command m and deriving a saturated process state from the initial process state $(l_{init}{:}\mathsf{loc}; x{:}\langle \mathsf{eval} \ulcorner m \urcorner\, l_{init}\rangle)$. The use of source-code position labels makes the answers to some of the primary questions asked of an alias analysis quite concise. For instance:

* *Might the first component of a pair created at label l_1 ever reference a pair created at label l_2?* Only if $\mathsf{cell}\, l_1\, \mathsf{fst}\, (\mathsf{loc}\, l_2)$ appears in the saturated process state (and likewise for the second component).
* *Might the first component of a pair created at label l_1 ever reference the same object as the first component of a pair created at label l_2?* Only if there is some l' such that $\mathsf{cell}\, l_1\, \mathsf{fst}\, (\mathsf{loc}\, l')$ and $\mathsf{cell}\, l_1\, \mathsf{fst}\, (\mathsf{loc}\, l')$ both appear in the saturated process state.

8.6 Related work

The technical aspects of linear logical approximation are similar to work done by Bozzano et al. [BDM02, BDM04], which was also based on the abstract interpretation of a logical specification in linear logic. They encode distributed systems and communication protocols in a framework that is similar to the linear fragment of SLS without equality. Abstractions of those programs are then used to verify properties of concurrent protocols that were encoded in the logic [BD02].

There are a number of significant difference between our work and Bozzano et al.'s, however. The style they use to encode protocols is significantly different from any of the SSOS specification styles presented in this dissertation. They used a general purpose approximation, which could therefore potentially be mechanized in the same way we mechanized transformations like operationalization; in contrast, the meta-approximation result described here captures a whole class of approximations. Furthermore, Bozzano et al.'s methods are designed to consider properties of a system as a whole, not static analyses of individual inputs as is the case in our

work.

Work by Might and Van Horn on abstracting abstract machines can be seen as a parallel approach to our methodology in a very different setting [MSV10, Mig10, VM10]. Their emphasis is on deriving a program approximation by approximating a *functional* abstract interpreter for a programming language's operational semantics. Their methodology is similar to ours in large part because we are doing the same thing in a different setting, deriving a program approximation by approximating a destination-passing SSOS specification (which we could, in turn, have derived from an ordered abstract machine by destination-adding).

Many of the steps that they suggest for approximating programs have close analogues in our setting. For instance, their *store-allocated bindings* are analogous to the SSOS environment semantics, and their *store-allocated continuations* – which they motivate by analogy to implementation techniques for functional languages like SML/NJ – are precisely the destinations that arise naturally from the destination-adding transformation. The first approximation step we take is forgetting about linearity in order to obtain a (non-terminating) persistent logical specification. This step is comparable to Might's first approximation step of "throwing hats on everything" (named after the convention in abstract interpretation of denoting the abstract version of a state space Σ as $\hat{\Sigma}$. The "mysterious" introduction of power domains that this entails is, in our setting, a perfectly natural result of relaxing the requirement that there be at most one persistent proposition $\mathsf{bind}\, x\, v$ for every x. As a final point of comparison, the "abstract allocation strategy" discussed in [VM10] is quite similar to our strategy of introducing and then approximating Skolem functions as a means of deriving a finite approximation. Our current discussion of Skolem functions in Section 8.4 is partially inspired by the relationship between our use of Skolemization and the discussion of abstract allocation in [VM10].

The independent discovery of a similar set of techniques for achieving similar goals in such different settings (though both approaches were to some degree inspired by Van Horn and Mairson's investigations of the complexity of k-CFA [VM07]) is another indication of the generality of both techniques, and the similarity also suggests that the wide variety of approximations considered in [VM10], as well as the approximations of object-oriented programming languages in [Mig10], can be adapted to this setting.

Part III

Reasoning about substructural logical specifications

Chapter 9

Generative invariants

So far, we have presented SLS as a framework for presenting transition systems. This view focuses on synthetic transitions as a way of relating pairs of process states, either with one transition $(\Psi; \Delta) \rightsquigarrow (\Psi'; \Delta')$ or with a series of transitions $(\Psi; \Delta) \rightsquigarrow^* (\Psi'; \Delta')$. This chapter will focus on another view of concurrent SLS specifications as *grammars* for describing well-formed process states. This view was presented previously in the discussions of adequacy in Section 4.4.1 and in Section 6.3.

The grammar-like specifications that describe well-formed process states are called *generative signatures*, and generative signatures can be used to specify sets of process states, or *worlds*. By the analogy with grammars, we could also describe worlds as *languages* of process states recognized by the grammar. In our previous discussions of adequacy in Section 4.4.1 and in Section 6.3, the relevant world was a set of process states that we could put in bijective correspondence with the states of an abstract machine.

Generative signatures are a significant extension of context-free grammars, both because of the presence of dependent types and because of the presence of linear and persistent resources in SLS. However, we will not endeavor to study generative signatures in their own right in this chapter or this dissertation. Rather, we will use generative signatures for one very specific purpose: showing that, under some generative signature Σ_{Gen} that defines a world $\mathcal{W}$, whenever $(\Psi; \Delta) \in \mathcal{W}$ and $(\Psi; \Delta) \rightsquigarrow_\Sigma (\Psi'; \Delta')$ it is always the case that $(\Psi'; \Delta') \in \mathcal{W}$. (The signature Σ encodes the transition system we are studying.) In such a case, a world or language of well-formed process states is called a *generative invariant* of Σ.

Type preservation

Narrowing our focus even further, in this chapter our sole use of generative invariants will be describing well-formedness and well-typedness invariants of the sorts of substructural operational semantics specifications presented in Part II. When we want to prove language safety for a small-step SOS specification like $e \mapsto e'$ from Section 6.6.2 and the beginning of Chapter 6, we define a judgment $x_1{:}tp_1, \ldots, x_n{:}tp_n \vdash e : tp$. This *typing judgment* expresses that e has type tp if the expression variables $x_1, \ldots, x_n$ are respectively assumed to have the types $tp_1, \ldots, tp_n$. (Note that tp is an *object-level type* as described in Section 9.3, not an LF type τ from Chapter 4.)

Well-typedness invariants are important because they allow us to prove *language safety*, the

property (discussed way back in the introduction) that a language specification is completely free from undefined behavior. The standard "safety = progress + preservation" formulation of type safety is primarily a statement about invariants. We specify some property ("well-typed with type tp"), show that it is invariant under execution (preservation: "if $e \mapsto e'$ and e has type tp, then e' has type tp"), and show that any state with that property has well-defined behavior (progress: "if e has type tp, it steps or is a value").

The purpose of this chapter is to demonstrate that generative invariants are a solid methodology for describing invariants of SLS specifications, especially well-formedness and -typedness invariants of substructural operational semantics specifications like the ones presented in Part II. As we have already seen, well-formedness invariants are major part of adequacy theorems. In the next chapter, we will show that well-typedness invariants are sufficient for proving progress theorems, meaning that generative invariants can form the basis of progress-and-preservation-style safety theorems for programming languages specified in SLS. These two chapters support the third refinement of our central thesis:

> **Thesis (Part III):** *The SLS specification of the operational semantics of a programming language is a suitable basis for formal reasoning about properties of the specified language.*

Overview

In Section 9.1 we review how generative signatures define a world and show how the *regular worlds* that Schürmann implemented in Twelf [Sch00] fall out as a special case of the worlds described by generative signatures. After this, the core of this chapter plays the same game – describing a well-formedness or well-typedness property with a generative signature and proving that the property is a generative invariant – five times. In each step, we motivate and explain new concepts.

* In Section 9.2 we extend the well-formedness invariant for sequential ordered abstract machines described in Section 6.3 to parallel ordered abstract machines with failure, setting up the basic pattern.
* In Section 9.3 we switch from specifying well-*formed* process states to specifying well-*typed* process states. This is not a large technical shift, but conceptually it is an important step from thinking about adequacy properties to thinking about preservation theorems.
* In Section 9.4 we describe how generative invariants can be established for the sorts of stateful signatures considered in Section 6.5. This specification introduces the *promise-then-fulfill* pattern and also requires us to consider *unique index* properties of specifications (Section 9.4.2).
* In Section 9.5 we consider invariants for specifications in the image of the destination-adding transformation from Chapter 7. This formalization, which is in essence a SLS encoding of Cervesato and Sans's type system from [CS13], also motivates the introduction of *unique index sets* to state unique index properties more concisely.
* In Section 9.6 we consider the peculiar case of first-class continuations, which require us to use persistent continuation frames as described in Section 7.2.4. Despite the superficial

similarities between the SSOS semantics for first-class continuations and the other SSOS semantics considered in this dissertation, first-class continuations fundamentally change the control structure, and this is reflected in a fundamental change to the necessary generative invariants.

We conclude in Section 9.7 with a brief discussion of the mechanization of generative invariants, though this is primarily left for future work. In general, this chapter aims to be the first word in the use of generative invariants, but it is by no means the last.

9.1 Worlds

Worlds are nothing more or less than sets of stable process states $(\Psi; \Delta)$ as summarized in Appendix A. In this chapter, we will specify worlds with the combination of two artifacts: an initial process state and a generative signature.

Definition 9.1. *A* generative signature *is a SLS signature where the ordered, mobile, and persistent atomic propositions can be separated into two sets – the* terminals *and the* nonterminals. *Synthetic transitions enabled by a generative signature only consume (or reference) nonterminals and LF terms, but their output variables can include LF variables, variables associated with terminals, and variables associated with nonterminals.*

The use of terminal/nonterminal terminology favors the view of generative signatures as context-free grammars, an analogy that holds well for ordered nonterminals. Mobile nonterminals behave more like obligations when we use them as part of the promise-then-fulfill pattern (Section 9.4 and beyond), and persistent nonterminals behave more like constraints.

A generative signature, together with an initial state $(\Psi_0; \Delta_0)$, describes a world with the help of the restriction operator $(\Psi; \Delta)\lightning_\Sigma$ introduced in Section 4.4.2. To recap, if $(\Psi; \Delta)$ is well-defined under the generative signature Σ_{Gen}, and Σ is any signature that includes all of the generative signature's terminals and all of its LF declarations but none of its nonterminals, then $(\Psi; \Delta)\lightning_\Sigma$ is only defined when the only remaining nonterminals in Δ are persistent and can therefore be filtered out of Δ. When the classification of terminals and nonterminals is clear, we will leave off the restricting signature and just write $(\Psi; \Delta)\lightning$.

As a concrete example, let $\mathsf{nt/foo}$ be a persistent nonterminal, let $\mathsf{nt/bar}$ be an ordered nonterminal, and let $\mathsf{t/baz}$ be an ordered terminal. Then $(x{:}\langle\mathsf{nt/bar}\rangle\ ord, y{:}\langle\mathsf{t/baz}\rangle\ ord)\lightning$ is not defined, $(y{:}\langle\mathsf{t/baz}\rangle\ ord)\lightning = (y{:}\langle\mathsf{t/baz}\rangle\ ord)$, and $(x{:}\langle\mathsf{nt/foo}\rangle\ pers, y{:}\langle\mathsf{t/baz}\rangle\ ord)\lightning = (y{:}\langle\mathsf{t/baz}\rangle\ ord)$. Recalling the two-dimensional notation from Chapter 4, we can re-present these three statements as follows:

$$\begin{array}{ccc}
(x{:}\langle\mathsf{nt/bar}\rangle\ ord, y{:}\langle\mathsf{t/baz}\rangle\ ord) & (y{:}\langle\mathsf{t/baz}\rangle\ ord) & (x{:}\langle\mathsf{nt/foo}\rangle\ pers, y{:}\langle\mathsf{t/baz}\rangle\ ord) \\
\text{////////////////////////////} & \text{//////////////} & \text{//////////////////////////////} \\
 & (y{:}\langle\mathsf{t/baz}\rangle\ ord) & (y{:}\langle\mathsf{t/baz}\rangle\ ord)
\end{array}$$

Definition 9.1 is intentionally quite broad – it need not even be decidable whether a process state belongs to a particular world.[1] Future tractable analyses will therefore presumably be

[1]Proof: consider the initial state $(x{:}\langle\mathsf{gen}\rangle\ ord)$ and the rule $\forall e.\, \forall v.\, \mathsf{gen} \bullet !(\mathsf{ev}\, e\, v) \rightarrowtail \{\mathsf{terminating}\, e\}$. The

based upon further restrictions of the very general Definition 9.1. Context-free grammars are one obvious specialization of generative signatures; we used this correspondence as an intuitive guide in Section 4.4.1. Perhaps less obviously, the regular worlds of Twelf [Sch00] are another specialization of generative signatures.

9.1.1 Regular worlds

The regular worlds used in Twelf [Sch00] are specified with sets of *blocks*. A block describes a little piece of an LF context, and is declared in the LF signature as follows:

$$\mathsf{blockname} : \mathsf{some}\ \{a_1{:}\tau_1\} \ldots \{a_n{:}\tau_n\}\ \mathsf{block}\ \{b_1{:}\tau'_1\} \ldots \{b_m{:}\tau'_m\}$$

A block declaration is well formed in the signature Σ if, by the definition of well-formed signatures from Figure 4.3, $\cdot \vdash_\Sigma a_1{:}\tau_1, \ldots, a_n{:}\tau_n\ \mathsf{ctx}$ and $a_1{:}\tau_1, \ldots, a_{i-1}{:}\tau_n \vdash_\Sigma b_1{:}\tau'_1, \ldots, b_m{:}\tau'_m\ \mathsf{ctx}$.

The first list of LF variable bindings $\{a_1{:}\tau_1\} \ldots \{a_n{:}\tau_n\}$ that come after the some keyword describe the types of concrete LF terms that must exist for the block to be well formed. The second list of LF variable bindings represents the bindings that the block actually adds to the LF context. The regular worlds of Twelf are specified with sets of block identifiers (block1 | ... | blockn). A set $\mathcal{S}$ of block identifiers and a Twelf signature Σ inductively define a world as follows: the empty context belongs to every regular world, and if

* Ψ is a well-formed LF context in the current world,
* $\mathsf{blockname} : \mathsf{some}\ \{a_1{:}\tau_1\} \ldots \{a_n{:}\tau_n\}\ \mathsf{block}\ \{b_1{:}\tau'_1\} \ldots \{b_m{:}\tau'_m\} \in \Sigma$ is one of the blocks in $\mathcal{S}$, and
* there is a σ such that $\Psi \vdash_\Sigma \sigma : a_1{:}\tau_1, \ldots, a_n{:}\tau_n$,

then $\Psi, b_1{:}\sigma\tau'_1, \ldots, b_m{:}\sigma\tau'_m$ is also a well-formed LF context in the current world. The *closed world*, which contains only the empty context, is specified by the empty set of block identifiers.

One simple example of a regular world (previously discussed in Section 4.4.1) is one that contains all contexts with just expression variables of LF type exp. This world can be described with the block blockexp:

$$\mathsf{blockexp} : \mathsf{some}\ \mathsf{block}\ \{x{:}\mathsf{exp}\}$$

If we had a judgment $\mathsf{natvar}\ x\ n$ that associated every LF variable x:exp with some natural number n:nat, then in order to make sure that every expression variable was associated with some natural number we would use the world described by this block:

$$\mathsf{blocknatvar} : \mathsf{some}\ \{n{:}\mathsf{nat}\}\ \mathsf{block}\ \{x{:}\mathsf{exp}\}\ \{nv{:}\mathsf{natvar}\ x\ n\}$$

The world described by the combination of blockexp and blocknatvar is one where every LF variable x:exp is associated with at most one LF variable of type $\mathsf{natvar}\ x\ n$. Assuming that there are no constants of type natvar,[2] this gives us a uniqueness property: if $\mathsf{natvar}\ x\ n$ and $\mathsf{natvar}\ x\ m$, then $m = n$.

predicate gen is a nonterminal, the predicate terminating is a terminal, and ev is the encoding of big-step evaluation $e \Downarrow v$ from Figure 6.1. The language described is isomorphic to the set of λ-calculus expressions that terminate under a call-by-value strategy, and membership in that set is undecidable.

[2]This is a property we can easily enforce with subordination, which was introduced in Section 4.1.3.

9.1.2 Regular worlds from generative signatures

A block declaration $\mathsf{blockname} : \mathsf{some}\ \{a_1{:}\tau_1\} \ldots \{a_n{:}\tau_n\}\ \mathsf{block}\ \{b_1{:}\tau'_1\} \ldots \{b_m{:}\tau'_m\}$ can be described by one rule in a generative signature:

$$\mathsf{blockname} : \forall a_1{:}\tau_1 \ldots \forall a_n{:}\tau_n.\ \{\exists b_1{:}\tau'_1 \ldots b_m{:}\tau'_m.\ \mathbf{1}\}$$

Because a regular world is just a set of blocks, the generative signature corresponding to a regular world contains one rule for each block in the regular worlds description. The world described by $(\mathsf{blockexp} \mid \mathsf{blockvar})$ corresponds to the following generative signature:

$$\begin{aligned}
&\mathsf{nat} : \mathsf{type},\\
&\ldots \textit{declare constants of type}\ \mathsf{nat} \ldots\\
&\mathsf{exp} : \mathsf{type},\\
&\ldots \textit{declare constants of type}\ \mathsf{exp} \ldots\\
&\mathsf{blockexp} : \{\exists x{:}\mathsf{exp}.\ \mathbf{1}\},\\
&\mathsf{blocknatvar} : \forall n{:}\mathsf{nat}.\ \{\exists x{:}\mathsf{exp}.\ \exists nv{:}\mathsf{natvar}\ x\ n.\ \mathbf{1}\}
\end{aligned}$$

Call this regular world signature Σ_{RW}. It is an extremely simple example of a generative signature – there are no terminals and no nonterminals – so the restriction operator has no effect. The world described by $(\mathsf{blockexp} \mid \mathsf{blocknatvar})$ is identical to the set of LF contexts Ψ such that $(\cdot;\cdot) \rightsquigarrow_{\Sigma_{RW}} (\Psi;\cdot)$.

9.1.3 Regular worlds in substructural specifications

It is a simple generalization to replace the proposition $\mathbf{1}$ in the head of the generative $\mathsf{block}*$ rules above with less trivial positive SLS propositions. In this way, we can extend the language of regular worlds to allow the introduction of ordered, mobile, and persistent SLS propositions as well. For instance, the rule $\mathsf{blockitem} : \forall n.\ \{\mathsf{item}\ n\}$, where item is a mobile predicate, describes the world of contexts that take the form $(\cdot;\ x_1{:}\langle \mathsf{item}\ n_1\rangle\ eph,\ \ldots,\ x_k{:}\langle \mathsf{item}\ n_k\rangle\ eph)$ for some numbers $n_1 \ldots n_k$. The world described by this generative signature is an invariant of a rule like

$$\mathsf{merge} : \forall n.\ \forall m.\ \forall p.\ \mathsf{item}\ n \bullet \mathsf{item}\ m \bullet !(\mathsf{plus}\ n\ m\ p) \rightarrowtail \{\mathsf{item}\ p\}$$

that combines two items, where plus is negative predicate defined with a deductive specification as in Figure 6.21.

Such substructural generalizations of regular worlds are sufficient for the encoding of stores in Linear LF [CP02] and stacks in Ordered LF [Pol01]. They also suffice to describe well-formedness invariants in Felty and Momigliano's sequential specifications [FM12]. However, regular worlds are insufficient for the invariants discussed in the remainder of this chapter.

9.1.4 Generative versus consumptive signatures

Through the example of regular worlds, we can explain why worlds are defined as sets of process states generated by a signature Σ_{Gen} and an initial state $(\Psi;\Delta)$:

$$\{(\Psi';\Delta'') \mid (\Psi;\Delta) \rightsquigarrow^*_{\Sigma_{Gen}} (\Psi';\Delta') \wedge (\Psi';\Delta')\lightning = (\Psi';\Delta'')\}$$

as opposed to the apparently symmetric case where worlds are sets of process states that *can generate* a final process state $(\Psi; \Delta)$ under a signature $\Sigma Cons$, which we will call a *consumptive* signature:

$$\{(\Psi'; \Delta'') \mid (\Psi'; \Delta') \rightsquigarrow^*_{\Sigma_{Gen}} (\Psi; \Delta) \wedge (\Psi'; \Delta')\lightning = (\Psi'; \Delta'')\}$$

Consumptive signatures look like generative signatures with the arrows turned around: we consume well-formed contexts using rules like $\forall e.\, \mathsf{eval}\, e \rightarrowtail \{\mathsf{safe}\}$ and $\forall f.\, \mathsf{safe} \bullet \mathsf{cont}\, f \rightarrowtail \{\mathsf{safe}\}$ instead of creating them with rules like $\forall e.\, \mathsf{gen} \rightarrowtail \{\mathsf{eval}\, e\}$ and $\forall f.\, \mathsf{gen} \rightarrowtail \{\mathsf{gen} \bullet \mathsf{cont}\, f\}$. One tempting property of consumptive signatures is that they open up the possibility of working with complete derivations rather than traces. That is, using a consumptive signature, we can talk about the set of process states $(\Psi; \Delta)$ where $\Psi; \Delta \vdash \mathsf{safe}\, lax$ rather than the set of process states where $(\cdot; x{:}\langle \mathsf{gen} \rangle\, ord) \rightsquigarrow^* (\Psi; \Delta)$.[3]

For purely context-free-grammar-like invariants, such as the PDA invariant from Section 4.4.1 and the SSOS invariant from Section 6.3, generative and consumptive signatures are effectively equivalent. However, for generative signatures describing regular worlds, there is no notion of turning the arrows around to get an appropriate consumptive signature. In particular, say we want to treat

$$\Psi_{good} = (x_1{:}\mathsf{exp}, nv_1{:}\mathsf{natvar}\, x_1\, n_1, x_2{:}\mathsf{exp}, nv_2{:}\mathsf{natvar}\, x_2\, n_2)$$

as a well-formed LF context but *not* treat

$$\Psi_{bad} = (x{:}\mathsf{exp}, nv_1{:}\mathsf{natvar}\, x\, n_1, nv_2{:}\mathsf{natvar}\, x\, n_2)$$

as well-formed. It is trivial to use Twelf's regular worlds or generative signatures to impose this condition, but it does not seem possible to use consumptive signatures for this purpose. There exists a substitution $(x /\!\!/ x_1, nv_1 /\!\!/ nv_1, x /\!\!/ x_2, nv_2 /\!\!/ nv_2)$ from Ψ_{good} to Ψ_{bad}; therefore, by variable substitution (Theorem 3.4), if there exists a derivation of $\Psi_{good} \vdash_\Sigma \mathsf{gen}\, lax$ there also exists a derivation of $\Psi_{bad} \vdash_\Sigma \mathsf{gen}\, lax$. This is related to the issues of variable and pointer (in)equality discussed in Section 6.5.1.

The generative signatures used to describe state in Section 9.4 and destination-passing style in Section 9.5 rely critically on the uniqueness properties that are provided by generative signatures and not by consumptive signatures.

9.2 Invariants of ordered specifications

We already introduced generative invariants for ordered abstract machine SSOS specifications in Section 6.3. In this section, we will extend that generative invariant to ordered abstract machines with parallel evaluation and recoverable failure.

In Figure 9.1 we define a flat ordered abstract machine with parallel features (parallel evaluation of the function and argument in an application, as discussed in Section 6.1.4 and Figure 6.3) and recoverable failure (as presented in Section 6.5.4 and Figure 6.20). To make sure there

[3] As long as Ψ and Δ contain only nonterminals – using consumptive signatures doesn't obviate the need for the restriction operation $(\Psi; \Delta)\lightning$ or some equivalent restriction operation.

```
eval: exp -> prop ord.
retn: exp -> prop ord.
cont: frame -> prop ord.
cont2: frame -> prop ord.
error: prop ord.
handle: exp -> prop ord.

;; Unit
ev/unit: eval unit >-> {retn unit}.

;; Sequential let
ev/let:  eval (let E \x. E' x) >-> {eval E * cont (let1 \x. E' x)}.
ev/let1: retn V * cont (let1 \x. E' x) >-> {eval (E' V)}.

;; Functions and parallel application
ev/lam:  eval (lam \x. E x) >-> {retn (lam \x. E x)}.
ev/app:  eval (app E1 E2) >-> {eval E1 * eval E2 * cont2 app1}.
ev/app1: retn (lam \x. E x) * retn V2 * cont2 app1
          >-> {eval (E V2)}.

;; Recoverable failure
ev/fail:   eval fail >-> {error}.
ev/catch:  eval (catch E1 E2) >-> {eval E1 * handle E2}.
ev/catcha: retn V * handle _ >-> {retn V}.
ev/catchb: error * handle E2 >-> {eval E2}.

ev/error:  error * cont _ >-> {error}.
ev/errerr: error * error * cont2 _ >-> {error}.
ev/errret: error * retn _ * cont2 _ >-> {error}.
ev/reterr: retn _ * error * cont2 _ >-> {error}.
```

Figure 9.1: Ordered abstract machine with parallel evaluation and failure

is still an interesting sequential feature, we also introduce a let-expression $\ulcorner \mathsf{let}\, x = e \,\mathsf{in}\, e' \urcorner = \mathsf{let} \ulcorner e \urcorner \lambda x. \ulcorner e' \urcorner$. The particular features are less important than the general setup, which effectively represents all the specifications from Chapter 6 that used only ordered atomic propositions.

Our goal is to describe a generative signature that represents the well-formed process states of the specification in Figure 9.1. What determines whether a process state is well formed? The intended adequacy theorem was our guide in Section 6.3, and the intended progress theorem will guide our hand in Section 9.3. An obvious minimal requirement is that every state Δ such that $(x{:}\langle \mathsf{eval} \rangle \ulcorner e \urcorner\, ord) \leadsto^* \Delta$ under the signature from Figure 9.1 must be well formed; otherwise well-formedness won't be invariant under evaluation! One option is therefore to make this correspondence precise, and to have the well formed states be *precisely* the states that are reachable in the process of evaluating syntactically valid expressions $\ulcorner e \urcorner$. That is, if $(x{:}\langle \mathsf{gen} \rangle\, ord) \leadsto^* \Delta$ under the generative signature and if Δ contains no instances of gen, then there should be an ex-

pression e such that $(x{:}\langle \mathsf{eval}\ulcorner e\urcorner\rangle\ ord) \rightsquigarrow^* \Delta$ under the signature from Figure 9.1. (Because gen is the only nonterminal, we can express that Δ contains no instances of gen with the restriction operator, writing $\Delta\lightning$.)[4]

The analogues of the unary grammar productions, associated with the terminals eval e, retn v, and error, are straightforward:

```
gen/eval:   gen >-> {eval E}.
gen/retn:   gen * !value V >-> {retn V}.
gen/error:  gen >-> {error}.
```

As in Section 6.3, we use a deductively-defined judgment value v to stipulate that we only return values. The process state $(y{:}\langle \mathsf{retn}\ulcorner e_1\ e_2\urcorner\rangle\ ord)$ is not well formed: the application expression $e_1\ e_2$ is not a value, and there is no e such that $(x{:}\langle \mathsf{eval}\ulcorner e\urcorner\rangle\ ord) \rightsquigarrow^* (y{:}\langle \mathsf{retn}\ulcorner e_1\ e_2\urcorner\rangle\ ord)$ under the signature from Figure 9.1.

There is a potential catch when we consider the rules for sequential continuations cont f and parallel continuations cont2 f. We expect a sequential continuation frame to be preceded by a single well-formed computation, and for a parallel continuation frame to be preceded by *two* well-formed computations, suggesting these rules:

```
gen/cont:  gen >-> {gen * cont F}.
gen/cont2: gen >-> {gen * gen * cont2 F}.
```

Even though gen/cont is exactly the rule for sequential continuations in Section 6.3, this approach conflicts with our guiding principle of reachability. Both parallel continuation frames cont f and sequential continuation frames cont2 f are indexed by LF terms f of type frame, but the parallel frame app1 cannot appear in a sequential continuation, nor can the sequential frame $(\mathsf{let1}\ \lambda x.e\ x)$ appear in a parallel frame.

This is fundamentally no more complicated than the restrictions we placed on the retn v terminal. All expressions (LF variables of type exp) can appear in exp e propositions (and in handle e propositions), but only some can appear in retn v frames. We describe that subset of frames with the negative atomic proposition value v, which is deductively defined. Similarly, only some frames can appear in cont f terminals, and only some frames can appear in cont2 f terminals. The former subset can be expressed by a negative atomic proposition okf f, and the latter by a negative atomic proposition okf2 f. Both of these are deductively defined. The full specification of this generative invariant is shown in Figure 9.2; we will refer to this generative signature as $\Sigma_{Gen9.2}$.

9.2.1 Inversion

Traditional inversion lemmas are a critical part of preservation properties for small-step operational semantics specifications. In traditional preservation theorems, we often start with a derivation of $e_1\ e_2 \mapsto e_1'\ e_2$ and another derivation of $\cdot \vdash e_1\ e_2 : tp$. An inversion lemma then proceeds by case analysis on the structure of the derivation $\cdot \vdash e_1\ e_2 : tp$, and allows us to conclude that $\cdot \vdash e_1 : tp' \rightharpoonup tp$ and that $\cdot \vdash e_2 : tp'$ for some object-level type tp'. In other words, an inversion

[4]We won't discuss the proof of this property, but the proof is not difficult to reconstruct; it follows the same contours as the proof of progress given in Chapter 10.

```
value: exp -> prop.
value/unit: value unit.
value/lam: value (lam \x. E x).

okf: frame -> prop.
okf/let1: okf (let1 \x. E' x).

okf2: frame -> prop.
okf2/app1: okf2 app1.

gen: prop ord.
gen/eval:   gen >-> {eval E}.
gen/retn:   gen * !value V >-> {retn V}.
gen/cont:   gen * !okf F >-> {gen * cont F}.
gen/cont2:  gen * !okf2 F >-> {gen * gen * cont2 F}.
gen/error:  gen >-> {error}.
gen/handle: gen >-> {gen * handle E2}.
```

Figure 9.2: Generative invariant: well-formed process states

lemma allows us to take knowledge about a term's structure and obtain information about the structure of typing derivation.

Inversion on a generative signature is intuitively similar: we take information about the structure of a process state and use it to learn about the generative trace that formed that process state. Concurrent equality (Section 4.3) is critical. None of the parts of the lemma below would hold if we did not equate traces such as

$$
\begin{array}{l}
\quad (x'{:}\langle \mathsf{gen} \rangle\ ord) \\
\{x_1, x_2, x_3\} \leftarrow \mathsf{gen/cont2}\ f\ (x' \bullet \mathsf{okf2/app1}) \\
\{y_1\} \leftarrow \mathsf{gen/eval}\ e_1\ x_1 \\
\{y_2\} \leftarrow \mathsf{gen/eval}\ e_2\ x_2 \\
\quad (y_1{:}\langle \mathsf{eval}\ e_1 \rangle\ ord,\ y_2{:}\langle \mathsf{eval}\ e_2 \rangle\ ord,\ x_3{:}\langle \mathsf{cont2}\ f \rangle\ ord)
\end{array}
$$

and

$$
\begin{array}{l}
\quad (x'{:}\langle \mathsf{gen} \rangle\ ord) \\
\{x_1, x_2, x_3\} \leftarrow \mathsf{gen/cont2}\ f\ (x' \bullet \mathsf{okf2/app1}) \\
\{y_2\} \leftarrow \mathsf{gen/eval}\ e_2\ x_2 \\
\{y_1\} \leftarrow \mathsf{gen/eval}\ e_1\ x_1 \\
\quad (y_1{:}\langle \mathsf{eval}\ e_1 \rangle\ ord,\ y_2{:}\langle \mathsf{eval}\ e_2 \rangle\ ord,\ x_3{:}\langle \mathsf{cont2}\ f \rangle\ ord)
\end{array}
$$

by concurrent equality.

The function of an inversion lemma is to conclude, based on the structure of a generated process state, something about the last step in the trace that generated it. This is less immediate than inversion on derivations because concurrent traces can have many steps which can all

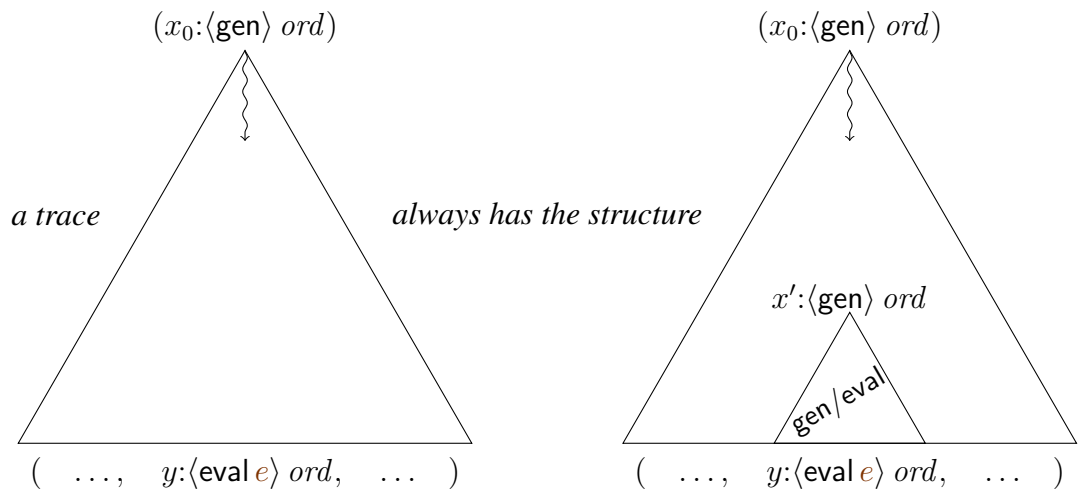


Figure 9.3: Graphical representation of part 1 of the inversion lemma for $\Sigma_{Gen9.2}$

equivalently be treated the last, such as the two gen/eval steps above. Another way of looking at the inversion lemma, which emphasizes that generative traces act like rewriting rules, is shown in Figure 9.3.

Lemma (Inversion – Figure 9.2)**.**

1. *If* $T :: (x_0{:}\langle\mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.2}} \Theta\{y{:}\langle\mathsf{eval}\ e\rangle\ ord\}$,[5]
   then $T = (T'; \{y\} \leftarrow \mathsf{gen/eval}\ e\ x')$.
2. *If* $T :: (x_0{:}\langle\mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.2}} \Theta\{y{:}\langle\mathsf{retn}\ v\rangle\ ord\}$,
   then $T = (T'; \{y\} \leftarrow \mathsf{gen/retn}\ v\ (x' \bullet !N))$,
   where $\cdot \vdash N : \mathsf{value}\ v\ true$.[6]
3. *If* $T :: (x_0{:}\langle\mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.2}} \Theta\{y_1{:}\langle\mathsf{gen}\rangle\ ord,\ y_2{:}\langle\mathsf{cont}\ f\rangle\ ord\}$,
   then $T = (T'; \{y_1, y_2\} \leftarrow \mathsf{gen/cont}\ f\ (x' \bullet !N))$,
   where $\cdot \vdash N : \mathsf{okf}\ f\ true$.
4. *If* $T :: (x_0{:}\langle\mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.2}} \Theta\{y_1{:}\langle\mathsf{gen}\rangle\ ord,\ y_2{:}\langle\mathsf{gen}\rangle\ ord,\ y_3{:}\langle\mathsf{cont2}\ f\rangle\ ord\}$,
   then $T = (T'; \{y_1, y_2, y_3\} \leftarrow \mathsf{gen/cont2}\ f\ (x' \bullet !N))$,
   where $\cdot \vdash N : \mathsf{okf2}\ f\ true$.
5. *If* $T :: (x_0{:}\langle\mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.2}} \Theta\{y{:}\langle\mathsf{error}\rangle\ ord\}$,
   then $T = (T'; \{y\} \leftarrow \mathsf{gen/error}\ x')$.
6. *If* $T :: (x_0{:}\langle\mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.2}} \Theta\{y_1{:}\langle\mathsf{gen}\rangle\ ord,\ y_2{:}\langle\mathsf{handle}\ e\rangle\ ord\}$,
   then $T = (T'; \{y_1, y_2\} \leftarrow \mathsf{gen/handle}\ e\ (x' \bullet !N))$.

In each instance above, $T' :: (x_0{:}\langle\mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.2}} \Theta\{x'{:}\langle\mathsf{gen}\rangle\ ord\}$*, where the variables* x_0 *and* x' *may or may not be the same. (They are the same iff* $T' = \diamond$*.)*

[5]Our notation for frames Θ and the tacking-on operation $\Theta\{\Delta\}$ are summarized in Appendix A.

[6]In this chapter, the signature associated with every deductive derivation ($\Sigma_{Gen9.2}$ in this case) is clear from the context and so we write $\cdot \vdash N : \mathsf{value}\ v\ true$ instead of $\cdot \vdash_{\Sigma_{Gen9.2}} N : \mathsf{value}\ v\ true$.

Proof. Each part follows by induction and case analysis on the last steps of T. In each case, we know that the trace cannot be empty, because the variable bindings $y{:}\langle \mathsf{eval}\, e\rangle\ ord$, $y{:}\langle \mathsf{retn}\, v\rangle\ ord$, $y_2{:}\langle \mathsf{cont}\, f\rangle\ ord$, $y_3{:}\langle \mathsf{cont2}\, f\rangle\ ord$, $y{:}\langle \mathsf{error}\rangle\ ord$, and $y_2{:}\langle \mathsf{handle}\, e\rangle\ ord$, respectively, appear in the final process state but not the initial process state. Therefore, $T = T''; S$ for some T'' and S.

There are two ways of formulating the proof of this inversion lemma. The specific formulation does a great deal of explicit case analysis but is closer in style to the preservation lemmas. We also give a more generic formulation of the proof which avoids much of this case analysis, in large part by operating in terms of the input and output interfaces introduced in Section 4.3.

Specific formulation (Part 4)
Given $(T''; S) :: (x_0{:}\langle \mathsf{gen}\rangle\ ord) \leadsto^*_{\Sigma_{Gen9.2}} \Theta\{y_1{:}\langle \mathsf{gen}\rangle\ ord,\ y_2{:}\langle \mathsf{gen}\rangle\ ord,\ y_3{:}\langle \mathsf{cont2}\, f\rangle\ ord\}$, we perform case analysis on S. We give two representative cases:

Case $S = \{z\} \leftarrow \mathsf{gen/eval}\, e\, x''$

We have that $\Theta\{y_1{:}\langle \mathsf{gen}\rangle\ ord,\ y_2{:}\langle \mathsf{gen}\rangle\ ord,\ y_3{:}\langle \mathsf{cont2}\, f\rangle\ ord\} = \Theta'\{z{:}\langle \mathsf{eval}\, e\rangle\ ord\}$. It cannot be the case that $z = y_1$, $z = y_2$, or $z = y_3$ – the propositions don't match. Therefore, we can informally describe the substructural context as a frame Θ_{2H} with two holes that are filled as $\Theta_{2H}\{y_1{:}\langle \mathsf{gen}\rangle\ ord,\ y_2{:}\langle \mathsf{gen}\rangle\ ord,\ y_3{:}\langle \mathsf{cont2}\, f\rangle\ ord\}\{z{:}\langle \mathsf{eval}\, e\rangle\ ord\}$. (We haven't actually introduced frames with two holes; the reasoning we do with two-hole contexts here could also be done following the structure of the cut admissibility proof, Theorem 3.6.)

If we call the induction hypothesis on T'', we get that

$$
\begin{array}{ll}
T'' = & (x_0{:}\langle \mathsf{gen}\rangle\ ord) \\
T''' & \\
& \Theta_{2H}\{x'{:}\langle \mathsf{gen}\rangle\ ord\}\{x''{:}\langle \mathsf{gen}\rangle\ ord\} \\
& \{y_1, y_2, y_3\} \leftarrow \mathsf{gen/cont2}\, f\, (x' \bullet !N) \\
& \Theta_{2H}\{y_1{:}\langle \mathsf{gen}\rangle\ ord,\ y_2{:}\langle \mathsf{gen}\rangle\ ord,\ y_3{:}\langle \mathsf{cont2}\, f\rangle\ ord\}\{x''{:}\langle \mathsf{gen}\rangle\ ord\}
\end{array}
$$

The steps $\{y_1, y_2, y_3\} \leftarrow \mathsf{gen/cont2}\, f\, (x' \bullet !N)$ and $\{z\} \leftarrow \mathsf{gen/eval}\, e\, x''$ can be permuted, so we let $T' = T'''; \{z\} \leftarrow \mathsf{gen/eval}\, e\, x''$ and have

$$
\begin{array}{ll}
T = & (x_0{:}\langle \mathsf{gen}\rangle\ ord) \\
T''' & \\
& \Theta_{2H}\{x'{:}\langle \mathsf{gen}\rangle\ ord\}\{x''{:}\langle \mathsf{gen}\rangle\ ord\} \\
& \{z\} \leftarrow \mathsf{gen/eval}\, e\, x'' \\
& \Theta_{2H}\{x'{:}\langle \mathsf{gen}\rangle\ ord\}\{z{:}\langle \mathsf{eval}\, e\rangle\ ord\} \\
& \{y_1, y_2, y_3\} \leftarrow \mathsf{gen/cont2}\, f\, (x' \bullet !N) \\
& \Theta_{2H}\{y_1{:}\langle \mathsf{gen}\rangle\ ord,\ y_2{:}\langle \mathsf{gen}\rangle\ ord,\ y_3{:}\langle \mathsf{cont2}\, f\rangle\ ord\}\{z{:}\langle \mathsf{eval}\, e\rangle\ ord\}
\end{array}
$$

Case $S = \{z_1, z_2, z_3\} \leftarrow \mathsf{gen/cont2}\, f'\, (x'' \bullet !N')$

If $z_1 = x_1$, $z_2 = x_2$, or $z_3 = x_3$, then the ordered structure of the context forces the rest of the equalities to hold and we succeed immediately letting $T' = T''$, $f = f'$, and $N = N'$.

If $z_1 \neq x_1$, $z_2 \neq x_2$, and $z_3 \neq x_3$, then we proceed by induction as in the gen/eval case above.

The only other possibilities allowed by the propositions associated with variables are that $z_1 = x_2$, which is impossible because it would force $z_2 = x_3$ and therefore force gen to equal cont2 f, and that $z_2 = x_1$, which is impossible because it would force $z_3 = x_2$ and therefore force cont2 f' to equal gen.

Generic formulation Let Var be the set of relevant variables – $\{y\}$ in parts 1, 2, and 5, $\{y_1, y_2\}$ in parts 3 and 6, and $\{y_1, y_2, y_3\}$ in part 4.

One possibility is that $\emptyset = S^\bullet \cap Var$. If so, it is always the case that $\emptyset = {}^\bullet S \cap Var$ as well, because Var contains no persistent atomic propositions or LF variables. By the induction hypothesis we then get that $T'' = T'''; S'$, where $S' = \{y\} \leftarrow \mathsf{gen/eval}\, e\, x'$ in part 1, $S' = \{y\} \leftarrow \mathsf{gen/retn}\, v\, (x' \bullet !N)$ in part 2, and so on. In each of the six parts, $S'^\bullet = Var$, so $\emptyset = S'^\bullet \cap {}^\bullet S$ and $(T'''; S'; S) = (T'''; S; S')$. We can conclude letting $T' = (T''; S)$.

If, on the other hand, $S^\bullet \cap Var$ is nonempty, we must show by case analysis that $S^\bullet = Var$ and that furthermore S is the step we were looking for. This is easy in parts 1, 2, and 5 where Var is a singleton set: there is only one rule that can produce an atomic proposition of type eval e, retn v, or error, respectively. In part 4, we observe that, if the variable bindings $y_1{:}\langle\mathsf{gen}\rangle\ ord$, $y_2{:}\langle\mathsf{gen}\rangle\ ord$, and $y_3{:}\langle\mathsf{cont2}\ f\rangle\ ord$ appear in order in the substructural context, there is no step in the signature $\Sigma_{Gen9.2}$ that has y_1 among its output variables that does not also have y_2 and y_3 among its output variables, no step that has y_2 among its output variables that does not also have y_1 and y_3 among its output variables, and so on. (This is a rephrasing of the reasoning we did in the gen/cont2 case of the proof above.) Parts 3 and 6 work by similar reasoning. □

The inversion lemma can be intuitively connected with the idea that the grammar described by a generative signature is *unambiguous*. This will not hold in general. If there was a rule $\mathsf{gen/redundant} : \mathsf{gen} \rightarrowtail \{\mathsf{gen}\}$ in $\Sigma_{Gen9.2}$, for instance, then the final step S could be $\{y_1\} \leftarrow \mathsf{gen/redundant}\, y'$, and this would invalidate our inversion lemma for parts 3, 4, and 6.

Conversely, if we tried to prove an inversion property about traces $(x{:}\langle\mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.2}} \Theta\{y{:}\langle\mathsf{gen}\rangle\ ord\}$, this property would again fail: $V = \{y\}$, and in the case where the last step S is driven by one of the rules gen/cont, gen/cont2, or gen/handle, $S^\bullet$ will be a strict superset of V.

The reason for preferring the generic formulation to the one based on more straightforward case analysis is that the generic formulation is much more compact. The specific formulation in its full elaboration would require enumerating 7 cases for each of the 6 inversion lemmas, leading to proof whose size is in $O(n^2)$ where n is the number of rules in the generative signature. This enormous proof does very little to capture the intuitive reasons why the steps we are interested in can always be rotated to the end. A goal of this chapter to reason we will emphasize the principles by which we can use to reason *concisely* about specifications.

9.2.2 Preservation

Theorem 9.2 ($\Sigma_{Gen9.2}$ is a generative invariant)**.** *If* $T_1 :: (x_0{:}\langle\mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.2}} \Delta$ *and* $S :: \Delta \lightning \rightsquigarrow \Delta'$ *under the signature from Figure 9.1, then* $T_2 :: (x_0{:}\langle\mathsf{gen}\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.2}} \Delta'$.

Again recalling the two-dimensional notation from Chapter 4, the statement of this theorem can be illustrated as follows (dashed lines represent outputs of the theorem):

$$
\begin{array}{ccc}
(x_0{:}\langle \mathsf{gen}\rangle\ ord) & & (x_0{:}\langle \mathsf{gen}\rangle\ ord) \\
\Sigma_{Gen9.2} \downarrow^{*} & & \Sigma_{Gen9.2} \downarrow^{*} \\
\Delta & & \Delta' \\
\Delta & \rightsquigarrow & \Delta'
\end{array}
$$

Proof. By case analysis on S. As in the proofs of Theorem 4.7 and Theorem 6.6, we enumerate the synthetic transitions possible under the signature in Figure 9.1, perform inversion on the structure of T_1, and then use the results of inversion to construct T_2. We give three illustrative cases corresponding to the fragment dealing with functions and parallel application.

Case $\{y\} \leftarrow \mathsf{ev/lam}\ (\lambda x.e)\ x :: \Theta\{x{:}\langle \mathsf{eval}\ (\mathsf{lam}\ \lambda x.e)\rangle\ ord\} \rightsquigarrow \Theta\{y{:}\langle \mathsf{retn}\ (\mathsf{lam}\ \lambda x.e)\rangle\ ord\}$

Applying inversion (Part 1) to T_1, we have

$$
\begin{array}{ll}
T_1 = & (x_0{:}\langle \mathsf{gen}\rangle\ ord) \\
T' & \\
 & \Theta\{x'{:}\langle \mathsf{gen}\rangle\ ord\} \\
 & \{x\} \leftarrow \mathsf{gen/eval}\ (\mathsf{lam}\ \lambda x.e)\ x' \\
 & \Theta\{x{:}\langle \mathsf{eval}\ (\mathsf{lam}\ \lambda x.e)\rangle\ ord\}
\end{array}
$$

We can use T' to construct T_2 as follows:

$$
\begin{array}{ll}
T_2 = & (x_0{:}\langle \mathsf{gen}\rangle\ ord) \\
T' & \\
 & \Theta\{x'{:}\langle \mathsf{gen}\rangle\ ord\} \\
 & \{y\} \leftarrow \mathsf{gen/retn}\ (\mathsf{lam}\ \lambda x.e)\ (x' \bullet !(\mathsf{value/lam}\ (\lambda x.e))) \\
 & \Theta\{y{:}\langle \mathsf{retn}\ (\mathsf{lam}\ \lambda x.e)\rangle\ ord\}
\end{array}
$$

Case $\{y_1, y_2, y_3\} \leftarrow \mathsf{ev/app}\ e_1\ e_2\ x$
$:: \Theta\{x{:}\langle \mathsf{eval}\ (\mathsf{app}\ e_1\ e_2)\rangle\ ord\}$
$\rightsquigarrow \Theta\{y_1{:}\langle \mathsf{eval}\ e_1\rangle\ ord,\ y_2{:}\langle \mathsf{eval}\ e_2\rangle\ ord,\ y_3{:}\langle \mathsf{cont2}\ \mathsf{app1}\rangle\ ord\}$

Applying inversion (Part 1) to T_1, we have

$$
\begin{array}{ll}
T_1 = & (x_0{:}\langle \mathsf{gen}\rangle\ ord) \\
T' & \\
 & \Theta\{x'{:}\langle \mathsf{gen}\rangle\ ord\} \\
 & \{x\} \leftarrow \mathsf{gen/eval}\ (\mathsf{app}\ e_1\ e_2)\ x' \\
 & \Theta\{x{:}\langle \mathsf{eval}\ (\mathsf{app}\ e_1\ e_2)\rangle\ ord\}
\end{array}
$$

We can use T' to construct T_2 as follows:

$$
\begin{array}{ll}
T_2 = & (x_0{:}\langle \mathsf{gen}\rangle\ ord) \\
T' & \\
& \Theta\{x'{:}\langle \mathsf{gen}\rangle\ ord\} \\
& \{y_1', y_2', y\} \leftarrow \mathsf{gen/cont2}\ \mathsf{app1}\ (x' \bullet !\mathsf{okf2/app1}) \\
& \{y_1\} \leftarrow \mathsf{gen/eval}\ e_1\ y_1' \\
& \{y_2\} \leftarrow \mathsf{gen/eval}\ e_2\ y_2' \\
& \Theta\{y_1{:}\langle \mathsf{eval}\ e_1\rangle\ ord,\ y_2{:}\langle \mathsf{eval}\ e_2\rangle\ ord,\ y_3{:}\langle \mathsf{cont2}\ \mathsf{app1}\rangle\ ord\}
\end{array}
$$

Case $\{y\} \leftarrow \mathsf{ev/app1}\ (\lambda x.\, e)\ v_2\ (x_1 \bullet x_2 \bullet x_3)$

$$
\begin{array}{l}
:: \Theta\{x_1{:}\langle \mathsf{retn}\ (\mathsf{lam}\ \lambda x.\, e)\rangle\ ord,\ x_2{:}\langle \mathsf{retn}\ v_2\rangle\ ord,\ x_3{:}\langle \mathsf{cont2}\ \mathsf{app1}\rangle\ ord\} \\
\quad \leadsto \Theta\{y{:}\langle \mathsf{eval}\ ([v_2/x]e)\rangle\ ord\}
\end{array}
$$

Applying inversion (Part 2, twice, and then Part 4) to T_1, we have

$$
\begin{array}{ll}
T_1 = & (x_0{:}\langle \mathsf{gen}\rangle\ ord) \\
T' & \\
& \Theta\{x'{:}\langle \mathsf{gen}\rangle\ ord\} \\
& \{x_1', x_2', x_3\} \leftarrow \mathsf{gen/cont2}\ \mathsf{app1}\ (x' \bullet !N) \\
& \{x_1\} \leftarrow \mathsf{gen/retn}\ (\mathsf{lam}\ \lambda x.e)\ (x_1' \bullet !N_1) \\
& \{x_2\} \leftarrow \mathsf{gen/retn}\ v_2\ (x_2' \bullet !N_2) \\
& \Theta\{x_1{:}\langle \mathsf{retn}\ (\mathsf{lam}\ \lambda x.\, e)\rangle\ ord,\ x_2{:}\langle \mathsf{retn}\ v_2\rangle\ ord,\ x_3{:}\langle \mathsf{cont2}\ \mathsf{app1}\rangle\ ord\}
\end{array}
$$

We can use T' to construct T_2 as follows:

$$
\begin{array}{ll}
T_2 = & (x_0{:}\langle \mathsf{gen}\rangle\ ord) \\
T' & \\
& \Theta\{x'{:}\langle \mathsf{gen}\rangle\ ord\} \\
& \{y\} \leftarrow \mathsf{gen/eval}\ ([v_2/x]e)\ x' \\
& \Theta\{y{:}\langle \mathsf{eval}\ ([v_2/x]e)\rangle\ ord\}
\end{array}
$$

The other cases, corresponding to the rules ev/unit, ev/fail, ev/catch, ev/catcha, ev/catchb, ev/error, ev/errerr, ev/errret, and ev/reterr all proceed similarly by inversion and reconstruction. □

Note that, in the case corresponding to the rule ev/app1, we obtained but did not use three terms $\cdot \vdash N : \mathsf{okf2}\ \mathsf{app1}\ true$, $\cdot \vdash N_1 : \mathsf{value}\ (\mathsf{lam}\ \lambda x.e)\ true$, and $\cdot \vdash N_2 : \mathsf{value}\ v_2\ true$. By traditional inversion on the structure of a deductive derivation, we know that $N = \mathsf{okf2/app1}$ and $N_1 = \mathsf{value/lam}\ (\lambda x.e)$, but that fact was also not necessary here.

9.3 From well-formed to well-typed states

In order to describe those expressions whose evaluations never get stuck, we introduce object level types tp and define a typing judgment $\Gamma \vdash e : tp$. We encode object-level types as LF terms classified by the LF type typ. The unit type $\ulcorner \mathbf{1} \urcorner = \mathsf{unittp}$ classifies units $\ulcorner \langle\rangle \urcorner = \mathsf{unit}$, and the function type $\ulcorner tp_1 \rightharpoonup tp_2 \urcorner = \mathsf{arr} \ulcorner tp_1 \urcorner \ulcorner tp_2 \urcorner$ classifies lambda expressions.

```
of: exp -> typ -> prop.

of/unit:  of unit unittp.
of/lam:   of (lam \x. E x) (arr Tp' Tp)
           <- (All x. of x Tp' -> of (E x) Tp).
of/app:   of (app E1 E2) Tp
           <- of E1 (arr Tp' Tp)
           <- of E2 Tp'.
of/fail:  of fail Tp.
of/catch: of (catch E1 E2) Tp
           <- of E1 Tp
           <- of E2 Tp.

off: frame -> typ -> typ -> prop.
off/let1: off (let1 \x. E' x) Tp' Tp
           <- (All x. of x Tp' -> of (E' x) Tp).

off2: frame -> typ -> typ -> typ -> prop.
off2/app1:  off2 app1 (arr Tp' Tp) Tp' Tp.

gen: typ -> prop ord.
gen/eval:   gen Tp * !of E Tp >-> {eval E}.
gen/retn:   gen Tp * !of V Tp * !value V >-> {retn V}.
gen/cont:   gen Tp * !off F Tp' Tp >-> {gen Tp' * cont F}.
gen/cont2:  gen Tp * !off2 F Tp1 Tp2 Tp
             >-> {gen Tp1 * gen Tp2 * cont2 F}.
gen/error:  gen Tp >-> {error}.
gen/handle: gen Tp * !of E2 Tp >-> {gen Tp * handle E2}.
```

Figure 9.4: Generative invariant: well-typed process states

In a syntax-directed type system, each syntactic construct is associated with a different typing rule. These are the typing rules necessary for describing the language constructs in Figure 9.1:

$$\frac{}{\Gamma \vdash \langle\rangle : \mathbf{1}} \qquad \frac{\Gamma, x{:}tp' \vdash e : tp}{\Gamma \vdash \lambda x.e : tp' \rightharpoonup tp} \qquad \frac{\Gamma \vdash e_1 : tp' \rightharpoonup tp \quad \Gamma \vdash e_2 : tp'}{\Gamma \vdash e_1\, e_2 : tp}$$

$$\frac{}{\Gamma \vdash \mathsf{fail} : tp} \qquad \frac{\Gamma \vdash e_1 : tp \quad \Gamma \vdash e_2 : tp}{\Gamma \vdash \mathsf{try}\, e_1\, \mathsf{ow}\, e_2 : tp}$$

We can adequately encode derivations of the judgment $x_1{:}tp_1, \ldots, x_n{:}tp_n \vdash e : tp$ as SLS derivations $x_1{:}\mathsf{exp}, \ldots, x_n{:}\mathsf{exp}; y_1{:}(\mathsf{of}\ x_1 \ulcorner tp_1 \urcorner)\ pers, \ldots, y_n{:}(\mathsf{of}\ x_1 \ulcorner tp_n \urcorner)\ pers \vdash \mathsf{of} \ulcorner e \urcorner \ulcorner tp \urcorner$ under the signature in Figure 9.3.

This typing judgment allows us to describe well-formed initial states, but it is not sufficient to describe intermediate states. To this end, we describe typing rules for frames, refining the negative predicates $\mathsf{okf}\ f$ and $\mathsf{okf2}\ f$ from Figure 9.2. The SLS proposition describing well-typed sequential frames is $(\mathsf{off}\ f \ulcorner tp' \urcorner \ulcorner tp \urcorner)$. This proposition expresses that the frame f *expects*

a returned result with type tp' and *produces* a computation with type tp.[7] The parallel version is $(\mathsf{off}\ f \ulcorner tp_1 \urcorner \ulcorner tp_2 \urcorner \ulcorner tp \urcorner)$, and expects two sub-computations with types tp_1 and tp_2, respectively, in order to produce a computation of type tp. These judgments are given in Figure 9.4.

The generative rules in Figure 9.4 are our first use of an *indexed* nonterminal, $\mathsf{gen} \ulcorner tp \urcorner$, which generates computations that, upon successful return, will produce values v such that $\cdot \vdash v : tp$.

9.3.1 Inversion

The structure of inversion lemmas is entirely unchanged, except that it has to account for type indices. We only state two cases of the inversion lemma, the one corresponding to gen/eval and the one corresponding to gen/cont. These two cases suffice to set up the template that all other cases follow.

Lemma (Inversion – Figure 9.2, partial)**.**

1. *If* $T :: (x_0{:}\langle \mathsf{gen}\ tp_0 \rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.4}} \Theta\{y{:}\langle \mathsf{eval}\ e \rangle\ ord\}$,
   then $T = (T'; \{y\} \leftarrow \mathsf{gen/eval}\ tp\ e\ (x' \bullet !N))$,
   where $\cdot \vdash N : \mathsf{of}\ e\ tp\ true$

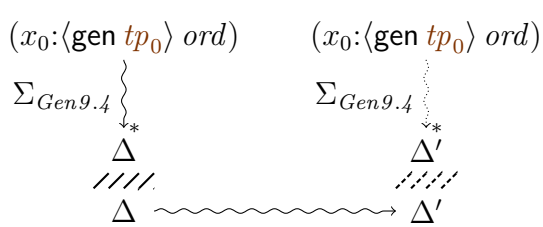
2. *If* $T :: (x_0{:}\langle \mathsf{gen}\ tp_0 \rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.4}} \Theta\{y_1{:}\langle \mathsf{gen}\ tp' \rangle\ ord,\ y_2{:}\langle \mathsf{cont}\ f \rangle\ ord\}$,
   then $T = (T'; \{y_1, y_2\} \leftarrow \mathsf{gen/cont}\ tp\ f\ tp'\ (x' \bullet !N))$,
   where $\cdot \vdash N : \mathsf{off}\ f\ tp'\ tp\ true$.

In each instance above, $T' :: (x_0{:}\langle \mathsf{gen}\ tp_0 \rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.4}} \Theta\{x'{:}\langle \mathsf{gen}\ tp \rangle\ ord\}$*, where the variables* x_0 *and* x' *may or may not be the same. (They are the same iff* $T' = \diamond$*, and if they are the same that implies* $tp_0 = tp$*.)*

9.3.2 Preservation

Theorem 9.3 only differs from Theorem 9.2 because it mentions the type index. Each object-level type tp_0 describes a different world (that is, a different set of SLS process states), and evaluation under the rules in Figure 9.1 always stays within the same world.

Theorem 9.3 ($\Sigma_{Gen9.4}$ is a generative invariant)**.** *If* $T_1 :: (x_0{:}\langle \mathsf{gen}\ tp_0 \rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.4}} \Delta$ *and* $S :: \Delta\lightning \rightsquigarrow \Delta'$ *under the signature from Figure 9.1, then* $T_2 :: (x_0{:}\langle \mathsf{gen}\ tp_0 \rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.4}} \Delta'$.



[7]The judgment we encode in SLS as $(\mathsf{off}\ f \ulcorner tp' \urcorner \ulcorner tp \urcorner)$ is written $f : tp' \Rightarrow tp$ in [Har12, Chapter 27].

In the proof of Theorem 9.2, we observed that the applicable inversion on the generative trace gave us derivations like $\cdot \vdash N : \mathsf{okf2}\,\mathsf{app1}\,true$. We did not need these side derivations to complete the proof, but we noted that they were amenable to traditional inversion. Traditional inversion will be critical in proving that the generative invariant described by $\Sigma_{Gen9.4}$ is preserved. It is a solved problem to describe, prove, and mechanize traditional inversion lemmas on deductive derivations; we merely point out when we are using a traditional inversion property in the proof below.

Proof. As always, the proof proceeds by enumeration, inversion, and reconstruction. We give two representative cases:

Case $\{y\} \leftarrow \mathsf{ev/app1}\,(\lambda x.\,e)\,v_2\,(x_1 \bullet x_2 \bullet x_2)$
$\quad :: \Theta\{x_1{:}\langle\mathsf{retn}\,(\mathsf{lam}\,\lambda x.\,e)\rangle\,ord,\ x_2{:}\langle\mathsf{retn}\,v_2\rangle\,ord,\ x_3{:}\langle\mathsf{cont2}\,\mathsf{app1}\rangle\,ord\}$
$\quad\quad \rightsquigarrow \Theta\{y{:}\langle\mathsf{eval}\,([v_2/x]e)\rangle\,ord\}$

Applying inversion to T_1, we have

$$
\begin{array}{ll}
T_1 = & (x_0{:}\langle\mathsf{gen}\,tp_0\rangle\,ord) \\
T' & \\
& \Theta\{x'{:}\langle\mathsf{gen}\,tp\rangle\,ord\} \\
& \{x'_1, x'_2, x_3\} \leftarrow \mathsf{gen/cont2}\,tp\,\mathsf{app1}\,tp''\,tp'\,(x' \bullet !N) \\
& \Theta\{x'_1{:}\langle\mathsf{gen}\,tp''\rangle\,ord,\ x'_2{:}\langle\mathsf{gen}\,tp'\rangle\,ord,\ x_3{:}\langle\mathsf{cont2}\,\mathsf{app1}\rangle\,ord\} \\
& \{x_1\} \leftarrow \mathsf{gen/retn}\,tp''\,(\mathsf{lam}\,\lambda x.e)\,(x'_1 \bullet !N_1 \bullet !N_{v1}) \\
& \{x_2\} \leftarrow \mathsf{gen/retn}\,v_2\,(x'_2 \bullet !N_2 \bullet !N_{v2}) \\
& \Theta\{x_1{:}\langle\mathsf{retn}\,(\mathsf{lam}\,\lambda x.\,e)\rangle\,ord,\ x_2{:}\langle\mathsf{retn}\,v_2\rangle\,ord,\ x_3{:}\langle\mathsf{cont2}\,\mathsf{app1}\rangle\,ord\}
\end{array}
$$

where

- $\cdot \vdash N : \mathsf{off2}\,\mathsf{app1}\,tp''\,tp'\,tp\,true$.
  By traditional inversion we know $tp'' = \mathsf{arr}\,tp'\,tp$ and $N = \mathsf{off2/app1}\,tp'\,tp$.
- $\cdot \vdash N_1 : \mathsf{of}\,(\mathsf{lam}\,\lambda x.e)\,\mathsf{arr}\,tp'\,tp\,true$.
  By traditional inversion we know $x{:}\mathsf{exp}; dx : \mathsf{of}\,x\,tp'\,pers \vdash N'_1 : \mathsf{of}\,e\,tp\,true$.
- $\cdot \vdash N_2 : \mathsf{of}\,v_2\,tp'$.

With these derivations, variable substitution (Theorem 3.4), and cut admissibility (Theorem 3.6), we have a derivation of $\cdot \vdash [\![N_2/dx]\!]([v_2/x]N'_1) : \mathsf{of}\,([v_2/x]e)\,tp\,true$.[8] We can therefore use T' to construct T_2 as follows:

$$
\begin{array}{ll}
T_2 = & (x_0{:}\langle\mathsf{gen}\,tp_0\rangle\,ord) \\
T' & \\
& \Theta\{x'{:}\langle\mathsf{gen}\,tp\rangle\,ord\} \\
& \{y\} \leftarrow \mathsf{gen/eval}\,tp\,([v_2/x]e)\,(x' \bullet !([\![N_2/dx]\!]([v_2/x]N'_1))) \\
& \Theta\{y{:}\langle\mathsf{eval}\,([v_2/x]e)\rangle\,ord\}
\end{array}
$$

Case $\{y_1, y_2\} \leftarrow \mathsf{ev/catch}\,(\lambda x.e)\,v_2\,x$
$\quad :: \Theta\{x{:}\langle\mathsf{eval}\,(\mathsf{catch}\,e_1\,e_2)\rangle\,ord\} \rightsquigarrow \Theta\{y_1{:}\langle\mathsf{eval}\,e_1\rangle\,ord,\ y_2{:}\langle\mathsf{handle}\,e_2\rangle\,ord\}]$

Applying inversion to T_1, we have

[8] We know by subordination that x is not free in tp, so $[v_2/x]tp = tp$.

$$
\begin{array}{ll}
T_1 = & (x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \\
T' & \\
& \Theta\{x'{:}\langle \mathsf{gen}\ tp\rangle\ ord\} \\
& \{x\} \leftarrow \mathsf{gen/eval}\ tp\ (\mathsf{catch}\ e_1\ e_2)\ (x' \bullet !N) \\
& \Theta\{x{:}\langle \mathsf{eval}\ (\mathsf{catch}\ e_1\ e_2)\rangle\ ord\}
\end{array}
$$

where $\cdot \vdash N : \mathsf{of}\ (\mathsf{catch}\ \mathsf{e}_1\ \mathsf{e}_2)\ tp$.

By traditional inversion on N we know $\cdot \vdash N_1 : \mathsf{of}\ e_1\ tp\ true$ and $\cdot \vdash N_2 : \mathsf{of}\ e_2\ tp\ true$. We can therefore use T' to construct T_2 as follows:

$$
\begin{array}{ll}
T_1 = & (x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \\
T' & \\
& \Theta\{x'{:}\langle \mathsf{gen}\ tp\rangle\ ord\} \\
& \{y_1', y_2\} \leftarrow \mathsf{gen/handle}\ tp\ e_2\ (x' \bullet !N_2) \\
& \{y_1\} \leftarrow \mathsf{gen/eval}\ tp\ e_1\ (y_1' \bullet !N_1) \\
& \Theta\{y_1{:}\langle \mathsf{eval}\ e_1\rangle\ ord,\ y_2{:}\langle \mathsf{handle}\ e_2\rangle\ ord\}
\end{array}
$$

The other cases follow the same pattern. □

Dealing with type preservation is, in an sense, no more difficult than dealing with well-formedness invariants. Theorem 9.3 furthermore follows the contours of a standard progress and preservation proof for an abstract machine like Harper's $\mathcal{K}\{\mathsf{nat}{\rightharpoonup}\}$ [Har12, Chapter 27]. Unlike the on-paper formalism used by Harper, the addition of parallel evaluation in our specification does not further complicate the statement or the proof of the preservation theorem.

9.4 State

Ambient state, encoded in mobile and persistent propositions, was used to describe mutable storage in Section 6.5.1, call-by-need evaluation in Section 6.5.2, and the environment semantics in Section 6.5.3. The technology needed to describe generative invariants for each of these specifications is similar. We will consider the extension of our program from Figure 9.1 with the semantics of mutable storage from Figure 6.14. This specification adds a mobile atomic proposition $\mathsf{cell}\ l\ v$, which the generative signature will treat as a new terminal.

The intuition behind mutable cells is that they exist in tandem with locations l of LF type mutable_loc, giving the non-control part of a process state the following general form:

$$(l_1{:}\mathsf{mutable\_loc}, \ldots, l_n{:}\mathsf{mutable\_loc};\ \langle \mathsf{cell}\ l_1\ v_1\rangle\ eph,\ \ \ldots,\ \langle \mathsf{cell}\ l_n\ v_n\rangle\ eph,\ \ \ldots)$$

Naively, we might attempt to describe such process states with the block-like rule $\mathsf{gen/cell/bad} : \forall v.\ !\mathsf{value}\ v \rightarrowtail \{\exists l.\mathsf{cell}\ l\ v\}$. The problem with such a specification is that it makes cells unable to refer to themselves, a situation that can certainly occur. A canonical example, using back-patching to implement recursion, is traced out in Figure 9.4, which describes a trace classified by:

$$(\cdot;\ x_0{:}\langle \mathsf{eval}\ulcorner \mathsf{let}\ f = (\mathsf{ref}\ \lambda x.\langle\rangle)\ \mathsf{in}\ \mathsf{let}\ x = (f := \lambda x.(!f)x)\ \mathsf{in}\ e\urcorner\rangle\ ord) \quad \rightsquigarrow^*$$
$$(l_1{:}\mathsf{mutable\_loc};\ \ y_2{:}\langle \mathsf{cell}\ l_1\ (\mathsf{lam}\ \lambda x.\ \mathsf{app}\ (\mathsf{get}\ (\mathsf{loc}\ l_1))\ x)\rangle\ eph,\ \ x_{17}{:}\langle \mathsf{eval}\ [(\mathsf{loc}\ l_1)/f, \mathsf{unit}/x]\ulcorner e\urcorner)\rangle\ ord)$$

The name of this problem is *parameter dependency* – the term v in gen/cell/bad has to be instantiated before the parameter l is introduced. As a result, the trace in Figure 9.4 includes a step

$$\{x_{16}, y_2\} \leftarrow \mathsf{ev/set2}\ \ldots\ (x_{15} \bullet x_{14} \bullet y_1)$$

that transitions from a state that can be described by Figure 9.2 extended with gen/cell/bad to a state that cannot be described by this signature. This means that gen/cell/bad cannot be the basis of a generative invariant: it's not invariant!

The solution is to create cells in two steps. The first rule, a promise rule, creates the location l and associates a mobile nonterminal gencell with that location. A second fulfill rule consumes that nonterminal and creates the actual mutable cell. Because gencell is a mobile nonterminal, the promise *must* be fulfilled in order for the final state to pass through the restriction operation. As we have already seen, there is not much of a technical difference between well-formedness invariants and well-typedness invariants; Figure 9.6 describes a generative signature that captures type information. This specification introduces two nonterminals. The first is the aforementioned mobile nonterminal gencell l, representing the promise to eventually create a cell corresponding to the location l. The second is a persistent nonterminal ofcell $l\ tp$. The collection of ofcell propositions introduced by a generative trace collectively plays the role of a *store typing* in [Pie02, Chapter 13] or a *signature* in [Har12, Chapter 35]. This promise-then-fulfill pattern appears to be an significant one, and it can be described quite naturally in generative signatures, despite being absent from work on regular-worlds-based reasoning about LF and Linear LF specifications.

9.4.1 Inversion

When we add mutable state, we must significantly generalize the *statement* of inversion lemmas. Derivations and expressions now exist in a world with arbitrary locations l:mutable_loc that are paired with persistent propositions ofcell $l\ tp$.[9]

Lemma (Inversion – Figure 9.6, partial)**.**

1. *If* $T :: (\cdot; x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.6}} (\Psi; \Theta\{y{:}\langle \mathsf{eval}\ e\rangle\ ord\})$,
   then $T = (T'; \{y\} \leftarrow \mathsf{gen/eval}\ tp\ e\ (x' \bullet !N))$,
   where $\Psi; \Delta \vdash N : \mathsf{of}\ e\ tp\ true$,
   $T' :: (\cdot; x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.6}} (\Psi'; \Theta\{x'{:}\langle \mathsf{gen}\ tp\rangle\ ord\})$,
   and Δ *is the persistent part of* $\Theta\{x'{:}\langle \mathsf{gen}\ tp\rangle\ ord\}$.

2. *If* $T :: (\cdot; x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.6}} (\Psi; \Theta\{y_1{:}\langle \mathsf{gen}\ tp'\rangle\ ord,\ y_2{:}\langle \mathsf{cont}\ f\rangle\ ord\})$,
   then $T = (T'; \{y_1, y_2\} \leftarrow \mathsf{gen/cont}\ tp\ f\ tp'\ (x' \bullet !N))$,
   where $\Psi; \Delta \vdash N : \mathsf{off}\ f\ tp'\ tp\ true$,

[9]This purely persistent world fits the pattern of regular worlds. As such, it can be described either with the single rule $\forall tp.\ \{\exists l.\ \mathsf{ofcell}\ l\ tp\}$ or with the equivalent block some tp:typ block l:mutable_loc, $x : \langle \mathsf{ofcell}\ l\ tp\rangle\ pers$.

$x_0{:}\langle \mathsf{eval}\ \ulcorner \mathsf{let}\ f = (\mathsf{ref}\ \lambda x.\langle\rangle)\ \mathsf{in}\ \mathsf{let}\ x = (f := \lambda x.(!f)x)\ \mathsf{in}\ e \urcorner\rangle$

$\{x_1, x_2\} \leftarrow \mathsf{ev/let1}\ \ldots\ x_0$

$x_1{:}\langle \mathsf{eval}\ \ulcorner \mathsf{ref}\ \lambda x.\langle\rangle \urcorner\rangle,\ \ x_2{:}\langle \mathsf{cont}\ (\mathsf{let1}\ \lambda f.\ulcorner \mathsf{let}\ x = (f := \lambda x.(!f)x)\ \mathsf{in}\ e\urcorner)\rangle$

$\{x_3, x_4\} \leftarrow \mathsf{ev/ref}\ \ldots\ x_1$

$x_3{:}\langle \mathsf{eval}\ \ulcorner \lambda x.\langle\rangle \urcorner\rangle,\ \ x_4{:}\langle \mathsf{cont}\ \mathsf{ref1}\rangle,\ \ x_2{:}\langle \mathsf{cont}\ (\mathsf{let1}\ \lambda f.\ulcorner \mathsf{let}\ x = (f := \lambda x.(!f)x)\ \mathsf{in}\ e\urcorner)\rangle$

$\{x_5\} \leftarrow \mathsf{ev/lam}\ \ldots\ x_3$

$x_5{:}\langle \mathsf{retn}\ \ulcorner \lambda x.\langle\rangle \urcorner\rangle,\ \ x_4{:}\langle \mathsf{cont}\ \mathsf{ref1}\rangle,\ \ x_2{:}\langle \mathsf{cont}\ (\mathsf{let1}\ \lambda f.\ulcorner \mathsf{let}\ x = (f := \lambda x.(!f)x)\ \mathsf{in}\ e\urcorner)\rangle$

$\{l_1, x_6, y_1\} \leftarrow \mathsf{ev/ref1}\ \ldots\ (x_5 \bullet x_4)$

$y_1{:}\langle \mathsf{cell}\ l_1\ \ulcorner \lambda x.\langle\rangle \urcorner\rangle,\ \ x_6{:}\langle \mathsf{retn}\ \mathsf{loc}\ l_1\rangle,\ \ x_2{:}\langle \mathsf{cont}\ (\mathsf{let1}\ \lambda f.\ulcorner \mathsf{let}\ x = (f := \lambda x.(!f)x)\ \mathsf{in}\ e\urcorner)\rangle$

$\{x_7\} \leftarrow \mathsf{ev/let1}\ \ldots\ (x_6 \bullet x_2)$

$y_1{:}\langle \mathsf{cell}\ l_1\ \ulcorner \lambda x.\langle\rangle \urcorner\rangle,\ \ x_7{:}\langle \mathsf{eval}\ (\mathsf{let}\ (\mathsf{set}\ (\mathsf{loc}\ l_1)\ (\mathsf{lam}\ \lambda x.\ \mathsf{app}\ (\mathsf{get}\ (\mathsf{loc}\ l_1))\ x))\ \lambda x.\ ([(\mathsf{loc}\ l_1)/f]\ulcorner e\urcorner))\rangle$

$\{x_8, x_9\} \leftarrow \mathsf{ev/let}\ \ldots x_7$

$y_1{:}\langle \mathsf{cell}\ l_1\ \ulcorner \lambda x.\langle\rangle \urcorner\rangle,\ \ x_8{:}\langle \mathsf{eval}\ (\mathsf{set}\ (\mathsf{loc}\ l_1)\ (\mathsf{lam}\ \lambda x.\ \mathsf{app}\ (\mathsf{get}\ (\mathsf{loc}\ l_1))\ x))\rangle,\ \ x_9{:}\langle \mathsf{cont}\ (\mathsf{let1}\ \lambda x.\ ([(\mathsf{loc}\ l_1)/f]\ulcorner e\urcorner))\rangle$

$\{x_{10}, x_{11}\} \leftarrow \mathsf{ev/set}\ \ldots\ x_8$

$y_1{:}\langle \mathsf{cell}\ l_1\ \ulcorner \lambda x.\langle\rangle \urcorner\rangle,\ \ x_{10}{:}\langle \mathsf{eval}\ (\mathsf{loc}\ l_1)\rangle,\ \ x_{11}{:}\langle \mathsf{cont}\ (\mathsf{set1}\ (\mathsf{lam}\ \lambda x.\ \mathsf{app}\ (\mathsf{get}\ (\mathsf{loc}\ l_1))\ x))\rangle,\ \ x_9{:}\langle \mathsf{cont}\ (\mathsf{let1}\ \lambda x.\ ([(\mathsf{loc}\ l_1)/f]\ulcorner e\urcorner))\rangle$

$\{x_{12}\} \leftarrow \mathsf{ev/loc}\ \ldots\ x_{10}$

$y_1{:}\langle \mathsf{cell}\ l_1\ \ulcorner \lambda x.\langle\rangle \urcorner\rangle,\ \ x_{12}{:}\langle \mathsf{retn}\ (\mathsf{loc}\ l_1)\rangle,\ \ x_{11}{:}\langle \mathsf{cont}\ (\mathsf{set1}\ (\mathsf{lam}\ \lambda x.\ \mathsf{app}\ (\mathsf{get}\ (\mathsf{loc}\ l_1))\ x))\rangle,\ \ x_9{:}\langle \mathsf{cont}\ (\mathsf{let1}\ \lambda x.\ ([(\mathsf{loc}\ l_1)/f]\ulcorner e\urcorner))\rangle$

$\{x_{13}, x_{14}\} \leftarrow \mathsf{ev/set1}\ \ldots\ (x_{12} \bullet x_{11})$

$y_1{:}\langle \mathsf{cell}\ l_1\ \ulcorner \lambda x.\langle\rangle \urcorner\rangle,\ \ x_{13}{:}\langle \mathsf{eval}\ (\mathsf{lam}\ \lambda x.\ \mathsf{app}\ (\mathsf{get}\ (\mathsf{loc}\ l_1))\ x)\rangle,\ \ x_{14}{:}\langle \mathsf{cont}\ (\mathsf{set2}\ l_1)\rangle,\ \ x_9{:}\langle \mathsf{cont}\ (\mathsf{let1}\ \lambda x.\ ([(\mathsf{loc}\ l_1)/f]\ulcorner e\urcorner))\rangle$

$\{x_{15}\} \leftarrow \mathsf{ev/lam}\ \ldots\ x_{13}$

$y_1{:}\langle \mathsf{cell}\ l_1\ \ulcorner \lambda x.\langle\rangle \urcorner\rangle,\ \ x_{15}{:}\langle \mathsf{retn}\ (\mathsf{lam}\ \lambda x.\ \mathsf{app}\ (\mathsf{get}\ (\mathsf{loc}\ l_1))\ x)\rangle,\ \ x_{14}{:}\langle \mathsf{cont}\ (\mathsf{set2}\ l_1)\rangle,\ \ x_9{:}\langle \mathsf{cont}\ (\mathsf{let1}\ \lambda x.\ ([(\mathsf{loc}\ l_1)/f]\ulcorner e\urcorner))\rangle$

$\{x_{16}, y_2\} \leftarrow \mathsf{ev/set2}\ \ldots\ (x_{15} \bullet x_{14} \bullet y_1)$

$y_2{:}\langle \mathsf{cell}\ l_1\ (\mathsf{lam}\ \lambda x.\ \mathsf{app}\ (\mathsf{get}\ (\mathsf{loc}\ l_1))\ x)\rangle,\ \ x_{16}{:}\langle \mathsf{retn}\ \mathsf{unit}\rangle,\ \ x_9\langle \mathsf{cont}\ (\mathsf{let1}\ \lambda x.\ ([(\mathsf{loc}\ l_1)/f]\ulcorner e\urcorner))\rangle$

$\{x_{17}\} \leftarrow \mathsf{ev/let1}\ \ldots\ (x_{16} \bullet x_9)$

$y_2{:}\langle \mathsf{cell}\ l_1\ (\mathsf{lam}\ \lambda x.\ \mathsf{app}\ (\mathsf{get}\ (\mathsf{loc}\ l_1))\ x)\rangle,\ \ x_{17}{:}\langle \mathsf{eval}\ [(\mathsf{loc}\ l_1)/f, \mathsf{unit}/x]\ulcorner e\urcorner\rangle$

Figure 9.5: Back-patching, with judgments (ord and eph) and arguments corresponding to implicit quantifiers elided

```
ofcell: mutable_loc -> typ -> prop pers.
gencell: mutable_loc -> prop lin.

value/loc: value (loc L).

of/loc: of (loc L) (reftp Tp)
         <- ofcell L Tp.
of/ref: of (ref E) (reftp Tp)
         <- of E Tp.
of/get: of (get E) Tp
         <- of E (reftp Tp).
of/set: of (set E1 E2) unittp
         <- of E1 (reftp Tp)
         <- of E2 Tp.

off/ref1: off ref1 Tp (reftp Tp).
off/get1: off get1 (reftp Tp) Tp.
off/set1: off (set1 E) (reftp Tp) unittp
           <- of E Tp.
off/set2: off (set2 L) Tp unittp
           <- ofcell L Tp.

gencell/promise: {Exists l. !ofcell l Tp * $gencell l}.
gencell/fulfill: $gencell L * !ofcell L Tp * !of V Tp * !value V
                  >-> {$cell L V}.
```

Figure 9.6: Generative invariant: well-typed mutable storage

$T' :: (\cdot; x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.6}} (\Psi'; \Theta\{x'{:}\langle \mathsf{gen}\ tp\rangle\ ord\})$,
and Δ *is the persistent part of* $\Theta\{x'{:}\langle \mathsf{gen}\ tp\rangle\ ord\}$.

3. *If* $T :: (\cdot; x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.6}} (\Psi; \Theta\{y{:}\langle \mathsf{cell}\ l\ v\rangle\ eph\})$,
then $T = (T'; \{y\} \leftarrow \mathsf{gencell/fulfill}\ l\ tp\ v\ (x' \bullet x_t \bullet !N \bullet !N_v))$,
where $x_t{:}\langle \mathsf{ofcell}\ l\ tp\rangle\ pers \in \Delta$, $\Psi; \Delta \vdash N : \mathsf{of}\ v\ tp\ true$, $\Psi; \Delta \vdash N_v : \mathsf{value}\ v\ true$,
$T' :: (\cdot; x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.6}} (\Psi'; \Theta\{x'{:}\langle \mathsf{gencell}\ l\rangle\ eph\})$,
and Δ *is the persistent part of* $\Theta\{x'{:}\langle \mathsf{gencell}\ l\rangle\ eph\}$.

Despite complicating the statement of inversion theorems, the addition of mutable state does nothing to change the structure of these theorems. The new inversion lemma (part 3 above) follows the pattern established in Section 9.2.1.

9.4.2 Uniqueness

To prove that our generative invariant for mutable storage is maintained, we need one property besides inversion; we'll refer to it as the *unique index* property. This is the property that, under the

generative signature described by $\Sigma_{Gen9.6}$, locations always map *uniquely* to persistent positive propositions x_t:ofcell l tp.

Lemma (Unique indices of $\Sigma_{Gen9.6}$)**.**

1. *If* $T :: (\cdot; x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.6}} (\Psi; \Delta)$,
   $x{:}\langle \mathsf{ofcell}\ l\ tp\rangle\ pers \in \Delta$,
   and $y{:}\langle \mathsf{ofcell}\ l\ tp'\rangle\ pers \in \Delta$,
   then $x = y$ *and* $tp = tp'$.

Proof. Induction and case analysis on the last steps of the trace T. □

9.4.3 Preservation

As it was with inversion, the statement of preservation is substantially altered by the addition of locations and mutable state, even though the structure of the proof is not. In particular, because ofcell is a *persistent* nonterminal, we have to expressly represent the fact that the restriction operator $(\Psi; \Delta)\lightning$ will modify the context Δ by erasing the store typing.

Theorem 9.4 ($\Sigma_{Gen9.6}$ is a generative invariant)**.** *If* $T_1 :: (\cdot; x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.6}} (\Psi; \Delta)$ *and* $S :: (\Psi; \Delta)\lightning \rightsquigarrow (\Psi'; \Delta')$ *under the combined signature from Figure 9.1 and Figure 6.14, then* $(\Psi'; \Delta') = (\Psi'; \Delta'')\lightning$ *for some* Δ'' *such that* $T_2 :: (\cdot; x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.6}} (\Psi'; \Delta'')$.

$$
\begin{tikzcd}
(\cdot;\ x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \arrow[d, squiggly, "\Sigma_{Gen9.6}"'] & (\cdot;\ x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \arrow[d, dotted, squiggly, "\Sigma_{Gen9.6}"'] \\
(\Psi; \Delta) \arrow[d, squiggly, "*"] & (\Psi'; \Delta'') \arrow[d, squiggly, "*"] \\
(\Psi; \Delta)\lightning \arrow[r, squiggly] & (\Psi'; \Delta')
\end{tikzcd}
$$

Proof. As always, the proof proceeds by enumeration, inversion, and reconstruction. The only interesting cases are the three that actually manipulate state, corresponding to ev/ref1, ev/get1, and ev/set2. Recall these three rules from Figure 6.14:

```
ev/ref1: retn V * cont ref1
          >-> {Exists l. $cell l V * retn (loc l)}.
ev/get1: retn (loc L) * cont get1 * $cell L V
          >-> {retn V * $cell L V}.
ev/set2: retn V2 * cont (set2 L) * $cell L _
          >-> {retn unit * $cell L V2}.
```

Reasoning about the last two cases is similar, so we only give the cases for ev/ref and ev/get1 below.

Case $\{l, y_1, y_2\} \leftarrow \mathsf{ev/ref1}\ v\ (x_1 \bullet x_2)$
$:: (\Psi;\ \Theta\{x_1{:}\langle \mathsf{retn}\ v\rangle\ ord,\ x_2{:}\langle \mathsf{cont}\ \mathsf{ref1}\rangle\ ord\})$
$\rightsquigarrow (\Psi,\ l{:}\mathsf{mutable\_loc};\ \Theta\{y_1{:}\langle \mathsf{cell}\ l\ v\rangle\ eph,\ y_2{:}\langle \mathsf{retn}\ (\mathsf{loc}\ l)\rangle\ ord\})$

$T_1 :: (\cdot;\ x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \rightsquigarrow^* (\Psi;\ \Theta'\{x_1{:}\langle \mathsf{retn}\ v\rangle\ ord,\ x_2{:}\langle \mathsf{cont}\ \mathsf{ref1}\rangle\ ord\})$ for some Θ' such that, for all Ξ, $(\Psi; \Theta'\{\Xi\})\lightning = (\Psi; \Theta\{\Xi\lightning\})$. Applying inversion to T_1, we have

$$
\begin{array}{ll}
T_1 = & (\cdot;\ x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \\
T' & \\
& (\Psi;\ \Theta'\{x'{:}\langle \mathsf{gen}\ tp\rangle\ ord\}) \\
& \{x'_1, x_2\} \leftarrow \mathsf{gen/cont}\ tp\ \mathsf{ref1}\ tp'\ (x' \bullet !N) \\
& (\Psi;\ \Theta'\{x'_1{:}\langle \mathsf{gen}\ tp'\rangle\ ord,\ x_2{:}\langle \mathsf{cont}\ \mathsf{ref1}\rangle\ ord\}) \\
& \{x_1\} \leftarrow \mathsf{gen/retn}\ v\ (x'_1 \bullet !N_1 \bullet !N_{v1}) \\
& (\Psi;\ \Theta'\{x_1{:}\langle \mathsf{retn}\ v\rangle\ ord,\ x_2{:}\langle \mathsf{cont}\ \mathsf{ref1}\rangle\ ord\})
\end{array}
$$

where Δ contains the persistent propositions from Θ' and where

- $\Psi; \Delta \vdash N : \mathsf{off}\ \mathsf{ref1}\ tp'\ tp\ true$. By traditional inversion we know $tp = \mathsf{reftp}\ tp'$ and $N = \mathsf{off/ref1}\ tp'$.
- $\Psi; \Delta \vdash N_1 : \mathsf{of}\ v\ tp'\ true$.
- $\Psi; \Delta \vdash N_{v1} : \mathsf{value}\ v\ true$.

We can use T' to construct T_2 as follows:

$$
\begin{array}{ll}
T_2 = & (\cdot;\ x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \\
T' & \\
& (\Psi;\ \Theta'\{x'{:}\langle \mathsf{gen}\ (\mathsf{reftp}\ tp')\rangle\ ord\}) \\
& \{l, z, y'_1\} \leftarrow \mathsf{gencell/promise}\ tp' \\
& \{y_1\} \leftarrow \mathsf{gencell/fulfill}\ l\ tp'\ v\ (y'_1 \bullet z \bullet !N_1 \bullet !N_{v1}) \\
& (\Psi, l{:}\mathsf{mutable\_loc}; \\
& \quad \Theta'\{z{:}\langle \mathsf{ofcell}\ l\ tp'\rangle\ pers,\ y_1{:}\langle \mathsf{cell}\ l\ v\rangle\ eph,\ x'{:}\langle \mathsf{gen}\ (\mathsf{ref}\ tp')\rangle\ ord\}) \\
& \{y_2\} \leftarrow \mathsf{gen/retn}\ (\mathsf{reftp}\ tp')\ (\mathsf{loc}\ l)\ (x' \bullet !(\mathsf{of/loc}\ l\ tp'\ z) \bullet !(\mathsf{value/loc}\ l)) \\
& (\Psi, l{:}\mathsf{mutable\_loc}; \\
& \quad \Theta'\{z{:}\langle \mathsf{ofcell}\ l\ tp'\rangle\ pers,\ y_1{:}\langle \mathsf{cell}\ l\ v\rangle\ eph,\ x'{:}\langle \mathsf{retn}\ (\mathsf{loc}\ l)\rangle\ ord\})
\end{array}
$$

Restriction removes the persistent nonterminal $z{:}\langle \mathsf{ofcell}\ l\ tp'\rangle\ pers$ from the context, so the restriction of T_2's output is $(\Psi,\ l{:}\mathsf{mutable\_loc};\ \Theta\{y_1{:}\langle \mathsf{cell}\ l\ v\rangle\ eph,\ y_2{:}\langle \mathsf{retn}\ (\mathsf{loc}\ l)\rangle\ ord\})$ as required.

Case $\{y_1, y_2\} \leftarrow \mathsf{ev/get1}\ l\ v\ (x_1 \bullet x_2 \bullet x_3)$

$$
\begin{array}{l}
:: (\Psi;\ \Theta\{x_1{:}\langle \mathsf{retn}\ (\mathsf{loc}\ l)\rangle\ ord,\ x_2{:}\langle \mathsf{cont}\ \mathsf{get1}\rangle\ ord,\ x_3{:}\langle \mathsf{cell}\ l\ v\rangle\ eph\}) \\
\quad \rightsquigarrow (\Psi;\ \Theta\{y_1{:}\langle \mathsf{retn}\ v\rangle\ ord,\ y_2{:}\langle \mathsf{cell}\ l\ v\rangle\ eph\})
\end{array}
$$

$$
\begin{array}{l}
T_1 :: (\cdot; x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \\
\quad \rightsquigarrow^* (\Psi;\ \Theta'\{x_1{:}\langle \mathsf{retn}\ (\mathsf{loc}\ l)\rangle\ ord,\ x_2{:}\langle \mathsf{cont}\ \mathsf{get1}\rangle\ ord,\ x_3{:}\langle \mathsf{cell}\ l\ v\rangle\ eph\})
\end{array}
$$

for some Θ' such that, for all Ξ, $(\Psi; \Theta'\{\Xi\})\lightning = (\Psi; \Theta\{\Xi\lightning\})$. Applying inversion to T_1, we have

$$
\begin{array}{ll}
T_1 = & (\cdot;\ x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \\
T' & \\
& (\Psi;\ \Theta'\{x'{:}\langle \mathsf{gen}\ tp\rangle\ ord,\ x'_3{:}\langle \mathsf{gencell}\ l\rangle\ eph\}) \\
& \{x_3\} \leftarrow \mathsf{gencell/fulfill}\ l\ tp'\ v\ (x'_3 \bullet z_1 \bullet !N_3 \bullet !N_{v3})
\end{array}
$$

```
gencount/finalize: $gencount N >-> {$counter N}.

gencell/promise: $gencount N
                  >-> {Exists l.
                       !ofcell l Tp * $gencell l N * $gencount (s N)}.

gencell/fulfill: $gencell L N * !ofcell L Tp * !of V Tp * !value V
                  >-> {$cell L N V}.
```

Figure 9.7: Generative invariants for cells with unique natural-number tags

$$
\begin{array}{l}
\quad (\Psi;\ \Theta'\{x'{:}\langle \mathsf{gen}\ tp\rangle\ ord,\ x_3{:}\langle \mathsf{cell}\ l\ v\rangle\ eph\}) \\
\{x_1', x_2\} \leftarrow \mathsf{gen/cont}\ tp\ \mathsf{get1}\ tp''\ (x' \bullet !N_2) \\
\{x_1\} \leftarrow \mathsf{gen/retn}\ tp''\ (\mathsf{loc}\ l)\ (x_1' \bullet !N_1 \bullet !N_{v1}) \\
\quad (\Psi;\ \Theta\{x_1{:}\langle \mathsf{retn}\ (\mathsf{loc}\ l)\rangle\ ord,\ x_2{:}\langle \mathsf{cont}\ \mathsf{get1}\rangle\ ord,\ x_2{:}\langle \mathsf{cell}\ l\ v\rangle\ eph\})
\end{array}
$$

where Δ contains the persistent propositions from Θ' and where

- $\Psi; \Delta \vdash N_2 : \mathsf{off}\ \mathsf{get1}\ tp''\ tp\ true$. By traditional inversion we know $tp'' = \mathsf{reftp}\ tp$ and $N_2 = \mathsf{off/get1}\ tp$.
- $\Psi; \Delta \vdash N_1 : \mathsf{of}\ (\mathsf{loc}\ l)\ (\mathsf{reftp}\ tp)\ true$. By traditional inversion we know $N_1 = \mathsf{of/loc}\ l\ tp\ x_1''$ where $x_1''{:}\langle \mathsf{ofcell}\ l\ tp\rangle\ pers \in \Delta$.
- $x_3'{:}\langle \mathsf{ofcell}\ l\ tp'\rangle\ pers \in \Delta$.
- $\Psi; \Delta \vdash N_3 : \mathsf{of}\ v\ tp'\ true$.
- $\Psi; \Delta \vdash N_{3v} : \mathsf{value}\ v\ true$.

By the uniqueness lemma, we have that $x_3' = x_1''$ and $tp' = tp$. Therefore, we can use T' to construct T_2 as follows:

$$
\begin{array}{ll}
T_1 = & \quad (\cdot;\ x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \\
& T' \\
& \quad (\Psi;\ \Theta'\{x'{:}\langle \mathsf{gen}\ tp\rangle\ ord,\ x_3'{:}\langle \mathsf{gencell}\ l\rangle\ eph\}) \\
& \{y_2\} \leftarrow \mathsf{gencell/fulfill}\ l\ tp\ v\ (x_3' \bullet z_1 \bullet !N_3 \bullet !N_{v3}) \\
& \quad (\Psi;\ \Theta'\{x'{:}\langle \mathsf{gen}\ tp\rangle\ ord,\ y_2{:}\langle \mathsf{cell}\ l\ v\rangle\ eph\}) \\
& \{y_1\} \leftarrow \mathsf{gen/retn}\ tp\ v\ (x' \bullet !N_3 \bullet !N_{v3}) \\
& \quad (\Psi;\ \Theta'\{y_1{:}\langle \mathsf{retn}\ v\rangle\ ord,\ y_2{:}\langle \mathsf{cell}\ l\ v\rangle\ eph\})
\end{array}
$$

$(\Psi;\ \Theta\{y_1{:}\langle \mathsf{retn}\ (\mathsf{loc}\ l), y_2{:}\langle \mathsf{cell}\ l\ v\rangle\ eph\rangle\ ord\})$ is the restriction of T_2's output, as required.

The other cases, notably ev/set2, follow the same pattern. □

9.4.4 Revisiting pointer inequality

As we discussed in Section 6.5.1, the fact that SLS variables cannot be directly checked for inequality complicates the representation of languages that can check for the inequality of locations. One way of circumventing this shortcoming is by keeping a runtime counter in the form of an ephemeral atomic proposition count n that counts the number of currently allocated cells; the

```
gen: typ -> dest -> prop lin.

gen/dest: {Exists d:dest. one}.
gen/eval: $gen T D * !of E T >-> {$eval E D}.
gen/retn: $gen T D * !of V T * !value V >-> {$retn V D}.
gen/cont: $gen T D * !off F T' T
           >-> {Exists d'. gen T' d' * $cont F d' D}.
```

Figure 9.8: Generative invariant: destination-passing ("obvious" formulation)

rule ev/ref1 that allocates a new cell must be modified to access and increment this counter and to attach the counter's value to the new cell. Inequality of those natural number tags can then be used as a proxy for inequality of locations.

A generative signature like the one in Figure 9.7 could be used to represent the invariant that each location and each cell is associated with a unique natural number. The techniques described in this chapter should therefore be sufficient to describe generative invariants of SSOS specifications that implement pointer inequality in this way.

9.5 Destination-passing

Destination-passing style specifications, as discussed in Chapter 7, are not a focus of this dissertation, but they deserve mention for two reasons. First, they are of paramount importance in the context of the logical framework CLF, a framework that lacks SLS's notions of order. Second, the work of Cervesato and Sans [CS13] is the most closely related work on describing progress and preservation properties for SSOS-like specifications; their work closely resembles a destination-passing specification. As such, the preservation property given in this section can be viewed an encoding of the proof by Cervesato and Sans in SLS.

In this section, we will work with an operational semantics derived from the signature given in Figure 7.5 (sequential evaluation of function application) in Chapter 7. To use sequential application instead of parallel evaluation of function application, we will need to give different typing rules for frames:

```
off/app1: off app1 Tp (app1 E) (arr Tp' Tp) Tp
           <- of E Tp'.
off/app2: off (app2 \x. E x) Tp' Tp
           <- (All x. of x Tp' -> of (E x) Tp).
```

Other than this change, our deductive typing rules stay the same.

When we move from ordered abstract machines to destination-passing style, the most natural adaptation of generative invariants is arguably the one given in Figure 9.8. In that figure, the core nonterminal is the mobile proposition gen $tp\ d$. The rule gen/dest, which creates destinations

```
gen: typ -> dest -> prop lin.
gendest: dest -> dest -> prop lin.

dest/promise: {Exists d'. $gendest d' D}.
dest/unused:  ($gendest D' D) >-> {one}.

gen/eval: $gen Tp D * !of E Tp >-> {$eval E D}.
gen/retn: $gen Tp D * !of V Tp * !value V >-> {$retn V D}.
gen/cont: $gen Tp D * !off F Tp' Tp * $gendest D' D
           >-> {$gen Tp' D' * $cont F D' D}.
```

Figure 9.9: Generative invariant: destination-passing (modified formulation)

freely, is necessary, as we can see from the following sequence of process states:

$$
\begin{aligned}
(d_0{:}\mathsf{dest};\ x_1{:}\langle \mathsf{eval}\ulcorner(\lambda x.e)\, e_2\urcorner\, d_0\rangle\ eph) &\rightsquigarrow \\
(d_0{:}\mathsf{dest}, d_1{:}\mathsf{dest};\ x_2{:}\langle \mathsf{eval}\ulcorner(\lambda x.e)\urcorner\, d_1\rangle\ eph,\ x_3{:}\langle \mathsf{cont}\,(\mathsf{app1}\ulcorner e_2\urcorner)\, d_1\, d_0\rangle\ eph) &\rightsquigarrow \\
(d_0{:}\mathsf{dest}, d_1{:}\mathsf{dest};\ x_4{:}\langle \mathsf{retn}\ulcorner(\lambda x.e)\urcorner\, d_1\rangle\ eph,\ x_3{:}\langle \mathsf{cont}\,(\mathsf{app1}\ulcorner e_2\urcorner)\, d_1\, d_0\rangle\ eph) &\rightsquigarrow \\
(d_0{:}\mathsf{dest}, d_1{:}\mathsf{dest}, d_2{:}\mathsf{dest};\ x_5{:}\langle \mathsf{eval}\ulcorner e_2\urcorner\, d_2\rangle\ eph,\ x_6{:}\langle \mathsf{cont}\,(\mathsf{app2}\ulcorner\lambda x.e\urcorner)\, d_2\, d_0\rangle\ eph) &\rightsquigarrow \ldots
\end{aligned}
$$

In the final state, d_1 is isolated, no longer mentioned anywhere else in the process state, so gen/dest must be used in the generative trace showing that the last state above is well-typed.

We will not use the form described in Figure 9.8 in this chapter, however. Instead, we will prefer the presentation in Figure 9.9. There are two reasons for this. First and foremost, this formulation meshes better with the promise-then-fulfill pattern that was necessary for state in Figure 9.6 and that is also necessary for continuations in Section 9.6 below. As a secondary consideration, using the first formulation would require us to significantly change the structure of our inversion lemmas. In previous inversion lemmas, proving that gen/cont could always be rotated to the end of a generative trace was simple, because it introduced no LF variables or persistent nonterminals. The gen/cont rule in Figure 9.8 does introduce an LF variable d', invalidating the principle used in Section 9.2.1.

The dest/promise rule in Figure 9.9 is interesting in that it requires each destination d' to be created along with foreknowledge of the destination, d, that the destination d' will return to. This effectively forces all the destinations into a tree structure from the moment of their creation onwards, a point that will become important when we modify Figure 9.9 to account for persistent destinations and first-class continuations. The root of this tree is the destination d_0 that already exists in the initial process state $(d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph)$.

9.5.1 Uniqueness and index sets

One consequence of the way we use the promise-then-fulfill pattern in this specification is that our unique index property becomes conceptually prior to our inversion lemma.

Lemma (Unique indices of $\Sigma_{Gen9.9}$)**.**

1. *If* $T :: (d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph) \leadsto^*_{\Sigma_{Gen9.9}} (\Psi; \Delta)$, $x{:}\langle \mathsf{gendest}\ d\ d_1\rangle\ eph \in \Delta$, *and* $y{:}\langle \mathsf{gendest}\ d\ d_2\rangle\ eph \in \Delta$, *then* $x = y$ *and* $d_1 = d_2$.

2. *If* $T :: (d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph) \leadsto^*_{\Sigma_{Gen9.9}} (\Psi; \Delta)$, $x{:}\langle \mathsf{gendest}\ d\ d_1\rangle\ eph \in \Delta$, *and* $y{:}\langle \mathsf{gen}\ tp\ d\rangle\ eph \in \Delta$, *then there is a contradiction.*

3. *If* $T :: (d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph) \leadsto^*_{\Sigma_{Gen9.9}} (\Psi; \Delta)$, $x{:}\langle \mathsf{gendest}\ d\ d_1\rangle\ eph \in \Delta$, *and* $y{:}\langle \mathsf{cont}\ f\ d\ d'\rangle\ eph \in \Delta$, *then there is a contradiction.*

4. *If* $T :: (d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph) \leadsto^*_{\Sigma_{Gen9.9}} (\Psi; \Delta)$, $x{:}\langle \mathsf{gen}\ tp_1\ d\rangle\ eph \in \Delta$, *and* $y{:}\langle \mathsf{gen}\ tp_2\ d\rangle\ eph \in \Delta$, *then* $x = y$ *and* $tp_1 = tp_2$.

5. *If* $T :: (d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph) \leadsto^*_{\Sigma_{Gen9.9}} (\Psi; \Delta)$, $x{:}\langle \mathsf{cont}\ f_1\ d\ d_1\rangle\ eph \in \Delta$, *and* $y{:}\langle \mathsf{cont}\ f_2\ d\ d_2\rangle\ eph \in \Delta$, *then* $x = y$, $f_1 = f_2$, *and* $d_1 = d_2$.

Proof. Induction and case analysis on last steps of the trace T; each part uses the previous parts (parts 2 and 3 use part 1, and parts 4 and 5 use parts 2 and 3). □

This lemma is a lengthy way of expressing what is ultimately a very simple property: that the second position of gendest is a unique index and that it passes on that unique indexing to the second position of gen and the second position of cont.

Definition 9.5. *A set* S *is a* unique index set *under a generative signature* Σ *and an initial state* $(\Psi; \Delta)$ *if, whenever*

* $\mathsf{a}/i \in S$,
* $\mathsf{b}/j \in S$,
* $(\Psi; \Delta) \leadsto^*_{\Sigma} (\Psi'; \Delta')$,
* $x{:}\langle \mathsf{a}\ t_1 \ldots t_n\rangle\ lvl \in \Delta'$, *and*
* $y{:}\langle \mathsf{b}\ s_1 \ldots s_m\rangle\ lvl' \in \Delta'$,

it is the case that $t_i = s_j$ *implies* $x = y$. *Of course, if* $x = y$, *that in turn implies that* $\mathsf{a} = \mathsf{b}$, $i = j$, $n = m$, $t_k = s_k$ *for* $1 \leq k \leq n$, *and* $lvl = lvl'$.

The complicated lemma above can then be captured by the dramatically more concise statement: $\{\mathsf{gendest}/1, \mathsf{gen}/2\}$ and $\{\mathsf{gendest}/1, \mathsf{cont}/2\}$ are both unique index sets under the signature $\Sigma_{Gen9.9}$ and the initial state $(d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph)$. In fact, we can extend the first unique index set to $\{\mathsf{gendest}/1, \mathsf{gen}/2, \mathsf{eval}/2, \mathsf{retn}/2\}$. Stating that $\{\mathsf{gendest}/1, \mathsf{gen}/2\}$ was a unique index property previously required 3 distinct statements, and it would take 10 distinct statements to express that $\{\mathsf{gendest}/1, \mathsf{gen}/2, \mathsf{eval}/2, \mathsf{retn}/2\}$ is a unique index property.[10] The

[10] Four positive statements (similar to parts 1, 4, and 5 of the lemma above) along $\binom{4}{2} = 6$ negative ones (similar to parts 2 and 3 of the lemma above).

unique index property for cells (Section 9.4.2) can be rephrased by saying that $\{\mathsf{ofcell}/1\}$ is a unique index set; $\{\mathsf{gencell}/1, \mathsf{cell}/1\}$ is also a unique index set in that specification.

It's also possible for unique index sets to be simply (and, presumably, mechanically) checked. This amounts to a very simple preservation property.

9.5.2 Inversion

Lemma (Inversion – Figure 9.9)**.**

1. *If* $T :: (d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph) \leadsto^*_{\Sigma_{Gen9.9}} (\Psi;\ \Theta\{y{:}\langle \mathsf{eval}\ e\ d\rangle\ eph\})$,
   then $T = (T'; \{y\} \leftarrow \mathsf{gen}/\mathsf{eval}\ tp\ d\ e\ (x' \bullet !N))$,
   where $\Psi; \Delta \vdash N : \mathsf{of}\ e\ tp\ true$,
   $T' :: (d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph) \leadsto^*_{\Sigma_{Gen9.9}} (\Psi;\ \Theta\{x'{:}\langle \mathsf{gen}\ tp\ d\rangle\ eph\})$,
   and Δ *is the persistent part of* $\Theta\{x'{:}\langle \mathsf{gen}\ tp\ d\rangle\ eph\}$.

2. *If* $T :: (d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph) \leadsto^*_{\Sigma_{Gen9.9}} (\Psi;\ \Theta\{y{:}\langle \mathsf{retn}\ v\ d\rangle\ eph\})$,
   then $T = (T'; \{y\} \leftarrow \mathsf{gen}/\mathsf{retn}\ tp\ d\ v\ (x' \bullet !N \bullet !N_v))$,
   where $\Psi; \Delta \vdash N : \mathsf{of}\ e\ tp\ true$, $\Psi; \Delta \vdash N_v : \mathsf{value}\ v\ true$,
   $T' :: (d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph) \leadsto^*_{\Sigma_{Gen9.9}} (\Psi;\ \Theta\{x'{:}\langle \mathsf{gen}\ tp\ d\rangle\ eph\})$,
   and Δ *is the persistent part of* $\Theta\{x'{:}\langle \mathsf{gen}\ tp\ d\rangle\ eph\}$.

3. *If* $T :: (d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph)$
   $\leadsto^*_{\Sigma_{Gen9.9}} (\Psi;\ \Theta\{y_1{:}\langle \mathsf{gen}\ tp'\ d'\rangle\ eph,\ y_2{:}\langle \mathsf{cont}\ f\ d'\ d\rangle\ eph\})$,
   then $T = (T'; \{y_1, y_2\} \leftarrow \mathsf{gen}/\mathsf{cont}\ tp\ d\ f\ tp'\ d'\ (x' \bullet !N \bullet z))$,
   where $\Psi; \Delta \vdash N : \mathsf{off}\ e\ tp'\ tp\ true$,
   $T' :: (d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph)$
   $\leadsto^*_{\Sigma_{Gen9.9}} (\Psi;\ \Theta\{x'{:}\langle \mathsf{gen}\ tp\ d\rangle\ eph,\ z{:}\langle \mathsf{gendest}\ d'\ d\rangle\ eph\})$,
   and Δ *is the persistent part of* $\Theta\{x'{:}\langle \mathsf{gen}\ tp\ d\rangle\ eph,\ z{:}\langle \mathsf{gendest}\ d'\ d\rangle\ eph\}$.

Proof. As with other inversion lemmas, each case follows by induction and case analysis on the last steps of T. The trace cannot be empty, so $T = T''; S$ for some T'' and S, and we let Var be the set of relevant variables $\{y\}$ in parts 1 and 2, and $\{y_1, y_2\}$ in part 3.

If $\emptyset = S^\bullet \cap Var$, the proof proceeds by induction as it did in Section 9.2.1.

If $S^\bullet \cap Var$ is nonempty, then we must again show by case analysis that $S^\bullet = Var$ and that furthermore S is the step we were looking for. As before, this is easy for the unary grammar productions where Var is a singleton set: there is only one rule that can produce an atomic proposition $\mathsf{eval}\ e\ d$ or $\mathsf{retn}\ v\ d$.

When Var is not a singleton (which only happens in part 3 for this lemma), we must use the unique index property to reason that if there is any overlap, that overlap must be total.

- Say $S = \{y_1, y_2''\} \leftarrow \mathsf{gen}/\mathsf{cont}\ tp\ d''\ f''\ tp'\ d'\ (x' \bullet !N \bullet z)$.
  Then the final state contains $y_2{:}\langle \mathsf{cont}\ f\ d'\ d\rangle\ eph$ and $y_2''{:}\langle \mathsf{cont}\ f''\ d'\ d''\rangle\ eph$. The shared d' and the unique index property ensures that $y_2 = y_2''$, $f = f''$, and $d = d''$.

* Say $S = \{y_1'', y_2\} \leftarrow \mathsf{gen/cont}\ tp\ d\ f\ tp'''\ d'\ (x' \bullet !N \bullet z)$.
  Then the final state contains $y_1{:}\langle \mathsf{gen}\ tp'\ d' \rangle\ eph$ and $y_1''{:}\langle \mathsf{gen}\ tp'''\ d' \rangle\ eph$. The shared d' and the unique index property ensures that $y_1 = y_1''$ and $tp' = tp'''$.

Therefore, $S = \{y_1, y_2\} \leftarrow \mathsf{gen/cont}\ tp\ d\ f\ tp'\ d'\ (x' \bullet !N \bullet z)$. □

9.5.3 Preservation

As we once again have no persistent nonterminals, we can return to the simpler form of the preservation theorem used in Theorem 9.2 and Theorem 9.3 (compared to the more complex formulation needed for Theorem 9.4).

Theorem 9.6 ($\Sigma_{Gen9.9}$ is a generative invariant)**.** *If* $T_1 :: (d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{gen}\ tp_0\ d_0 \rangle\ eph) \leadsto^*_{\Sigma_{Gen9.9}} (\Psi; \Delta)$ *and there is a step* $S :: (\Psi; \Delta) \lightning \leadsto (\Psi'; \Delta')$ *under the signature from Figure 7.5, then* $T_2 :: (d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{gen}\ tp_0\ d_0 \rangle\ eph) \leadsto^*_{\Sigma_{Gen9.9}} (\Psi'; \Delta')$.

$$
\begin{array}{ccc}
(d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{gen}\ tp_0\ d_0 \rangle\ eph) & & (d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{gen}\ tp_0\ d_0 \rangle\ eph) \\
\Sigma_{Gen9.9} \downarrow^* & & \Sigma_{Gen9.9} \downarrow^* \\
(\Psi; \Delta) & & (\Psi'; \Delta') \\
\lightning & & \lightning \\
(\Psi; \Delta) & \leadsto & (\Psi'; \Delta')
\end{array}
$$

Proof. As usual, we enumerate the synthetic transitions possible under the signature in Figure 7.5, perform inversion on the structure of T_1, and then use the results of inversion to construct T_2. We give one illustrative case.

Case $\{d_2, y_1, y_2\} \leftarrow \mathsf{ev/app1}\ (\lambda x.e)\ d_1\ e_2\ d\ (x_1 \bullet x_2)$
$:: (\Psi;\ \Theta\{x_1{:}\langle \mathsf{retn}\ (\mathsf{lam}\ \lambda x.e)\ d_1 \rangle\ eph,\ x_2{:}\langle \mathsf{cont}\ (\mathsf{app1}\ e_2)\ d_1\ d \rangle\ eph\})$
$\leadsto (\Psi, d_2{:}\mathsf{dest};\ \Theta\{y_1{:}\langle \mathsf{eval}\ e_2\ d_2 \rangle\ eph,\ y_2{:}\langle \mathsf{cont}\ (\mathsf{app2}\ \lambda x.e)\ d_2\ d \rangle\ eph\})$

Applying inversion (Part 2, then Part 3) to T_1, we have

$$
\begin{array}{ll}
T_1 = & (d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{gen}\ tp_0\ d_0 \rangle\ eph) \\
T' & \\
& (\Psi;\ \Theta\{x'{:}\langle \mathsf{gen}\ tp\ d \rangle\ eph,\ z_1{:}\langle \mathsf{gendest}\ d_1\ d \rangle\ eph\}) \\
& \{x_1', x_2\} \leftarrow \mathsf{gen/cont}\ tp\ d\ (\mathsf{app1}\ e_2)\ tp'\ d_1\ (x' \bullet !N_2 \bullet z_1) \\
& (\Psi;\ \Theta\{x_1'{:}\langle \mathsf{gen}\ tp'\ d_1 \rangle\ eph,\ x_2{:}\langle \mathsf{cont}\ (\mathsf{app1}\ e_2)\ d_1\ d \rangle\ eph\}) \\
& \{x_1\} \leftarrow \mathsf{gen/retn}\ tp'\ d_1\ v\ (x_1' \bullet !N_1 \bullet !N_{v1}) \\
& (\Psi;\ \Theta\{x_1{:}\langle \mathsf{retn}\ (\mathsf{lam}\ \lambda x.e)\ d_1 \rangle\ eph,\ x_2{:}\langle \mathsf{cont}\ (\mathsf{app1}\ e_2)\ d_1\ d \rangle\ eph\})
\end{array}
$$

where Δ contains the persistent propositions from Θ and where

- $\Psi; \Delta \vdash N_2 : \mathsf{off}\ (\mathsf{app1}\ e_2)\ tp'\ tp\ true$. By traditional inversion we know there exists tp'' and N_2' such that $tp' = \mathsf{arr}\ tp''\ tp$ and $\Psi; \Delta \vdash N_2' : \mathsf{of}\ e_2\ tp''\ true$.
- $\Psi; \Delta \vdash N_1 : \mathsf{of}\ (\mathsf{lam}\ \lambda x.e)\ \mathsf{arr}\ tp'\ tp\ true$. By traditional inversion we know there exists N_1' where $\Psi, x{:}\mathsf{exp}; \Delta, dx : (\mathsf{of}\ x\ tp')\ pers \vdash N_1' : \mathsf{of}\ e\ tp\ true$.

We can use T' to construct T_2 as follows:

```
offuture: exp -> typ -> prop pers.
genfuture: dest -> exp -> prop lin.

of/future: of X Tp <- offuture X Tp.

future/promise: {Exists d. Exists x. $genfuture d x * !offuture x Tp}.

future/compute: $genfuture D X * !offuture X Tp
                 >-> {$gen Tp D * $promise D X}.

future/bind:    $genfuture D X * !offuture X Tp * !of V Tp * !value V
                 >-> {!bind X V}.
```

Figure 9.10: Generative invariant: futures

$$
\begin{array}{ll}
T_2 = & (d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph) \\
T' & \\
& (\Psi;\ \Theta\{x'{:}\langle \mathsf{gen}\ tp\ d\rangle\ eph,\ z_1{:}\langle \mathsf{gendest}\ d_1\ d\rangle\ eph\}) \\
& \{\} \leftarrow \mathsf{dest/unused}\ d_1\ d\ z_1 \\
& \{d_2, z_2\} \leftarrow \mathsf{dest/promise}\ d \\
& (\Psi, d_2{:}\mathsf{dest};\ \Theta\{x'{:}\langle \mathsf{gen}\ tp\ d\rangle\ eph,\ z_2{:}\langle \mathsf{gendest}\ d_2\ d\rangle\ eph\}) \\
& \{y_1', y_2\} \leftarrow \mathsf{gen/cont}\ tp\ d\ (\mathsf{app2}\lambda x.e)\ tp''\ d_2 \\
& \qquad\qquad (x' \bullet !(\mathsf{off/app2}\ tp''\ (\lambda x.e)\ tp\ (\lambda x, dx.\, !N_1')) \bullet z_2) \\
& (\Psi, d_2{:}\mathsf{dest};\ \Theta\{y_1'{:}\langle \mathsf{gen}\ tp''\ d_2\rangle\ eph,\ y_2{:}\langle \mathsf{cont}\ (\mathsf{app2}\ \lambda x.e)\ d_2\ d\rangle\ eph\}) \\
& \{y_1, y_2\} \leftarrow \mathsf{gen/eval}\ tp''\ d_2\ e_2\ (y_1' \bullet !N_2') \\
& (\Psi, d_2{:}\mathsf{dest};\ \Theta\{y_1{:}\langle \mathsf{eval}\ e_2\ d_2\rangle\ eph,\ y_2{:}\langle \mathsf{cont}\ (\mathsf{app2}\ \lambda x.e)\ d_2\ d\rangle\ eph\})
\end{array}
$$

The other cases follow the same pattern. □

9.5.4 Extensions

Generative invariants for parallel evaluation (Figure 7.6) and the alternate semantics of parallelism and failure (Figure 7.7) as described in Section 7.2.1 are straightforward extensions of the development in this section. Synchronization (Section 7.2.2) and futures (Section 7.2.3) are a bit more interesting from the perspective of generative invariants and preservation. Figure 9.10 is one proposal for a generative invariant for our SLS encoding of futures, but we leave further consideration for future work.

9.6 Persistent continuations

The final specification style we will cover in detail is the use of persistent continuations as discussed in Section 7.2.4 as a way of giving an SSOS semantics for first-class continuations (Figure 7.11). The two critical rules from Figure 7.11 are repeated below: ev/letcc captures the

```
gen: prop lin.
ofdest: dest -> typ -> prop pers.
gendest: dest -> dest -> prop lin.

value/contn: value (contn D).

of/letcc: of (letcc \x. E x) Tp
           <- (All x. of x (conttp Tp) -> of (E x) Tp).
of/contn: of (contn D) (conttp Tp)
           <- ofdest D Tp.
of/throw: of (throw E1 E2) Tp'
           <- of E1 Tp
           <- of E2 (conttp Tp).

off/throw1: off (throw1 E2) Tp Tpx
             <- of E2 (conttp Tp).
off/throw2: off (throw2 V1) (conttp Tp) Tpx
             <- of V1 Tp
             <- value V1.

dest/promise: {Exists d'. $gendest d' D * !ofdest d' Tp'}.
dest/fulfill: $gendest D' D *
              !off F Tp' Tp * !ofdest D' Tp' * !ofdest D Tp
               >-> {!cont F D' D}.

gen/eval: $gen * !ofdest D Tp * !of E Tp >-> {eval E D}.
gen/retn: $gen * !ofdest D Tp * !of V Tp * !value V >-> {retn V D}.
```

Figure 9.11: Generative invariant: persistent destinations and first-class continuations

destination representing the current continuation and the rule ev/throw2 throws away the continuation represented by d_2 and throws computation to the continuation represented by the dk.

```
ev/letcc:  $eval (letcc \x. E x) D >-> {$eval (E (contn D)) D}.
ev/throw2: $retn (contn DK) D2 * !cont (throw2 V1) D2 D
            >-> {$retn V1 DK}.
```

While the setup of Figure 9.9 is designed to make the transition to persistent continuations and letcc seem less unusual, this section still represents a radical shift.

It should not be terribly surprising that the generative invariant for persistent continuations is rather different than the other generative invariants. Generative invariants capture patterns of specification, and we have mostly concentrated on patterns that facilitate concurrency and communication. Persistent continuations, on the other hand, are a pattern mostly associated with first-class continuations. There is not an obvious way to integrate continuations and parallel or concurrent evaluation, and the proposal by Moreau and Ribbens in [MR96] is not straightforward to adapt to the semantic specifications we gave in Chapter 7.

Consider again the gendest/promise rule from Figure 9.9. The rule consumes no nonterminals and is the only rule introducing LF variables, so any $T :: (d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{gen}\ tp_0\ d_0\rangle\ eph) \rightsquigarrow^* (\Psi; \Delta)$ under $\Sigma_{Gen9.9}$ can be factored into two parts $T = T_1; T_2$ where T_1 contains only steps that use gendest/promise. The computational effect of Theorem 9.6 is that T_1 grows to track the tree-structured shape of the stack, both past and present. We could record, if we wanted to, the *past* structure of the control stack by adding a persistent nonterminal ghostcont $f\ d'\ d$ and modifying dest/unused in Figure 9.9 as follows:

```
dest/unused:  $gendest D' D >-> {!ghostcont F D' D}.
```

Once we make the move to persistent continuations, however, there's no need to create a ghost continuation, we can just have the rule dest/unused (renamed to dest/fulfill in Figure 9.11) create the continuation itself. To make this work, dest/promise predicts the type that will be associated with a newly-generated destination d by generating a persistent nonterminal ofdest $d\ tp$. (This is just like how gencell/promise in Figure 9.6 predicts the type of a location l by generating a persistent nonterminal ofcell $l\ tp$.) Then, dest/fulfill checks to make sure that the generated continuation frame has the right type relative to the destinations it connects.

Taken together, the rules dest/promise and dest/fulfill rules in Figure 9.11 create a tree-structured map of destinations starting from an initial destination d_0 and an initial persistent atomic proposition ofdest $d_0\ tp_0$, and the dest/fulfill rule ensures that every destination on this map encodes a specific and well-typed stack of frames that can be read off by following destinations back to the root d_0. The initial ofdest proposition takes over for the mobile proposition gen $tp_0\ d_0$ that formed the root of our tree in all previous specifications. The mobile gen nonterminal no longer needs indices, and just serves to place a single eval or retn somewhere on the well-typed map of destinations. The initial state of our generative traces is therefore $(d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{ofdest}\ d_0\ tp_0\rangle\ pers, z{:}\langle \mathsf{gen}\rangle\ eph)$; this is reflected in the the preservation theorem.

Lemma (Unique indices of $\Sigma_{Gen9.11}$)**.** *Both* $\{\mathsf{ofdest}/1\}$ *and* $\{\mathsf{gendest}/1, \mathsf{cont}/2\}$ *are unique index sets under the initial state* $(d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{ofdest}\ d_0\ tp_0\rangle\ pers, z{:}\langle \mathsf{gen}\rangle\ eph)$ *and the signature* $\Sigma_{Gen9.11}$.

Proof. Induction and case analysis on the last steps of a given trace. □

Lemma (Inversion – Figure 9.11)**.**

1. *If* $T :: (d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{ofdest}\ d_0\ tp_0\rangle\ pers,\ z{:}\langle \mathsf{gen}\rangle\ eph) \rightsquigarrow^*_{\Sigma_{Gen9.11}} (\Psi;\ \Theta\{y{:}\langle \mathsf{eval}\ e\ d\rangle\ eph\})$,
   then $T = (T'; \{y\} \leftarrow \mathsf{gen/eval}\ d\ tp\ e\ (z' \bullet x \bullet !N))$,
   where $x{:}\langle \mathsf{ofdest}\ d\ tp\rangle\ pers \in \Delta$, $\Psi; \Delta \vdash N : \mathsf{of}\ e\ tp\ true$,
   $T' :: (d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{ofdest}\ d_0\ tp_0\rangle\ pers,\ z{:}\langle \mathsf{gen}\rangle\ eph) \rightsquigarrow^*_{\Sigma_{Gen9.11}} (\Psi;\ \Theta\{z'{:}\langle \mathsf{gen}\rangle\ eph\})$,
   and Δ *is the persistent part of* $(\Psi;\ \Theta\{z'{:}\langle \mathsf{gen}\rangle\ eph\})$.

2. *If* $T :: (d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{ofdest}\ d_0\ tp_0\rangle\ pers,\ z{:}\langle \mathsf{gen}\rangle\ eph) \rightsquigarrow^*_{\Sigma_{Gen9.11}} (\Psi;\ \Theta\{y{:}\langle \mathsf{retn}\ v\ d\rangle\ eph\})$,
   then $T = (T'; \{y\} \leftarrow \mathsf{gen/retn}\ d\ tp\ v\ (z' \bullet x \bullet !N \bullet !N_v))$,
   where $x{:}\langle \mathsf{ofdest}\ d\ tp\rangle\ pers \in \Delta$, $\Psi; \Delta \vdash N : \mathsf{of}\ v\ tp\ true$, $\Psi; \Delta \vdash N : \mathsf{value}\ v\ true$,
   $T' :: (d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{ofdest}\ d_0\ tp_0\rangle\ pers,\ z{:}\langle \mathsf{gen}\rangle\ eph) \rightsquigarrow^*_{\Sigma_{Gen9.11}} (\Psi;\ \Theta\{z'{:}\langle \mathsf{gen}\rangle\ eph\})$,
   and Δ *is the persistent part of* $(\Psi;\ \Theta\{z'{:}\langle \mathsf{gen}\rangle\ eph\})$.

3. *If* $T :: (d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{ofdest}\ d_0\ tp_0\rangle\ pers,\ z{:}\langle \mathsf{gen}\rangle\ eph)$
$\qquad \rightsquigarrow^*_{\Sigma_{Gen9.11}} (\Psi;\ \Theta\{y{:}\langle \mathsf{cont}\ f\ d'\ d\rangle\ pers\})$,
then $T = (T'; \{y\} \leftarrow \mathsf{dest/fulfill}\ d'\ d\ f\ tp'\ tp\ (y' \bullet !N \bullet x' \bullet x))$,
where $x'{:}\langle \mathsf{ofdest}\ d'\ tp'\rangle\ pers \in \Delta$, $x{:}\langle \mathsf{ofdest}\ d\ tp\rangle\ pers \in \Delta$, $\Psi; \Delta \vdash N : \mathsf{off}\ f\ tp'\ tp\ true$
$T' :: (d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{ofdest}\ d_0\ tp_0\rangle\ pers, z{:}\langle \mathsf{gen}\rangle\ eph)$
$\qquad \rightsquigarrow^*_{\Sigma_{Gen9.11}} (\Psi;\ \Theta\{y'{:}\langle \mathsf{gendest}\ d'\ d\rangle\ eph\})$,
and Δ *is the persistent part of* $(\Psi;\ \Theta\{z'{:}\langle \mathsf{gendest}\ d'\ d\rangle\ eph\})$.

Proof. Induction and case analysis on last steps of the trace T; each case is individually quite simple because Var is always a singleton $\{y\}$. While we introduce a persistent atomic proposition cont in part 3, the step that introduces this proposition can always be rotated to the end of a trace because cont propositions cannot appear in the input interface of any step in under the generative signature $\Sigma_{Gen9.11}$. This is a specific property of $\Sigma_{Gen9.11}$, but it also follows from the definition of generative signatures (Definition 9.1), which stipulates that transitions enabled by a generative signature cannot consume or mention terminals like cont.

As an aside, z will always equal z' in parts 1 and 2, but we'll never need to rely on this fact. □

Theorem 9.7 ($\Sigma_{Gen9.11}$ is a generative invariant)**.**

If $T_1 :: (d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{ofdest}\ d_0\ tp_0\rangle\ pers, z{:}\langle \mathsf{gen}\rangle\ eph) \rightsquigarrow^*_{\Sigma_{Gen9.11}} (\Psi; \Delta)$ *and* $S :: (\Psi; \Delta)\lightning \rightsquigarrow (\Psi'; \Delta')$ *under the signature from Figure 7.2.4, then* $(\Psi'; \Delta') = (\Psi'; \Delta'')\lightning$ *for some* Δ'' *such that* $T_2 :: (d_0{:}\mathsf{dest}; x_0{:}\langle \mathsf{ofdest}\ d_0\ tp_0\rangle\ pers, z{:}\langle \mathsf{gen}\rangle\ eph) \rightsquigarrow^*_{\Sigma_{Gen9.11}} (\Psi'; \Delta'')$.

$$\begin{tikzcd}
(d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{ofdest}\ d_0\ tp_0\rangle\ pers,\ z{:}\langle \mathsf{gen}\rangle\ eph) \arrow[d, squiggly, "\Sigma_{Gen9.11}"'] & (d_0{:}\mathsf{dest};\ x_0{:}\langle \mathsf{ofdest}\ d_0\ tp_0\rangle\ pers,\ z{:}\langle \mathsf{gen}\rangle\ eph) \arrow[d, squiggly, "\Sigma_{Gen9.11}"'] \\
(\Psi; \Delta) \arrow[d, slashed] & (\Psi'; \Delta'') \arrow[d, slashed] \\
(\Psi; \Delta)\lightning \arrow[r, squiggly] & (\Psi'; \Delta')
\end{tikzcd}$$

Proof. As always, the proof proceeds by enumeration, inversion, and reconstruction; the cases are all fundamentally similar to the ones we have already seen. □

9.7 On mechanization

In this chapter, we have shown that generative invariants can describe well-formedness and well-typedness properties of the full range of specifications discussed in Part II of this dissertation. We have furthermore shown that these generative invariants are a suitable basis for reasoning about type preservation in these specifications. All of these proofs have a common 3-step structure:

1. Straightforward unique index properties,
2. An inversion lemma that mimics the structure of the generative signature, and

3. A preservation theorem that proceeds by enumerating transitions, applying inversion to the given generative trace, and using the result to construct a new generative trace.

Despite the fact that the inversion lemmas in this chapter technically use induction, they do so in such a trivial way that is quite possible to imagine that inversion lemmas could be automatically synthesized from a generative signature. Unique index properties may be less straightforward to synthesize, but like termination and mode properties in Twelf, they should be entirely straightforward to verify. Only the last part of step 3, the reconstruction that happens in a preservation theorem, has the structure of a more general theorem proving task. Therefore, there is reason to hope that we can mechanize the tedious results in the results in this chapter in a framework that does much of the work of steps 1 and 2 automatically.

Chapter 10

Safety for substructural specifications

In Chapter 9, we showed how the preservation theorem could be established for a wide variety of SSOS semantics, both ordered abstract machines and destination-passing style semantics. The methodology of generative invariants we espoused goes significantly beyond previous work on type preservation for operational semantics specifications in substructural logic. Neither Linear LF encodings by Pfenning, Cervesato, and Reed [CP02, Ree09a], nor the Ordered LF encodings of Felty and Momigliano [FM12], discussed preservation for concurrent specifications or for first-class continuations.

More fundamentally, however, this previous work does not even provide a *language* for talking about progress theorems, the other critical companion of type safety theorems. These previous approaches were universally based on complete derivations. These complete derivations have the flavor of derivations in a big-step semantics, and it is difficult or even impossible to talk about progress for such specifications. The purpose of this chapter is to establish that the SLS framework's traces T and steps S, which correspond to partial proofs, provide a suitable basis for stating progress theorems (and therefore language safety theorems) and for proving these theorems.

We do not discuss progress and safety for the full range of specifications from Part II or Chapter 9, however. Instead, we will just discuss progress for two examples: the ordered abstract machine specification with parallelism and failure used as an example in Figure 9.1, and the extension of this specification with mutable storage. The rest is left for future work, though we claim that these two examples serve to get across all the concepts necessary to prove progress theorems for SSOS specifications. Ultimately, it is not only possible to prove progress and safety theorems using SSOS specifications in SLS; it's also reasonably straightforward.

10.1 Backwards and forwards through traces

In the last chapter, we worked on traces exclusively by induction and case analysis on the *last* steps of a generative trace. This form of case analysis and induction on the last steps of a trace can also be used to prove progress for sequential SSOS specifications, and it is actually necessary to prove progress for SSOS specifications with first-class continuations (discussed in Section 7.2.4 and Section 9.6) in this way, though we leave the details of this argument as future

work. However, for the ordered abstract machine example from Figure 9.1, the other direction is more natural: we work by induction and case analysis on the *first* steps of a generative trace. The branching structure introduced by parallel continuation frames (that is, ordered propositions $\mathsf{cont2}\ f$) is what makes it more natural to work from the beginning of a generative trace, rather than the end.

The proof of progress relies critically on one novel property: that transitions in the generative trace do not tamper with terminals. Formally, we need to know that if $\Theta\{\Delta\} \rightsquigarrow^*_{\Sigma Gen} \Delta'$ under some generative signature ΣGen and if Δ contains only terminals, then there is some Θ' such that the final state Δ' matches $\Theta'\{\Delta\}$. We will implicitly use this property in most of the cases of the progress theorem below.

This property holds for all the generative signatures in Chapter 9, but establishing this property for generative signatures *in general* necessitates a further restriction of what counts as a generative signature (Definition 9.1). To see why, let $\Delta = (x_1{:}\langle \mathsf{retn}\ v\rangle\ ord, x_2{:}\langle \mathsf{cont}\ f\rangle\ ord)$ and consider the generative rule $\forall e.\{\mathsf{eval}\ e\}$, which is allowed under Definition 9.1. This rule could "break" the context by dropping an ordered $\mathsf{eval}\ e$ proposition in between x_1 and x_2. A sufficient general condition for avoiding this problem is to demand that any generative rule that produces ordered atomic propositions mentions an ordered nonterminal as a premise. (This is related to the property called *separation* in [SP08].)

10.2 Progress for ordered abstract machines

The progress property is that, if $T :: (x_0{:}\langle \mathsf{gen}\ tp_0\rangle\ ord) \rightsquigarrow^*_{\Sigma_{Gen9.4}} \Delta$ and $\Delta\lightning$, then one of these three possibilities hold:

1. $\Delta \rightsquigarrow \Delta'$ under the signature from Figure 9.1,
2. $\Delta = y{:}\langle \mathsf{retn}\ v\rangle\ ord$, where v is a value, or
3. $\Delta = y{:}\langle \mathsf{error}\rangle\ ord$.

This is exactly the form of a traditional progress theorem: if a process state is well typed, it either takes a step under the dynamic semantics or is a final state (terminating with an error or returning a value).

The presence of parallel evaluation in Figure 9.1 necessitates that we generalize our induction hypothesis. The statement above is a straightforward corollary of Theorem 10.1 below.

Theorem 10.1 (Progress for ordered abstract machines)**.** *If* $T :: \Theta\{x{:}\langle \mathsf{gen}\ tp\rangle\ ord\} \rightsquigarrow_{\Sigma_{Gen9.4}} \Delta$ *and* $\Delta\lightning$*, then either*

- $\Delta \rightsquigarrow \Delta'$ *under the signature from Figure 9.1 for some* Δ'*, or else*
- $T = (T_1; \{y\} \leftarrow \mathsf{gen}/\mathsf{retn}\ tp\ v\ x; T_2)$ *and* $\cdot \vdash N : \mathsf{value}\ v\ true$*, or else*
- $T = (T_1; \{y\} \leftarrow \mathsf{gen}/\mathsf{error}\ tp\ x; T_2)$.

In the proof of Theorem 10.1, we will not detail the parts of the proof that already arise in traditional proofs of progress for abstract machines. These missing details can be factored into two lemmas. The first lemma is that if $\cdot \vdash N : \mathsf{of}\ e\ tp\ true$, then the process state $\Theta\{x{:}\langle \mathsf{eval}\ e\rangle\ ord\}$ can always take a step; this lemma justifies the classification of eval as an *active* proposition as

described in Section 7.2.2 and in [PS09]. The second lemma is traditionally called a *canonical forms* lemma: it verifies, by case analysis on the structure of typing derivations and values, that well-typed values returned to a well-typed frames can always take a step.

Proof. By induction and case analysis on the first steps of T. We cannot have $T = \diamond$, because we cannot apply restriction to a context containing the nonterminal gen tp. So $T = S; T'$, and either $x \notin {}^\bullet S$ or $x \in {}^\bullet S$.

If $x \notin {}^\bullet S$, then $T' :: \Theta'\{x{:}\langle \mathsf{gen}\ tp\rangle\ ord\} \rightsquigarrow_{\Sigma_{Gen9.4}} \Delta$ and we can succeed by immediate appeal to the induction hypothesis.

If $x \in {}^\bullet S$, then we perform case analysis on the possible transitions enabled by $\Sigma_{Gen9.4}$:

* $S = \{y\} \leftarrow \mathsf{gen/eval}\ e\ tp\ (x \bullet !N)$ where $\cdot \vdash N : \mathsf{of}\ e\ tp\ true$.
  Because eval is a terminal, $\Delta = \Theta'\{y{:}\langle \mathsf{eval}\ e\rangle\ ord\}$, and we proceed by case analysis on N to show that the derivation can always take a step (eval is an active proposition).

* $S = \{y\} \leftarrow \mathsf{gen/retn}\ tp\ v\ (x \bullet !N \bullet !N_v)$ – succeed immediately.

* $S = \{y\} \leftarrow \mathsf{gen/error}\ tp\ x$ – succeed immediately.

* $S = \{y_1', y_2\} \leftarrow \mathsf{gen/cont}\ tp\ f\ tp'\ (x \bullet !N)$ where $\cdot \vdash N : \mathsf{off}\ f\ tp'\ tp\ true$.
  Invoke the i.h. on $T' : \Theta\{y_1'{:}\langle \mathsf{gen}\ tp'\rangle\ ord,\ y_2{:}\langle \mathsf{cont}\ f\rangle\ ord\} \rightsquigarrow_{\Sigma_{Gen9.4}} \Delta$, and then perform case analysis on the result to prove that $\Delta \rightsquigarrow \Delta'$:
  - If $\Delta \rightsquigarrow \Delta'$, then we're done.
  - If $T' = (T_1'; \{y_1\} \leftarrow \mathsf{gen/retn}\ tp'\ v\ (y_1' \bullet !N' \bullet !N_v'); T_2')$,
    then because retn and cont are terminals, $\Delta = \Theta'\{y_1{:}\langle \mathsf{retn}\ v\rangle\ ord,\ y_2{:}\langle \mathsf{cont}\ f\rangle\ ord\}$, and we proceed by simultaneous case analysis on N, N', and N_v' (canonical forms lemma).
  - If $T' = (T_1'; \{y_1\} \leftarrow \mathsf{gen/error}\ tp'\ y_1'; T_2')$,
    then because error and cont are terminals, $\Delta = \Theta'\{y_1{:}\langle \mathsf{error}\rangle\ ord,\ y_2{:}\langle \mathsf{cont}\ f\rangle\ ord\}$, and we have $\{z\} \leftarrow \mathsf{ev/error}\ f\ (y_1 \bullet y_2) :: \Delta \rightsquigarrow \Theta'\{z{:}\langle \mathsf{error}\rangle\ ord\}$.

* $S = \{y_1', y_2', y_3\} \leftarrow \mathsf{gen/cont2}\ tp\ f\ tp_1\ tp_2\ (x \bullet !N)$ where $\cdot \vdash N : \mathsf{off2}\ f\ tp_1\ tp_2\ tp\ true$.
  Invoke the i.h. twice on $T' : \Theta\{y_1'{:}\langle \mathsf{gen}\ tp_1\rangle\ ord,\ y_2'{:}\langle \mathsf{gen}\ tp_2\rangle\ ord,\ y_3{:}\langle \mathsf{cont2}\ f\rangle\ ord\}$, once to see what happens to y_1', and another time to see what happens to y_2', and then perform case analysis on the result to prove that $\Delta \rightsquigarrow \Delta'$:
  - If either invocation returns the first disjunctive possibility, that $\Delta \rightsquigarrow \Delta'$, then we're done.
  - If both invocations return the second disjunctive possibility, then T' contain two steps $\{y_1\} \leftarrow \mathsf{gen/retn}\ tp_1\ v_1\ (y_1' \bullet !N_1 \bullet !N_{v1})$ and $\{y_2\} \leftarrow \mathsf{gen/retn}\ tp_2\ v_2\ (y_2' \bullet !N_2 \bullet !N_{v2})$. Because retn and cont2 are terminals, $\Delta = \Theta'\{y_1{:}\langle \mathsf{retn}\ v_1\rangle\ ord,\ y_2{:}\langle \mathsf{retn}\ v_2\rangle\ ord,\ y_3{:}\langle \mathsf{cont2}\ f\rangle\ ord\}$, and we proceed by simultaneous case analysis on N, N_1, N_{v1}, N_2, and N_{v2} (canonical forms lemma).
  - In all the remaining cases, one of the subcomputations becomes an error and the other one becomes another error or a returned value. In any of these cases, $\Delta \rightsquigarrow \Delta'$ by one of the rules ev/errret, ev/reterr, or by ev/ererr.

* $S = \{y_1', y_2\} \leftarrow \mathsf{gen/handle}\ tp\ e_2\ (x \bullet !N)$.
  Invoke the i.h. on $T' : \Theta\{y_1'{:}\langle \mathsf{gen}\ tp'\rangle\ ord,\ y_2{:}\langle \mathsf{handle}\ e_2\rangle\ ord\} \rightsquigarrow_{\Sigma_{Gen9.4}} \Delta$, and then perform case analysis on the result to prove that $\Delta \rightsquigarrow \Delta'$:
  - If $\Delta \rightsquigarrow \Delta'$, then we're done.
  - If $T' = (T_1'; \{y_1\} \leftarrow \mathsf{gen/retn}\ tp'\ v\ (y_1' \bullet !N' \bullet !N_v'); T_2')$,
    then because retn and cont are terminals, $\Delta = \Theta'\{y_1{:}\langle \mathsf{retn}\ v\rangle\ ord,\ y_2{:}\langle \mathsf{cont}\ f\rangle\ ord\}$,
    and we have $\{z\} \leftarrow \mathsf{ev/catcha}\ v\ e_2\ (y_1 \bullet y_2) :: \Delta \rightsquigarrow \Theta'\{z{:}\langle \mathsf{retn}\ v\rangle\ ord\}$.
  - If $T' = (T_1'; \{y_1\} \leftarrow \mathsf{gen/error}\ tp'\ y_1'; T_2')$,
    then because error and cont are terminals, $\Delta = \Theta'\{y_1{:}\langle \mathsf{error}\rangle\ ord,\ y_2{:}\langle \mathsf{handle}\ e\rangle\ ord\}$,
    and we have $\{z\} \leftarrow \mathsf{ev/catchb}\ e_2\ (y_1 \bullet y_2) :: \Delta \rightsquigarrow \Theta'\{z{:}\langle \mathsf{eval}\ e_2\rangle\ ord\}$.

This covers all possible first steps in the trace T, and thus completes the proof. □

10.3 Progress with mutable storage

Developing progress proofs to for stateful specifications requires a property that is dual to unique index sets (Definition 9.5, Section 9.5.1). Unique index sets require that there will be only ever be *at most one* proposition of a certain form, and the dual property, *assured index sets*, require that there is always *at least one* proposition of a certain form.

Definition 10.2. *A set S is an* assured index set *at a type τ under a generative signature Σ and an initial state $(\Psi; \Delta)$ if, whenever $(\Psi; \Delta) \rightsquigarrow^*_\Sigma (\Psi'; \Delta')$, then $\Psi \vdash_\Sigma t : \tau$ implies that, for some $\mathsf{a}/i \in S$, $x{:}\langle \mathsf{a}\ t_1 \ldots t_n\rangle\ lvl \in \Delta'$ where $t_i = t$.*

The set $\{\mathsf{gencell}/1, \mathsf{cell}/1\}$ is both a unique index set and an assured index set under $\Sigma_{Gen9.6}$ and the initial state $(\cdot; x_0{:}\langle \mathsf{gen}\ tp\rangle\ ord)$. The latter property is critical to finishing the extension of Theorem 10.1 proof in certain cases where we invoke the canonical forms lemma. When we invoked the canonical forms lemma in the cont branch of that proof, we started with the knowledge that $\Delta = \Theta'\{y_1{:}\langle \mathsf{retn}\ v\rangle\ ord,\ y_2{:}\langle \mathsf{cont}\ f\rangle\ ord\}$. Two new outcomes are introduced when we introduce mutable state as discussed in Section 6.5.1 and Section 9.4. The first is the possibility that $v = \mathsf{loc}\ l$ while $f = \mathsf{get1}$, and the second is the possibility that $f = \mathsf{set2}\ l$. In each case, we cannot proceed with rule ev/get1 or rule ev/set2, respectively, unless we also know that there is a variable binding $z{:}\mathsf{cell}\ l\ v'$ in Δ. We know precisely this because $\{\mathsf{gencell}/1, \mathsf{cell}/1\}$ is an assured index set, because $\Psi \vdash l{:}\mathsf{mutable\_loc}$, and because gencell propositions, as nonterminals, cannot appear in the generated process state Δ. Therefore, in the former case we can produce a step $\{y', z'\} \leftarrow \mathsf{ev/get1}\ l\ v'\ (y_1 \bullet y_2 \bullet z)$, and in the later case we can produce a step $\{y', z'\} \leftarrow \mathsf{ev/set2}\ v\ l\ v'\ (y_1 \bullet y_2 \bullet z)$.

10.4 Safety

We conclude by presenting the safety theorem for the ordered abstract machine specification from Figure 9.1. This theorem relates the encoding of the usual deductive formulation of the typing judgment, of $e\ tp$, to a progress property stated in terms of substructural process states.

Theorem 10.3 (Safety for ordered abstract machines)**.** *If* $T :: (x{:}\langle \mathsf{eval}\, e\rangle\ ord) \leadsto^* \Delta$ *under the signature from Figure 9.1 and* $\cdot \vdash N : \mathsf{of}\ e\ tp$*, then either*

* $\Delta \leadsto \Delta'$ *under the signature from Figure 9.1 for some* Δ'*, or else*
* $\Delta = (y{:}\langle \mathsf{retn}\ v\rangle\ ord)$ *and* $\cdot \vdash N : \mathsf{value}\ v$*, or else*
* $\Delta = (y{:}\langle \mathsf{error}\rangle\ ord)$.

Proof. First, by induction and case analysis on the last steps of T, we show that for all Δ' such that $T' :: (x{:}\langle \mathsf{eval}\, e\rangle\ ord) \leadsto^* \Delta'$ under the signature from Figure 9.1, we can construct a generative trace $T_g :: (x_0{:}\langle \mathsf{gen}\ tp\rangle\ ord) \leadsto^*_{\Sigma_{Gen9.4}} \Delta'$:

Base case $T' = \diamond$.
Construct $T_g = \{x\} \leftarrow \mathsf{gen/eval}\ tp\ e\ (x_0 \bullet !N) :: (x_0{:}\langle \mathsf{gen}\ tp\rangle\ ord) \leadsto^*_{\Sigma_{Gen9.4}} (x{:}\langle \mathsf{eval}\, e\rangle\ ord)$.

Inductive case $T' = T''; S$, where $T'' :: (x_0{:}\langle \mathsf{gen}\ tp\rangle\ ord) \leadsto^*_{\Sigma_{Gen9.4}} \Delta''$ and $S :: \Delta'' \leadsto_{\Sigma_{Gen9.4}} \Delta'$.
By the induction hypothesis, we have $T'_g :: (x_0{:}\langle \mathsf{gen}\ tp\rangle\ ord) \leadsto^*_{\Sigma_{Gen9.4}} \Delta''$. By preservation (Theorem 9.3) on T'_g and S, we have $T'_g :: (x_0{:}\langle \mathsf{gen}\ tp\rangle\ ord) \leadsto^*_{\Sigma_{Gen9.4}} \Delta'$.

This means, in particular, that we can construct $T_g :: (x_0{:}\langle \mathsf{gen}\ tp\rangle\ ord) \leadsto^*_{\Sigma_{Gen9.4}} \Delta$.

By the progress theorem (Theorem 10.1) on T_g, there are three possibilities:

* If $\Delta \leadsto \Delta'$, then we're done.
* If $T_g :: (T_1; \{y\} \leftarrow \mathsf{gen/retn}\ tp\ v\ (x_0 \bullet !N' \bullet !N'_v); T_2)$, then by a trivial case analysis on T_1 and T_2 we can conclude that both are empty and, therefore, that $\Delta = (y{:}\langle \mathsf{retn}\ v\rangle\ ord)$.
* If $T_g :: (T_1; \{y\} \leftarrow \mathsf{gen/error}\ tp\ x_0; T_2)$, then by a trivial case analysis on T_1 and T_2 we can conclude that both are empty and, therefore, that $\Delta = (y{:}\langle \mathsf{error}\rangle\ ord)$.

This concludes the proof of safety. □

Chapter 11

Conclusion

This document has endeavored to support the following thesis:

> **Thesis Statement:** *Logical frameworks based on a rewriting interpretation of substructural logics are suitable for modular specification of programming languages and formal reasoning about their properties.*

In the service of this thesis, we first developed a logical framework of substructural logical specifications (SLS) based on a rewriting interpretation of ordered linear lax logic (OL_3). Part I of the dissertation discusses the design of this logical framework, and in the process firmly establishes the elegant connection between two sets of techniques:

1. Canonical forms and hereditary substitution in a logical framework, on one hand, and
2. Focused derivations and cut admissibility in logic, on the other.

The broad outlines of this connection have been known for a decade, but this dissertation gives the first account of the connection that generalizes to all logical connectives. This connection allows the SLS framework to be presented as a syntactic refinement of focused ordered linear lax logic; the steps and traces of SLS, which provide its rewriting interpretation, are justified as partial proofs in focused ordered linear lax logic. SLS does move beyond the connection with focused logic due to the introduction of concurrent equality, which allows logically independent steps in a trace to be reordered; we conjecture that the resulting equivalence relation imposed on our logical framework is analogous to the one given by multifocusing in logic, but a full exposition of this connection is left for future work.

The SLS framework acts as a bridge between the world of logical frameworks, where deductive derivations are the principal objects of study, and the world of rewriting logic, where rewriting sequences that are similar to SLS traces are the principal objects of study. Part II of this dissertation discusses a number of ways of describing operational semantics specifications in SLS, using ordered resources to encode control structures, using mobile/linear resources to encode mutable state and concurrent communication, and using persistent resources to represent memoization and binding. Different styles of specification are connected to each other through systematic transformations on SLS specifications that we prove to be generally sound, a methodology named the *logical correspondence*, following Danvy et al.'s functional correspondence.

Most of the systematic transformations discussed in Chapter 6 and Chapter 7 – operationalization, defunctionalization, and destination-adding – are implemented in the SLS prototype implementation. Utilizing this implementation, we show in Appendix B that it is possible to fuse together a single coherent SLS specification of a MiniML language with concurrency, state, and communication using various different styles of specification, including natural semantics where appropriate.

This dissertation also discusses two different methodologies for formally reasoning about properties of operational semantics specifications in SLS. The program analysis methodology considered in Chapter 8 allows us to derive effectively executable abstractions of programming language semantics directly from operational semantics specifications in SLS. The methodology of progress, preservation, and type safety considered in Chapter 9 and Chapter 10 is presented as a natural extension of traditional "safety = progress + preservation" reasoning. In a sense, the work described in this document has pushed our ability to reason *formally* about properties of SLS specifications (and substructural operational semantics specifications in particular) some distance beyond our ability to *informally* reason about these specifications. An important direction for future work will be to move beyond the misleadingly-sequential language of SLS traces and develop a more user-friendly language for writing, talking, and thinking about traces in SLS, especially generative traces.

Appendix A

Process states summary

In this dissertation, we emphasize the use of *process states* to describe the internal states of the evolving systems we are interested in. This view is an extension of Miller's *processes-as-formula* interpretation [Mil93, DCS12]. Of course, a process state is not a formula; Section 4.7.2 discusses our emphasis on SLS process states instead of SLS formulas as the fundamental representation of the internal states of evolving systems.

A process state as defined in Chapter 4 has the form $(\Psi; \Delta)_\sigma$, though outside of Chapter 4 we never use the associated substitution σ, writing $(\Psi; \Delta)$ to indicate the empty substitution $\sigma = \cdot$. The first-order context Ψ, which is sometimes omitted, is also called the *LF context* in SLS because Spine Form LF is the first-order term language of SLS (Section 4.1). Δ is a *substructural context*.

A.1 Substructural contexts

A substructural context (written as Δ and occasionally as Ξ) is a sequence of variable bindings $x{:}T\ lvl$ – all the variables x bound in a context must be distinct. In SLS, lvl is either ord (ordered resources), eph (mobile resources, also called ephemeral or linear resources), or $pers$ (persistent resources).

In stable process states, T is usually a *suspended positive atomic proposition* $\langle Q \rangle$. The permeability of a positive atomic proposition (ordered, mobile/linear/ephemeral, or persistent) is one of its intrinsic properties (Section 2.5.4, Section 3.3.2), so we can write $x{:}\langle Q \rangle$ instead of $x{:}\langle Q \rangle\ ord$, $x{:}\langle Q \rangle\ eph$, or $x{:}\langle Q \rangle\ pers$ if the permeability of Q is known from the context. So the encoding of the string [< > ([ ]), described in the introduction as

$$\mathsf{left(sq)}\ \ \mathsf{left(an)}\ \ \mathsf{right(an)}\ \ \mathsf{left(pa)}\ \ \mathsf{left(sq)}\ \ \mathsf{right(sq)}\ \ \mathsf{right(pa)}$$

is more properly described as

$$x_1{:}\langle \mathsf{left\ sq} \rangle,\ \ x_2{:}\langle \mathsf{left\ an} \rangle,\ \ x_3{:}\langle \mathsf{right\ an} \rangle,\ \ x_4{:}\langle \mathsf{left\ pa} \rangle,\ \ x_5{:}\langle \mathsf{left\ sq} \rangle,\ \ x_6{:}\langle \mathsf{right\ sq} \rangle,\ \ x_7{:}\langle \mathsf{right\ pa} \rangle$$

We write $x_1{:}\langle \mathsf{left\ sq} \rangle$ instead of $x_1{:}\langle \mathsf{left\ sq} \rangle\ ord$ above, leaving implicit the fact that left and right are ordered predicates.

It is also possible, in *nested* SLS specifications (Section 5.1, Section 6.1), to have variable bindings $x{:}A^{-}\ ord$, $x{:}A^{-}\ eph$, and $x{:}A^{-}\ pers$. These nested specifications act much like rules in the SLS signature, though mobile rules ($x{:}A^{-}\ eph$) can only be used one time, and ordered rules ($x{:}A^{-}\ ord$) can only be used one time and only in one particular part of the context (Figure 5.2).

Chapter 3 treats substructural contexts strictly as sequences, but in later chapters we treat substructural contexts in a more relaxed fashion, allowing mobile/linear/ephemeral and persistent variable bindings to be tacitly reordered relative to one another other and relative to ordered propositions. In this relaxed reading, $(x_1{:}\langle Q_1\rangle\ ord, x_2{:}\langle Q_2\rangle\ ord)$ and $(x_2{:}\langle Q_2\rangle\ ord, x_1{:}\langle Q_1\rangle\ ord)$ are not equivalent contexts but $(x_3{:}\langle Q_3\rangle\ pers, x_2{:}\langle Q_2\rangle\ ord)$ and $(x_2{:}\langle Q_2\rangle\ ord, x_3{:}\langle Q_3\rangle\ pers)$ are.

A *frame* Θ represents a context with a hole in it. The notation $\Theta\{\Delta\}$ tacks the substructural context Δ into the hole in Θ; the context and the frame must have disjoint variable domains for this to make sense. In Chapter 3, frames are interrupted sequences of variable bindings $x_1{:}T_1\ lvl, \ldots x_n{:}T_n\ lvl, \Box, x_{n+1}{:}T_{n+1}\ lvl, \ldots x_m{:}T_m\ lvl$, where the box represents the hole. In later chapters, this is relaxed in keeping with the relaxed treatment of contexts modulo reordering of mobile and persistent variable bindings.

A.2 Steps and traces

Under focusing, a SLS proposition can correspond to some number of synthetic transitions (Section 2.4, Section 4.2.6). The declaration $\mathsf{rule} : Q_1 \bullet Q_2 \rightarrowtail \{Q_3 \bullet Q_2\}$[1] in an SLS signature Σ, where Q_1 is ordered, Q_2 is mobile, and Q_3 is persistent, is associated with this synthetic transition:

$$\Theta\{x_1{:}\langle Q_1\rangle\ ord,\ \ x_2{:}\langle Q_2\rangle\ eph\} \rightsquigarrow_\Sigma \Theta\{y_1{:}\langle Q_3\rangle\ pers,\ \ y_2{:}\langle Q_2\rangle\ eph\}$$

The variable bindings x_1 and x_2 no longer appear in $\Theta\{y_1{:}\langle Q_3\rangle\ pers, y_2{:}\langle Q_2\rangle\ eph\}$. The proof terms associated with synthetic transitions are *steps* (Section 4.2.6), and the step corresponding to the synthetic transition above is written as $\{y_1, y_2\} \leftarrow \mathsf{rule}\,(x_1 \bullet x_2)$. As described in Section 4.2.6, we can relate the step to the synthetic transition like this:

$$\{y_1, y_2\} \leftarrow \mathsf{rule}\,(x_1 \bullet x_2) :: \Theta\{x_1{:}\langle Q_1\rangle\ ord,\ \ x_2{:}\langle Q_2\rangle\ eph\} \rightsquigarrow_\Sigma \Theta\{y_1{:}\langle Q_3\rangle\ pers,\ \ y_2{:}\langle Q_2\rangle\ eph\}$$

As described in Section 4.2.7, we can also use a more Hoare-logic inspired notation:

$$\begin{array}{c} \Theta\{x_1{:}\langle Q_1\rangle\ ord,\ \ x_2{:}\langle Q_2\rangle\ eph\} \\ \{y_1, y_2\} \leftarrow \mathsf{rule}\,(x_1 \bullet x_2) \\ \Theta\{y_1{:}\langle Q_3\rangle\ pers,\ \ y_2{:}\langle Q_2\rangle\ eph\} \end{array}$$

The reflexive-transitive closure of $\rightsquigarrow_\Sigma$ is $\rightsquigarrow^*_\Sigma$, and the proof terms witnessing these sequences of synthetic transitions are *traces* $T ::= \diamond \mid S \mid T;T$. *Concurrent equality* (Section 4.3) is an equivalence relation on traces that allows us to rearrange the steps $S_1 = \{P_1\} \leftarrow R_1$ and $S_2 = \{P_2\} \leftarrow R_2$ in a trace when the variables introduced by P_1 (the output interface of S_1, written $S_1{}^\bullet$) are not mentioned in R_2 (the input interface of S_2, written ${}^\bullet S_2$) and vice versa.

[1] This is synonymous with the proposition $Q_1 \bullet Q_2 \rightarrowtail \bigcirc(Q_3 \bullet Q_2)$ (Section 4.2).

Appendix B

A hybrid specification of Mini-ML

In this section, we present the specification that was illustrated in Figure 5.4 as a full SLS specification. This specification is a hybrid or chimera: it has individual features that are presented using big-step natural semantics, nested ordered abstract machine semantics, flat ordered abstract machine semantics, and destination-passing semantics.

Illustrating the logical correspondence methodology that we introduced in Chapter 5 and expounded upon in Chapters 6 and 7, all these specifications can transformed into a common flat destination-passing semantics. With the exception of Concurrent ML primitives, which were only alluded to in Section 7.2.2, all the pieces of this specification (or very similar variants) have been presented elsewhere in the dissertation. The specification in this section is careful to present the entire SLS specification, as opposed to other examples in which the relevant LF declarations were almost always omitted.

The lowest common denominator of destination-passing semantics can be represented in CLF, and the SLS implementation is able to output CLF code readable in Schack-Nielsen's Celf implementation [SNS08]. The implemented logic programming engine of Celf is therefore able to *execute* Mini-ML programs encoded as terms of type exp in our hybrid specification.

B.1 Pure Mini-ML

There are various toy languages calling themselves "Mini-ML" in the literature. All Mini-MLs reflect some of the flavor of functional programming while avoiding features such as complex pattern-matching and datatype declarations that make the core language of Standard ML [MTHM97] a bit more complicated. Of course, Mini-MLs universally avoid the sophisticated ML module language as well.

Like the PCF language [Plo77], Mini-ML variants usually have at least a fixed-point operator, unary natural numbers, and functions. We add Boolean and pair values to this mix, as well as the arbitrary choice operator $\ulcorner e_1 \,\textcircled{?}\, e_2 \urcorner = \mathsf{arb} \ulcorner e_1 \urcorner \ulcorner e_2 \urcorner$ from Section 6.4.1. The specification in Section B.1.2 is an encoding of the natural semantics judgment $\ulcorner e \Downarrow v \urcorner = \mathsf{ev} \ulcorner e \urcorner \ulcorner v \urcorner$ presented throughout Chapter 6. The language is pure – the only effect is nontermination – so we can fully specify the language as a big-step operational semantics.

B.1.1 Syntax

```
exp: type.
lam: (exp -> exp) -> exp.                ; fn x => e
app: exp -> exp -> exp.                  ; e(e)
fix: (exp -> exp) -> exp.                ; fix x.e
true: exp.                               ; tt
false: exp.                              ; ff
ite: exp -> exp -> exp -> exp.           ; if e then et else ef
zero: exp.                               ; z
succ: exp -> exp.                        ; s(e)
case: exp -> exp -> (exp -> exp) -> exp. ; case e of z => ez | s x => es
unit: exp.                               ; <>
pair: exp -> exp -> exp.                 ; <e1, e2>
fst: exp -> exp.                         ; e.1
snd: exp -> exp.                         ; e.2
arb: exp -> exp -> exp.                  ; e1 ?? e2
```

B.1.2 Natural semantics

```
#mode ev + -.
ev: exp -> exp -> prop.

ev/lam:   ev (lam \x. E x) (lam \x. E x).

ev/app:   ev (app E1 E2) V
           <- ev E1 (lam \x. E x)
           <- ev E2 V2
           <- ev (E V2) V.

ev/fix:   ev (fix \x. E x) V
           <- ev (E (fix \x. E x)) V.

ev/true:  ev true true.

ev/false: ev false false.

#mode caseb + + + -.
caseb: exp -> exp -> exp -> exp -> prop.

ev/ite:   ev (ite E Et Ef) V
           <- ev E V'
           <- caseb V' Et Ef V.

case/t:   caseb true Et Ef V
           <- ev Et V.

case/f:   caseb false Et Ef V
           <- ev Et V.

ev/zero:  ev zero zero.
```

```
ev/succ:  ev (succ E) (succ V)
           <- ev E V.

#mode casen + + + -.
casen: exp -> exp -> (exp -> exp) -> exp -> prop.

ev/case:  ev (case E Ez \x. Es x) V
           <- ev E V'
           <- casen V' Ez (\x. Es x) V.

case/z:   casen zero Ez (\x. Es x) V
           <- ev Ez V.

case/s:   casen (succ V) Ez (\x. Es x) V
           <- ev (Es V) V.

ev/unit:  ev unit unit.

ev/pair:  ev (pair E1 E2) (pair V1 V2)
           <- ev E1 V1
           <- ev E2 V2.

ev/fst:   ev (fst E) V1
           <- ev E (pair V1 V2).

ev/snd:   ev (snd E) V2
           <- ev E (pair V1 V2).

ev/arb1:  ev (arb E1 E2) V
           <- ev E1 V.

ev/arb2:  ev (arb E1 E2) V
           <- ev E2 V.
```

B.2 State

The strength of an ordered abstract machine semantics specification is its ability to handle modular addition of *stateful* features. While Section 6.5 discussed the modular extension of *flat* ordered abstract machines, nested ordered abstract machines are also perfectly capable of handling stateful features such as mutable storage (Section 6.5.1) and call-by-need recursive suspensions (Section 6.5.2).

B.2.1 Syntax

```
mutable_loc: type.
loc: mutable_loc -> exp.                  ; (no concrete syntax)
ref: exp -> exp.                          ; ref e
get: exp -> exp.                          ; !e
set: exp -> exp -> exp.                   ; e1 := e2
```

```
bind_loc: type.
issusp: bind_loc -> exp.                  ; (no concrete syntax)
thunk: (exp -> exp) -> exp.               ; thunk x.e
force: exp -> exp.                        ; force e
```

B.2.2 Nested ordered abstract machine semantics

In Section 6.5.1, we discussed mutable storage as a flat ordered abstract machine (Figure 6.14), but it is equally straightforward to describe a nested ordered abstract machine for mutable storage.

```
cell: mutable_loc -> exp -> prop lin.

ev/loc: eval (loc L) >-> {retn (loc L)}.

ev/ref: eval (ref E1)
         >-> {eval E1 *
              (All V1. retn V1
                >-> {Exists l. retn (loc l) * $cell l V1})}.

ev/get: eval (get E1)
         >-> {eval E1 *
              (All L. retn (loc L) * $cell L V
                >-> {retn V * $cell L V})}.

ev/set: eval (set E1 E2)
         >-> {eval E1 *
              (All L. retn (loc L)
                >-> {eval E2 *
                     (All V2. All Vignored. retn V2 * $cell L Vignored
                       >-> {retn unit * $cell L V2})})}.
```

B.2.3 Flat ordered abstract machine semantics

In Section 6.5.2, we gave both a semantics for call-by-need recursive suspensions, both as a flat ordered abstract machine (Figure 6.16) and a nested ordered abstract machine (Figure 6.17). However, the nested ordered abstract machine from Figure 6.17 uses the with connective $A^- \,\&\, B^-$, and our implementation of defunctionalization transformation doesn't handle this connective. Therefore, we repeat the flat ordered abstract machine from Figure 6.16. Note, however, that there is no technical reason why $A^- \,\&\, B^-$ should be difficult to handle; any actual difficulty is mostly in terms of making sure uncurrying (Section 6.2.2) does something sensible.

```
susp: bind_loc -> (exp -> exp) -> prop lin.
blackhole: bind_loc -> prop lin.
bind: bind_loc -> exp -> prop pers.

force1: frame.
bind1: bind_loc -> frame.
```

```
ev/susp:       eval (issusp L) >-> {retn (issusp L)}.

ev/thunk:      eval (thunk \x. E x)
                >-> {Exists l. $susp l (\x. E x) * retn (issusp l)}.

ev/force:      eval (force E) >-> {eval E * cont force1}.

ev/suspended1: retn (issusp L) * cont force1 * $susp L (\x. E' x)
                >-> {eval (E' (issusp L)) * cont (bind1 L) * $blackhole L}.

ev/suspended2: retn V * cont (bind1 L) * $blackhole L
                >-> {retn V * !bind L V}.

ev/memoized:   retn (issusp L) * cont force1 * !bind L V
                >-> {retn V}.
```

B.3 Failure

The reason we introduced frames in Section 6.2.3 was to allow the semantics of recoverable failure to talk generically about all continuations. In Section B.3.2, we generalize the semantics from Section 6.5.4 by having exceptions carry a value.

B.3.1 Syntax

```
raise: exp -> exp.                        ; raise e
try: exp -> (exp -> exp) -> exp.          ; try e catch x.ef
```

B.3.2 Flat ordered abstract machine semantics

```
handle: (exp -> exp) -> prop ord.
error: exp -> prop ord.

raise1: frame.

ev/raise:   eval (raise E) >-> {eval E * cont raise1}.

ev/raise1:  retn V * cont raise1 >-> {error V}.

ev/try:     eval (try E1 (\x. E2 x))
             >-> {eval E1 * handle (\x. E2 x)}.

error/cont: error V * cont F >-> {error V}.

error/hand: error V * handle (\x. E2 x) >-> {eval (E2 V)}.

retn/hand:  retn V * handle (\x. E2 x) >-> {retn V}.
```

B.4 Parallelism

While ordered abstract machines can represent parallel evaluation, and the operationalization transformation can expose it, parallel ordered abstract machines and the destination-adding transformation do not interact a helpful way. Therefore, for our hybrid specification, we will describe parallel evaluation at the destination-passing level, as in Section 7.2.1.

B.4.1 Destination-passing semantics

Instead of the parallel pairs shown in Figure 6.8 and Figure 7.6, we will use a parallel let construct $\ulcorner \mathsf{letpar}\ (x_1, x_2) = (e_1, e_2)\ \mathsf{in}\ e \urcorner = \mathsf{letpar} \ulcorner e_1 \urcorner \ulcorner e_2 \urcorner\ \lambda x_1.\ \lambda x_2. \ulcorner e \urcorner$.

```
cont2: frame -> dest -> dest -> dest -> prop lin.

letpar: exp -> exp -> (exp -> exp -> exp) -> exp.
letpar1: (exp -> exp -> exp) -> frame.

ev/letpar:  eval (letpar E1 E2 \x. \y. E x y) D
             >-> {Exists d1. eval E1 d1 *
                  Exists d2. eval E2 d2 *
                  cont2 (letpar1 \x. \y. E x y) d1 d2 D}.

ev/letpar1: retn V1 D1 * retn V2 D2 *
            cont2 (letpar1 \x. \y. E x y) D1 D2 D
             >-> {eval (E V1 V2) D}.
```

B.4.2 Integration of parallelism and exceptions

We have discussed two semantics for parallel evaluation. The first semantics, in Section 6.5.4, only raised an error if both parallel branches terminated and one raised an error. The second semantics, in Section 7.2.1, raised an error if *either* branch raised an error, and then allowed the other branch to return a value.

We will demonstrate a third option here, the sequential exception semantics used by Manticore [FRR08]. An error raised by the second scrutinee e_2 of letpar will only be passed up the stack if the first scrutinee e_1 returns a value. We also represent Manticore's *cancellation* – if the first branch of a parallel evaluation raises an exception, then rather than passively waiting for the second branch to terminate we proactively walk up its stack attempting to cancel the computation.

```
cancel: dest -> prop lin.

ev/errorL:    error V D1 * cont2 X D1 D2 D
               >-> {error V D * cancel D2}.

ev/errorR:    retn _ D1 * error V D2 * cont2 _ D1 D2 D
               >-> {error V D}.
```

```
cancel/eval:  eval _ D * cancel D >-> {one}.
cancel/retn:  retn _ D * cancel D >-> {one}.
cancel/error: error _ D * cancel D >-> {one}.
cancel/cont:  cont _ D' D * cancel D >-> {cancel D'}.
cancel/cont2: cont2 _ D1 D2 D * cancel D >-> {cancel D1 * cancel D2}.
```

B.5 Concurrency

Concurrent ML is an excellent example of the power of destination-passing specifications. The Concurrent ML primitives allow a computation to develop a rich interaction structure that does not mesh well with the use of ordered logic, but the destination-passing style allows for a clean specification that is fundamentally like the one used for simple synchronization in Section 7.2.2. This account directly follows Cervesato et al.'s account [CPWW02], similarly neglecting negative acknowledgements.

B.5.1 Syntax

```
channel: type.
spawn: exp -> exp.                          ; spawn e
exit: exp.                                  ; exit
newch: exp.                                 ; channel
chan: channel -> exp.                       ; (no concrete syntax)
sync: exp -> exp.                           ; sync e
send: exp -> exp -> exp.                    ; send c e
recv: exp -> exp.                           ; recv c
always: exp -> exp.                         ; always e
choose: exp -> exp -> exp.                  ; e1 + e2
never: exp.                                 ; 0
wrap: exp -> (exp -> exp) -> exp.           ; wrap e in x.e'
```

B.5.2 Natural semantics

Many of the pieces of Concurrent ML do not interact with concurrency directly; instead, they build channels and event values that drive synchronization. In our hybrid specification methodology, we can give these pure parts of the Concurrent ML specification a big-step natural semantics specification.

```
ev/chan:    ev (chan C) (chan C).

ev/always:  ev (always E1) (always V1)
             <- ev E1 V1.

ev/recv:    ev (recv E1) (recv V1)
             <- ev E1 V1.

ev/send:    ev (send E1 E2) (send V1 V2)
             <- ev E1 V1
             <- ev E2 V2.
```

```
ev/choose:  ev (choose E1 E2) (choose V1 V2)
             <- ev E1 V1
             <- ev E2 V2.

ev/never:   ev never never.

ev/wrap:    ev (wrap E1 \x. E2 x) (wrap V1 \x. E2 x)
             <- ev E1 V1.
```

B.5.3 Destination-passing semantics

The destination-passing semantics of Concurrent ML has three main parts. The first part, a spawn primitive, creates a new disconnected thread of computation – the same kind of disconnected thread that we used for the interaction of parallelism and failure in Section 7.2.1. The newch primitive creates a new channel for communication.

```
terminate: dest -> prop lin.

term/retn:  retn _ D * terminate D >-> {one}.
term/error: error _ D * terminate D >-> {one}.

ev/spawn:   eval (spawn E) D
             >-> {retn unit D *
                  Exists d'. eval E d' * terminate d'}.

ev/newch:   eval newch D >-> {Exists c. retn (chan c) D}.
```

The critical feature of Concurrent ML is synchronization, which is much more complex than the simple synchronization described in Section 7.2.2, and has something of the flavor of the labels described in that section. An action can include many alternatives, but if a send and a receive can simultaneously take place along a single channel, then the synch/communicate rule can enable both of the waiting synch $v\ d$ propositions to proceed evaluating as eval propositions.

Here as in Cervesato et al.'s specification, events are atomically paired up using the backward-chaining action rules, which are not transformed: the intent is for the action predicate to act like a backtracking, backward-chaining logic programs in the course of evaluation.

```
#mode action + - -.
action: exp -> exp -> (exp -> exp) -> prop.

act/t: action (always V) (always V) (\x. x).
act/s: action (send (chan C) V) (send (chan C) V) (\x. x).
act/v: action (recv (chan C)) (recv (chan C)) (\x. x).
act/l: action (choose Event1 Event2) Lab (\x. E x)
        <- action Event1 Lab (\x. E x).
act/r: action (choose Event1 Event2) Lab (\x. E x)
        <- action Event2 Lab (\x. E x).
act/w: action (wrap Event1 \x. E2 x) Lab (\x. app (lam (\x. E2 x)) (E x))
        <- action Event1 Lab (\x. E x).
```

```
synch: exp -> dest -> prop lin.

sync1: frame.

ev/sync:
  eval (sync E1) D >-> {Exists d1. eval E1 d1 * cont sync1 d1 D}.

ev/sync1:
  retn W D1 * cont sync1 D1 D >-> {synch W D}.

synch/always:
  synch Event D *
  !action Event (always V') (\x. E x)
   >-> {eval (E V') D}.

synch/communicate:
  synch Event1 D1 *
  !action Event1 (send (chan C) V) (\x. E1 x) *
  synch Event2 D2 *
  !action Event2 (recv (chan C)) (\x. E2 x)
   >-> {eval (E1 unit) D1 * eval (E2 V) D2}.
```

B.6 Composing the semantics

Within Standard ML, we can read the various specifications described in this appendix and use directives to successively operationalize, defunctionalize, add destinations, and output CLF that is readable by the Celf implementation.

```
CM.make "../../sls/sources.cm";
fun HEADING s = print ("\n\n== "^s^" ==\n\n");
Frontend.init ();

HEADING "NATURAL SEMANTICS";
Frontend.reset ();
Frontend.read "#operationalize \"ord-nested.auto.sls\" \
\                 (ev ~> eval retn)\
\                 (casen ~> casen retn)\
\                 (caseb ~> caseb retn).";
Frontend.load "compose/pure-exp.sls";
Frontend.load "compose/pure-natsem.sls";
Frontend.load "compose/concur-exp.sls";
Frontend.load "compose/concur-natsem.sls";
Frontend.read "#operationalize stop.";

HEADING "ORDERED ABSTRACT MACHINES (nested)";
Frontend.reset ();
Frontend.read "#defunctionalize \"ord-flat.auto.sls\" \
\                 (cont frame : ord).";
Frontend.load "ord-nested.auto.sls";
```

```
Frontend.load "compose/imp-exp.sls";
Frontend.load "compose/imp-ordmachine.sls";
Frontend.read "#defunctionalize stop.";

HEADING "ORDERED ABSTRACT MACHINES (flat)";
Frontend.reset ();
Frontend.read "#destadd \"dest.auto.sls\" \
\                 dest eval retn error casen caseb.";
Frontend.load "ord-flat.auto.sls";
Frontend.load "compose/control-exp.sls";
Frontend.load "compose/control-ordmachine.sls";
Frontend.load "compose/susp-ordmachine.sls";

HEADING "DESTINATION-PASSING";
Frontend.reset ();
Frontend.read "#clf \"miniml.clf\".";
Frontend.load "dest.auto.sls";
Frontend.load "compose/par-dest1.sls";
Frontend.load "compose/par-dest2.sls";
Frontend.load "compose/concur-dest1.sls";
Frontend.load "compose/concur-dest2.sls";
Frontend.read "#clf stop.";
```

Bibliography

[ABDM03] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19. ACM, 2003. 5.2

[ADM04] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters*, 90(5):223–232, 2004. doi:10.1016/j.ipl.2004.02.012. 5.2

[ADM05] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 342(1):149–172, 2005. doi:10.1016/j.tcs.2005.06.008. 5.2

[Age04] Mads Sig Ager. From natural semantics to abstract machines. In *Logic Based Program Synthesis and Transformation (LOPSTR'04)*, pages 245–261. Springer LNCS 3573, 2004. 5.2, 6.5

[AK99] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1999. 4.6.1

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools, Second Edition*. Pearson Education, Inc, 2007. 8.5

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992. doi:10.1093/logcom/2.3.297. 1.1, 2.1, 2.3, 2.3.1, 2.5

[And01] Jean-Marc Andreoli. Focussing and proof construction. *Annals of Pure and Applied Logic*, 107:131–163, 2001. doi:10.1016/S0168-0072(00)00032-4. 2.4, 2.6, 3, 4.6.1

[AP01] Pablo Armelín and David Pym. Bunched logic programming. In *Proceedings International Joint Conference on Automated Reasoning (IJCAR'01)*, pages 289–304. Springer LNCS 2083, 2001. 4.6.1

[Bar96] Andrew Barber. Dual intuitionistic linear logic. Technical Report ECS-LFCS-96-347, Laboratory for Foundations of Computer Sciences, University of Edinburgh, 1996. 2.1

[BB90] Gerard Berry and Gerard Boudol. The chemical abstract machine. In *Principles of*

Programming Languages, pages 81–94. ACM, 1990. 5.2

[BD02] Marco Bozzano and Giorgio Delzanno. Automated protocol verification in linear logic. In *Principles and Practice of Declarative Programming (PPDP'02)*, pages 38–49. ACM, 2002. 8.6

[BDM02] Marco Bozzano, Giorgio Delzanno, and Maurizio Martelli. An effective fixpoint semantics for linear logic programs. *Theory and Practice of Logic Programming*, 2(1):85–122, 2002. doi:10.1017/S1471068402001254. 8.6

[BDM04] Marco Bozzano, Giorgio Delzanno, and Maurizio Martelli. Model checking linear logic specifications. *Theory and Practice of Logic Programming*, 4(5–6):573–619, 2004. doi:10.1017/S1471068404002066. 8.6

[BE06] Ahmed Bouajjani and Javier Esparza. Rewriting models of boolean programs. In F. Pfenning, editor, *Proceedings of the 17th International Conference on Term Rewriting and Applications (RTA'06)*, pages 136–150. Springer LNCS 4098, 2006. 1.1

[Bel82] Nuel D. Belnap. Display logic. *Journal of Philosophical Logic*, 11(4):375–417, 1982. doi:10.1007/BF00284976. 3.8

[BG96] Paola Bruscoli and Alessio Guglielmi. A linear logic view of Gamma style computations as proof searches. In Jean-Marc Andreoli, Chris Hankin, and Daniel Le Métayer, editors, *Coordination programming: mechanisms, models and semantics*, pages 249–273. Imperial College Press, 1996. 4.6.2

[BRF10] Hariolf Betz, Frank Raiser, and Thom Frühwirth. A complete and terminating execution model for constraint handling rules. *Theory and Practice of Logic Programming*, 10(4–6):597–610, 2010. doi:10.1017/S147106841000030X. 4.6.2

[BW96] Nick Benton and Philip Wadler. Linear logic, monads and the lambda calculus. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS'96)*, pages 420–431, New Brunswick, NJ, 1996. 2.5, 2.5.3

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL'77)*, pages 238–252. ACM, 1977. 8

[CCP03] Bor-Yuh Evan Chang, Kaustuv Chaudhuri, and Frank Pfenning. A judgmental analysis of linear logic. Technical Report CMU-CS-03-131R, Department of Computer Science, Carnegie Mellon University, April 2003. Revised December 2003. 2.1, 2, 2.5.3

[CDD+85] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, Laurent Hascoet, and Gilles Kahn. Natural semantics on the computer. Technical Report 416, INRIA, June 1985. 5.1

[CDDK86] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ml. In *LISP and Functional Programming (LFP'86)*, pages 13–27. ACM, 1986. 5.1

[CDE+11] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-

Oliet, José Meseguer, and Carolyn Talcott. *Maude Manual (Version 2.6)*. University of Illinois, Urbana-Champaign, 2011. 1.3, 4.6.2

[CdPR99] Iliano Cervesato, Valeria de Paiva, and Eike Ritter. Explicit substitutions for linear logical frameworks. In *Proceedings of the Workshop on Logical Frameworks and Meta-Languages*, Paris, France, 1999. 3.8

[CH12] Flávio Cruz and Kuen-Bang (Favonia) Hou. Parallel programming and cost semantics, May 2012. Unpublished note. 6.1

[Cha06] Kaustuv Chaudhuri. *The Focused Inverse Method for Linear Logic*. PhD thesis, Carnegie Mellon University, 2006. 2.3, 2.3.2, 2.5, 14, 3.4.1

[Cha08] Kaustuv Chaudhuri. Focusing strategies in the sequent calculus of synthetic connectives. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-15)*, pages 467–481. Springer LNCS 5330, 2008. 2.3.9

[Che12] James Cheney. A dependent nominal type theory. *Logical Methods in Computer Science*, 8:1–29, 2012. doi:10.2168/LMCS-8(1:8)2012. 6.5.1

[Chi95] Jawahar Lal Chirimar. *Proof Theoretic Approach To Specification Languages*. PhD thesis, University of Pennsylvania, 1995. 1.1, 2.1

[CHP00] Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. *Theoretical Computer Science*, 232(1–2):133–163, 2000. doi:10.1016/S0304-3975(99)00173-5. 4.6.1

[Cla87] Keith L. Clark. Negation as failure. In *Readings in nonmonotonic logic*, pages 311–325. Morgan Kaufmann Publishers, Inc., 1987. 4.6.1

[CMS09] Kaustuv Chaudhuri, Dale Miller, and Alexis Saurin. Canonical sequent proofs via multi-focusing. In *Fifth IFIP International Conference on Theoretical Computer Science (TCS 2008, Track B)*, pages 383–396, 2009. 4.3.1, 4.3.1

[CP02] Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information and Computation*, 179(1):19–75, 2002. doi:10.1006/inco.2001.2951. 2.1, 3.3.3, 4.1, 4.1.1, 4.1.3, 4.4, 5.2, 9.1.3, 10

[CPP08] Kaustuv Chaudhuri, Frank Pfenning, and Greg Price. A logical characterization of forward and backward chaining in the inverse method. *Journal of Automated Reasoning*, 40(2–3):133–177, 2008. doi:10.1007/s10817-007-9091-0. 4.6.1

[CPS+12] Iliano Cervesato, Frank Pfenning, Jorge Luis Sacchini, Carsten Schürmann, and Robert J. Simmons. Trace matching in a concurrent logical framework. In *Proceedings of the International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'12)*, Copenhagen, Denmark, 2012. 4.3, 4.3

[CPWW02] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-2002-002, Department of Computer Science, Carnegie Mellon University, March 2002. Revised May 2003. 1.1, 2.1, 4.4, 5, 5.1, 7, 7.2, 7.2.2, B.5

[Cra10] Karl Crary. Higher-order representation of substructural logics. In *International*

Conference on Functional Programming (ICFP'10), pages 131–142. ACM, 2010. 3.8, 4.7.3

[CS09] Iliano Cervesato and Andre Scedrov. Relating state-based and process-based concurrency through linear logic (full-version). *Information and Computation*, 207(10):1044–1077, 2009. doi:10.1016/j.ic.2008.11.006. 1, 1.1, 2.1, 3, 4.6.2, 4.7.2

[CS13] Iliano Cervesato and Thierry Sans. Substructural meta-theory of a type-safe language for web programming. *Fundamenta Informaticae*, 2013. 9, 9.5

[Dan03] Olivier Danvy. A rational deconstruction of Landin's SECD machine. Technical Report RS-03-33, BRICS, October 2003. 6

[Dan08] Olivier Danvy. Defunctionalized interpreters for programming languages. In *International Conference on Functional Programming (ICFP'08)*, pages 131–142. ACM, 2008. Invited talk. 5.2, 5.2, 6.5.5, 6.6.2

[Dav96] Rowan Davies. A temporal-logic approach to binding-time analysis. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS'96)*, pages 184–195, New Brunswick, New Jersey, 1996. 6.6.3

[DCS12] Yuxing Deng, Iliano Cervesato, and Robert J. Simmons. Relating reasoning methodologies in linear logic and process algebra. In *Second International Workshop on Linearity (LINEARITY'12)*, Tallinn, Estonia, 2012. 4.7.2, A

[Die90] Volker Diekert. *Combinatorics on Traces*. Springer LNCS 454, 1990. 4.3

[DMMZ12] Olivier Danvy, Kevin Millikin, Johan Munk, and Ian Zerny. On inter-deriving small-step and big-step semantics: A case study for storeless call-by-need evaluation. *Theoretical Computer Science*, 435:21–42, 2012. doi:10.1016/j.tcs.2012.02.023. 5.2, 6.6.1

[DP09] Henry DeYoung and Frank Pfenning. Reasoning about the consequences of authorization policies in a linear epistemic logic. In *Workshop on Foundations of Computer Security*, pages 9–23, Los Angelas, CA, 2009. 4.6.3

[ER12] Chucky Ellison and Grigore Roşu. An executable formal semantics of C with applications. In *Principles of Programming Languages (POPL'12)*, pages 533–544. ACM, 2012. 1

[FLHT01] Christian Fermüller, Alexander Leitsch, Ullrich Hustadt, and Tanel Tammet. Resolution decision procedures. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, chapter 25, pages 1791–1849. Elsevier Science and MIT Press, 2001. 4.6.2

[FM97] Matt Fairtlough and Michael Mendler. Propositional lax logic. *Information and Computation*, 137(1):1–33, 1997. doi:10.1006/inco.1997.2627. 2.5.3, 3.1, 4.2

[FM12] Amy Felty and Alberto Momigliano. Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. *Journal of Automated Reasoning*, 48(1):43–105, 2012. doi:10.1007/s10817-010-9194-x. 5.2, 9.1.3, 10

[FRR08] Matthew Fluet, Mike Rainey, and John Reppy. A scheduling framework for general-purpose parallel languages. In *International Conference on Functional Programming (ICFP'08)*, pages 241–252. ACM, 2008. 6.5.4, B.4.2

[Gen35] Gerhard Gentzen. Untersuchungen über das logische schließen. i. *Mathematische Zeitschrift*, 39(1):176–210, 1935. doi:10.1007/BF01201353. 2.3.8, 2.6

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987. doi:10.1016/0304-3975(87)90045-4. 2.1

[Gir91] Jean-Yves Girard. On the sex of angels, December 1991. Post to LINEAR mailing list, archived at http://www.seas.upenn.edu/~sweirich/types/archive/1991/msg00123.html. 2.5.2

[Gir93] Jean-Yves Girard. On the unity of logic. *Annals of Pure and Applied Logic*, 59(3):201–217, 1993. doi:10.1016/0168-0072(93)90093-S. 2.5, 2.5.4

[GM02] Harald Ganzinger and David A. McAllester. Logical algorithms. In *International Conference on Logic Programming (ICLP'02)*, pages 209–223. Springer LNCS 2401, 2002. 8.1

[GMN11] Andrew Gacek, Dale Miller, and Gopalan Nadathur. Nominal abstraction. *Information and Computation*, 209(1):48–73, 2011. doi:10.1016/j.ic.2010.09.004. 6.5.1

[GNRT10] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *Principles of Programming Languages (POPL'10)*, pages 43–56. ACM, 2010. 8

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, 1989. 3.3.3

[Har12] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012. 3.3.4, 6, 6.4.1, 6.5.1, 6.5.2, 6.5.4, 8, 6.5.5, 8.5.1, 7, 9.3.2, 9.4

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993. doi:10.1145/138027.138060. 1.2, 2.2, 4.1, 6.3

[HL07] Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17(4–5):613–673, 2007. doi:10.1017/S0956796807006430. 2.2, 4.1, 4.1.2, 4.1.3, 4.1.3, 6.3

[HM92] John Hannan and Dale Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992. doi:10.1017/S0960129500001559. 5.2, 6.5

[HM94] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994. doi:10.1006/inco.1994.1036. 4.6.1

[Hoa71] C. A. R. Hoare. Proof of a program: FIND. *Communications of the ACM*, 14(1):39–45, 1971. doi:10.1145/362452.362489. 4.2.7

[Hue97] Gérard Huet. Functional pearl: The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997. 3.2.1

[JK12] Jean-Baptiste Jeannin and Dexter Kozen. Computing with capsules. In *14th International Workshop on Descriptional Complexity of Formal Systems (DCFS'12)*, pages 1–19. Springer LNCS 7386, 2012. 6.5.1

[JNS05] Rada Jagadeesan, Gopalan Nadathur, and Vijay Saraswat. Testing concurrent systems: An interpretation of intuitionistic logic. In *Foundations of Software Technology and Theoretical Computer Science*, pages 517–528. Springer LNCS 3821, 2005. 4.3, 4.6.2

[Kah87] Gilles Kahn. Natural semantics. Technical Report 601, INRIA, February 1987. 5.1

[KKS11] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-Chieh Shan. Shifting the stage: Staging with delimited control. *Journal of Functional Programming*, 21(6):617–662, 2011. doi:10.1017/S0956796811000256. 6.6.3

[Kri09] Neelakantan R. Krishnaswami. Focusing on pattern matching. In *Principles of Programming Languages (POPL'09)*, pages 366–378. ACM, 2009. 3.3.3

[Lam58] Joachim Lambek. The mathematics of sentence structure. *American Mathematical Monthly*, 65(3):154–170, 1958. URL: http://www.jstor.org/stable/2310058. 1.1, 3, 3.1

[Lan64] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964. URL: 10.1093/comjnl/6.4.308. 6

[Lau02] Olivier Laurent. *Étude de la polarisation en logique*. PhD thesis, Université de la Méditerranée - Aix-Marseille II, 2002. 3.3.2

[Lau04] Olivier Laurent. A proof of the focalization property of linear logic, May 2004. Unpublished note, available online: http://perso.ens-lyon.fr/olivier.laurent/llfoc.pdf. 2.3.3

[LCH07] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of Standard ML. In *Principles of Programming Languages (POPL'07)*, pages 173–184. ACM, 2007. 1

[Lev04] Paul B. Levy. *Call-by-push-value. A functional/imperative synthesis*. Semantic Structures in Computation. Springer, 2004. 2.3.2

[LG09] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009. doi:10.1016/j.ic.2007.12.004. 5.2, 2, 6.4

[LM09] Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logic. *Theoretical Computer Science*, 410(46):4747–4768, 2009. doi:10.1016/j.tcs.2009.07.041. 2.3.2, 2.3.8

[LPPW05] Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In *Principles and Practice of Declarative Programming (PPDP'05)*, pages 35–46. ACM, 2005. 4.6, 4.6.2, 4.7.3, 8.1

[LZH08] Daniel R. Licata, Noam Zeilberger, and Robert Harper. Focusing on binding and computation. In *Proceedings of the 23rd Annual Symposium on Logic in Computer Science (LICS'08)*, pages 241–252, Pittsburgh, Pennsylvania, 2008. 3.3.3

[Coq10] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2010. Version 8.3. URL: http://coq.inria.fr. 1

[MC12] Chris Martens and Karl Crary. LF in LF: Mechanizing the metatheory of LF in Twelf. In *Proceedings of the International Workshop on Logical Frameworks and Meta-Langauges: Theory and Practice (LFMTP'12)*, Copenhagen, Denmark, 2012. 2.2, 3.8

[McA02] David A. McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537, 2002. doi:10.1145/581771.581774. 8.1

[Mig10] Matthew Might. Abstract interpreters for free. In *Static Analysis Symposium (SAS'10)*, pages 407–421. Springer LNCS 6337, 2010. 8.6

[Mil93] Dale Miller. The π-calculus as a theory in linear logic: preliminary results. In *Extensions of Logic Programming (ELP'92)*, pages 242–264. Springer LNCS 660, 1993. 1.1, A

[Mil96] Dale Miller. Forum: A multiple-conclusion specification logic. *Theoretical Computer Science*, 165(1):201–232, 1996. doi:10.1016/0304-3975(96)00045-X. 1.1

[Mil02] Dale Miller. Higher-order quantification and proof search. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology and Software Technology (AMAST'02)*, pages 60–75. Springer LNCS 2422, 2002. 5.2

[Mil09] Dale Miller. Formalizing operational semantic specifications in logic. *Electronic Notes in Theoretical Computer Science*, 246:147–165, 2009. Proceedings of WFLP 2008. doi:10.1016/j.entcs.2009.07.020. 2.1

[ML96] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996. 2.5.3

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51(1–2):125–157, 1991. doi:10.1016/0168-0072(91)90068-W. 4.6.1

[Mor95] Glyn Morrill. Higher-order linear logic programming of categorial deduction. In *Proceedings of the Meeting of the European Chapter of the Association for Computational Linguistics*, pages 133–140, 1995. 5.2

[Mos04] Peter D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:195–228, 2004. doi:10.1016/j.jlap.2004.03.008. 6.5.5

[MP92] Spiro Michaylov and Frank Pfenning. Natural semantics and some of its metatheory in elf. In *Extensions of Logic Programming (ELP'91)*, pages 299–344. Springer LNCS 596, 1992. 5.2

[MP01] Richard Moot and Mario Piazza. Linguistic applications of first order intuitionistic

linear logic. *Journal of Logic, Language and Information*, 10(2):211–232, 2001. doi:10.1023/A:1008399708659. 5.2

[MP09] Sean McLaughlin and Frank Pfenning. Efficient intuitionistic theorem proving with the polarized inverse method. In R.A. Schmidt, editor, *Proceedings of the 22nd International Conference on Automated Deduction (CADE-22)*, pages 230–244. Springer LNAI 5663, 2009. 2.3.2

[MR96] Luc Moreau and Daniel Ribbens. The semantics of pcall and fork in the presence of first-class continuations and side-effects. In *Parallel Symbolic Languages and Systems (PSLS'95)*, pages 53–77. Springer LNCS 1068, 1996. 7.2, 9.6

[MSV10] Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In *Programming Language Design and Implementation (PLDI'10)*, pages 305–315. ACM, 2010. 8.6

[MT10] Paul-André Melliès and Nicholas Tabareau. Resource modalities in tensor logic. *Annals of Pure and Applied Logic*, 161(5):632–653, 2010. doi:10.1016/j.apal.2009.07.018. 2.5.2

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML*. MIT Press, 1997. B.1

[Mur89] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989. doi:10.1109/5.24143. 4.3

[Mur08] Tom Murphy VII. *Modal Types for Mobile Code*. PhD thesis, Carnegie Mellon University, 2008. 5.2

[NM09] Vivek Nigam and Dale Miller. Algorithmic specifications in linear logic with subexponentials. In *Principles and Practice of Declarative Programming (PPDP'09)*, pages 129–140. ACM, 2009. 3, 3.2

[NNH05] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 2005. 8.4, 8.4.4

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007. 1

[NPP08] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *Transactions on Computational Logic*, 9(3), 2008. doi:10.1145/1352582.1352591. 3.1.1, 4.1, 4.1.3

[PD01] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications (IMLA'99)*, Trento, Italy, July 1999. doi:10.1017/S0960129501003322. 3.1

[Pfe89] Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Proceedings of the 4th Annual Symposium on Logic in Computer Science (LICS'89)*, pages 313–322, Pacific Grove, California, 1989. 6.1.1

[Pfe00] Frank Pfenning. Structural cut elimination 1. Intuitionistic and classical logic. *Information and Computation*, 157(1–2):84–141, 2000. doi:10.1006/inco.1999.2832. 2.3.3, 2.3.6, 2.3.6, 2.3.8, 3.4

[Pfe04] Frank Pfenning. Substructural operational semantics and linear destination-passing style. In *Programming Languages and Systems*, page 196. Springer LNCS 3302, 2004. Abstract of invited talk. 1.2, 2.1, 5, 5.1, 6.2.2, 7.2

[Pfe08] Frank Pfenning. Church and Curry: Combining intrinsic and extrinsic typing. In C.Benzmüller, C.Brown, J.Siekmann, and R.Statman, editors, *Reasoning in Simple Type Theory: Festschrift in Honor of Peter B. Andrews on His 70th Birthday*, volume 17 of *Studies in Logic*. College Publications, 2008. 3.3.3, 4.1.2, 4.7.3

[Pfe12a] Frank Pfenning. Lecture notes on backtracking, February 2012. Lecture notes for 15-819K: Logic Programming at Carnegie Mellon University, available online: http://www.cs.cmu.edu/~fp/courses/lp/lectures/lp-all.pdf. 3

[Pfe12b] Frank Pfenning. Lecture notes on backward chaining, March 2012. Lecture notes for 15-816: Linear Logic at Carnegie Mellon University, available online: http://www.cs.cmu.edu/~fp/courses/15816-s12/Lectures/17-bwdchaining.pdf. 6.6.1

[Pfe12c] Frank Pfenning. Lecture notes on chaining and focusing, February 2012. Lecture notes for 15-816: Linear Logic at Carnegie Mellon University, available online: http://www.cs.cmu.edu/~fp/courses/15816-s12/Lectures/09-focusing.pdf. 2.3.3, 2.3.8, 2.5, 9

[Pfe12d] Frank Pfenning. Lecture notes on ordered forward chaining, March 2012. Lecture notes for 15-816: Linear Logic at Carnegie Mellon University, available online: http://www.cs.cmu.edu/~fp/courses/15816-s12/Lectures/16-ordchain.pdf. 6.6.1

[Pfe12e] Frank Pfenning. Lecture notes on substructural operational semantics, February 2012. Lecture notes for 15-816: Linear Logic at Carnegie Mellon University, available online: http://www.cs.cmu.edu/~fp/courses/15816-s12/Lectures/12-ssos.pdf. 5.1

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. 9.4

[Plo77] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, 1977. doi:10.1016/0304-3975(77)90044-5. B.1

[Plo04] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004. Reprinted with corrections from Aarhus University technical report DAIMI FN-19. doi:10.1016/j.jlap.2004.05.001. 2, 5.1, 6, 6.6.2

[Pol00] Jeff Polakow. Linear logic programming with ordered contexts. In *Principles and Practice of Declarative Programming (PPDP'00)*, pages 68–79. ACM, 2000. 4.6.1

[Pol01] Jeff Polakow. *Ordered Linear Logic and Applications*. PhD thesis, Carnegie Mellon University, 2001. 3.1, 4.1, 4.4, 4.6.1, 9.1.3

[PP99] Jeff Polakow and Frank Pfenning. Natural deduction for intuitionistic non-commutative linear logic. In *Typed Lambda Calculus and Applications (TLCA'99)*, pages 295–309. Springer LNCS 1581, 1999. 3

[PS99a] Frank Pfenning and Carsten Schürmann. Algorithms for equality and unification in the presence of notational definitions. In T. Altenkirch, B. Reus, and W. Naraschewski, editors, *Types for Proofs and Programs (TYPES'98)*, pages 179–193. Springer LNCS 1657, 1999. 4.2.1

[PS99b] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206. Springer LNAI 1632, 1999. 1, 5.2

[PS03] Adam Poswolsky and Carsten Schürmann. Factoring report. Technical Report YALEU/DCS/TR-1256, Department of Computer Science, Yale University, November 2003. 6.4.2, 6.4.3

[PS09] Frank Pfenning and Robert J. Simmons. Substructural operational semantics as ordered logic programming. In *Proceedings of the 24th Annual Symposium on Logic in Computer Science (LICS'09)*, pages 101–110, Los Angeles, California, 2009. 1.2, 2.1, 2.5, 2.5.2, 2.5.3, 3.6.1, 4.7.3, 5, 5.1, 6.5, 6.5.3, 7.2, 7.2.2, 10.2

[Pym02] David J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*. Applied Logic Series. Springer, 2002. 3, 3.2

[Ree09a] Jason Reed. *A Hybrid Logical Framework*. PhD thesis, Carnegie Mellon University, 2009. 4.1, 4.1.2, 4.7.3, 5.2, 10

[Ree09b] Jason Reed. A judgmental deconstruction of modal logic, January 2009. Unpublished note, available online: http://www.cs.cmu.edu/~jcreed/papers/jdml.pdf. 2.5.3

[Ree09c] Jason Reed. Queue logic: An undisplayiable logic?, April 2009. Unpublished note, available online: http://www.cs.cmu.edu/~jcreed/papers/queuelogic.pdf. 3.8

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*. ACM, 1972. 5.2

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual Symposium on Logic in Computer Science (LICS'02)*, pages 55–74, 2002. 4.2.8

[RŞ10] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010. doi:10.1016/j.jlap.2010.03.012. 1

[Sau08] Alexis Saurin. Towards ludics programming: Interactive proof search. In *Proceedings of the International Conference on Logic Programming (ICLP'08)*, pages

253–268. Springer LNCS 5366, 2008. 4.6.2

[Sch00] Carsten Schürmann. *Automating the Meta-Theory of Deductive Systems*. PhD thesis, Carnegie Mellon University, 2000. 9, 9.1, 9.1.1

[SH93] Peter Schroeder-Heister. Rules of definitional reflection. In *Proceedings of the 8th Annual Symposium on Logic and Computer Science (LICS'93)*, pages 222–232, Montreal, Canada, 1993. 3.1.1

[Shi88] Olin Shivers. Control flow analysis in scheme. In *Programming Language Design and Implementation (PLDI'88)*, pages 164–174. ACM, 1988. 8.4

[Sim09] Robert J. Simmons. Linear functions and coverage checking stuff with holes in it, December 2009. Unpublished note, available online: http://www.cs.cmu.edu/~rjsimmon/papers/rfl6-cover.pdf. 3, 3.2.1

[Sim11] Robert J. Simmons. Structural focalization. *CoRR*, abs/1109.6273, 2011. 2.3, 2.3.4

[SN07] Anders Schack-Nielsen. Induction on concurrent terms. In *Proceedings of the International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'07)*, pages 37–51, Bremen, Germany, 2007. 4.4, 5.1, 6.1

[SN11] Anders Schack-Nielsen. *Implementing Substructural Logical Frameworks*. PhD thesis, IT University of Copenhagen, 2011. 4, 4.6

[SNS08] Anders Schack-Nielsen and Carsten Schürmann. Celf — a logical framework for deductive and concurrent systems (system description). In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR'08)*, pages 320–326. Springer LNCS 5195, 2008. 3.2, 4.6, 4.7.3, B

[SNS10] Andres Schack-Nielsen and Carsten Schürmann. Curry-style explicit substitutions for the linear and affine lambda calculus. In J. Gisel and R. Hähnle, editors, *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR'10)*, pages 1–14. Springer LNCS 6173, 2010. 3.8

[SP08] Robert J. Simmons and Frank Pfenning. Linear logical algorithms. In *Proceedings of the International Colloquium on Automata, Languages and Programming, Track B (ICALP'08)*, pages 336–347. Springer LNCS 5126, 2008. 3.6.1, 8.1, 10.1

[SP11a] Robert J. Simmons and Frank Pfenning. Logical approximation for program analysis. *Higher-Order and Symbolic Computation*, 24(1–2):41–80, 2011. doi:10.1007/s10990-011-9071-2. 1.2, 2.5.3, 5.1, 6.5.3, 7, 7, 7.1, 1, 7.1, 7.2, 8, 8.3, 8.3, 8.4.2

[SP11b] Robert J. Simmons and Frank Pfenning. Weak focusing for ordered linear logic. Technical Report CMU-CS-2011-147, Department of Computer Science, Carnegie Mellon University, April 2011. 6, 2.5.3, 3.1.1

[TCP12] Bernardo Toninho, Luis Caires, and Frank Pfenning. Functions as session-typed processes. In *Foundations of Software Science and Computational Structures (FOSSACS'12)*, pages 346–360. Springer LNCS 7213, 2012. 6.5.3

[vB91] J. van Benthem. *Language in Action: Categories, Lambdas and Dynamic Logic*, volume 130 of *Studies in Logic and the Foundations of Mathematics*, chapter 16,

pages 225–250. North-Holland, Amsterdam, 1991. 5.2

[Vir99] Roberto Virga. *Higher-Order Rewriting with Dependent Types*. PhD thesis, Carnegie Mellon University, 1999. 4.1.3

[VM07] David Van Horn and Harry G. Mairson. Relating complexity and precision in control flow analysis. In *International Conference on Functional Programming (ICFP'07)*, pages 85–96. ACM, 2007. 8.6

[VM10] David Van Horn and Matthew Might. Abstracting abstract machines. In *International Conference on Functional Programming (ICFP'10)*, pages 51–62. ACM, 2010. 8.6

[WCPW02] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-2002-101, Department of Computer Science, Carnegie Mellon University, March 2002. Revised May 2003. 1.1, 1.1, 1.4, 2.2, 3.1, 3.3.3, 3.4.2, 3.8, 4, 4.1, 4.3

[Zei08a] Noam Zeilberger. Focusing and higher-order abstract syntax. In *Principles of Programming Languages*, pages 359–369. ACM, 2008. 3.1.1

[Zei08b] Noam Zeilberger. On the unity of duality. *Annals of Pure and Applied Logic*, 157(1–3):66–96, 2008. doi:10.1016/j.apal.2008.01.001. 2.3.2, 10