Logical frameworks for specifying and reasoning about stateful and concurrent languages*

Robert J. Simmons

July 5, 2010

Abstract

Substructural logics, such as linear logic and ordered logic, have an inherent notion of state and state change. This makes them a natural choice for developing logical frameworks that specify evolving stateful systems. Our previous work has shown that the so-called *forward reasoning* fragment of ordered linear logic can be used to give clear, concise, and modular specifications of stateful and concurrent features of programming languages. I propose to show that a logical framework based on forward reasoning in ordered linear logic can also be used to formally reason about properties of programming languages in ways that can be verified by both human readers and mechanized proof assistants.

1 Introduction

Robust and flexible frameworks for specifying and reasoning about programming languages form the bedrock of programming languages research. In part, this is because they provide a common convention: historically, the most successful specification framework, Structural Operational Semantics (i.e. SOS), is the *lingua franca* of PL research. Even more critical is the ability of a framework to permit reasoning about properties of programming languages; in the case of SOS, the traditionally interrelated notion of "Safety = Progress + Preservation" has essentially come to define what it means for a programming language to make sense.

One problem with SOS specifications of programming languages is that they are *non-modular*: especially when dealing with imperative and concurrent programming language features, the natural specification of language *parts* cannot, in general, be combined to give a natural specification of the language *whole*. This makes it difficult to straightforwardly extend a pure "toy" programming language with a feature like imperative references without a complete overhaul of the language's definition and type safety proof; a similar overhaul is necessary to extend the definition of a large sequential language to handle concurrency. This has troubling practical consequences: it means that it is more difficult to scale from a small language formalization to a large one, and it means that existing language specifications may be difficult to augment with new features.

The solution at the heart of this thesis involves two interconnected ideas. One idea is a *specification style*, Substructural Operational Semantics (i.e. SSOS). SSOS specifications generalize abstract machine specifications and permit clear and modular specification of programming language features that involve state and concurrency. The other idea is a *logical framework* based on forward reasoning in ordered linear logic. The framework I present can adequately encode both the static and dynamic

Subversion Rev : 1185

semantics of SSOS specifications, and the framework also facilitates formal reasoning about properties of specified programming languages. Furthermore, a logic programming interpretation of the logical framework can be used to simulate the dynamic semantics of SSOS specifications.

Proposed work

My thesis work will defend the following statement:

> **Thesis Statement:** *Logical frameworks based on forward reasoning in substructural logics are suitable for modular specification of programming languages and formal reasoning about their properties.*

The overall structure of this proposal is as follows: Section 2 discusses the problem of non-modular programming language specification. Section 3 motivates a logical basis for evolving systems, and Section 4 sketches a logical framework based on these principles. Section 5 returns to the specification problem motivated in Section 2, presenting modular SSOS specifications in this new logical framework and reasoning about type safety via progress and preservation proofs. Section 6 discusses a number of other applications of the framework and specification style; these applications generally fall under the umbrella of applying logical transformations to SSOS specifications.

In the remainder of this section, I outline my existing and expected contributions in more detail.

A logical framework for evolving systems The logical framework that I plan to develop and use is organized around a presentation of logic as a *state transition system*. This is outlined in Section 3, and a complete account of intuitionistic linear logic as a state transition system is an expected contributions of my thesis work. When we want to specify a system with a notion of evolution or state transition — such as a programming language's dynamic semantics — we can encode transitions in the specified system as transitions in the logical framework. Embedded within this state-transition-based logic is a more traditional logical framework organized around a canonical-forms-based presentation of logic. When we want to specify a deductive system that is defined by inference rules — such as the typing rules of a programming language — we can encode derivations in the specified system as canonical proof terms in the logical framework.

The logical framework discussed in Section 4 is in many ways quite similar to existing systems, especially CLF [WCPW02]. The critical difference is that the framework I present is based upon the presentation of substructural logic as a state transition system; this change of perspective becomes critical when describing and verifying properties of these programming languages. There are a number of other differences, most notably the generalization from linear logic to ordered logic and the strong separation between transitions and deductions, though I see these differences as somewhat less fundamental. As part of my thesis work I will present a full development of the standard metatheory of this new logical framework; I anticipate that this will be straightforward, as many of the details of the framework are familiar.

Canonical-forms-based presentations of logic are usually given a logic programming semantics that is backward chaining (or "top-down"). The state transition-based logic, on the other hand, naturally corresponds to a forward chaining (or "bottom-up") logic programming semantics. The particular relationship between the state-transition-based fragment of the logic and the canonical-forms-based fragment of the logic, then, corresponds to a particular way of understanding the integration of forward and backward chaining in a logic programming language. This logic programming interpretation is useful for our purposes, because it allows a SSOS specification of a programming language to be directly executed as an interpreter for the specified programming language. An existing contribution is Ollibot, an implementation of the forward chaining fragment of the logical

framework given in Section 4.[1] I expect to extend Ollibot with facilities for backwards chaining in the course of this thesis; I also plan to explore efficient and distributed execution of Ollibot programs.

Substructural Operational Semantics (SSOS) specifications While I am interested generally in the specification of evolving systems in substructural logics, my focus will be on the aforementioned SSOS specifications. SSOS specifications are similar to specifications of abstract machines for control, but the specifications are *local*: transition rules do not describe the entire state, just the parts that change. The particular style of SSOS specifications that I plan to focus on, originally presented in [PS09], is outlined in Section 5. The style of specification considered in this section gives clean and modular specifications of stateful and concurrent programming language features; its most significant limitation is that it does not handle non-local control features like first-class continuations in a modular way.

In Section 5, I also outline a methodology for describing the static semantics of SSOS specifications and reasoning about their safety, which is an original contribution of this thesis proposal. As demonstrated in Section 5.6 and the appendix, this methodology is still incomplete; however, I expect to show that the techniques in this section can be adapted to allow for both modular specifications of a language's semantics and modular proofs of a language's safety. I also expect to develop a tool to facilitate the mechanical verification of these type safety properties.

In Section 6, I discuss a number of further applications, all of which fall under the general theme of transforming SSOS specifications. My previous work in this area has been primarily focused on applying logical transformations to derive manifestly correct program analyses from the operational semantics of a programming language specified in SSOS [SP09, SP10]. I also plan to consider the use of logical transformations for other of other purposes, including enhancing the modularity of specifications, introducing alternative evaluation strategies, and connecting high-level and low-level specifications.

2 Modular and non-modular specification

In the introduction and thesis statement, the phrase *modular specification* was used but not defined. Modularity means different things in different settings; in this section, we will motivate the problem by considering modular and non-modular extensions to a very simple programming language. We start by assuming that we have natural numbers n and some operations (such as addition) on those natural numbers; expressions e can then be defined by giving a BNF specification:

$$e ::= \mathsf{num}(n) \mid e_1 + e_2$$

Call this language of numbers and addition $\mathcal{L}_0$; we next give a small-step structural operational semantics for $\mathcal{L}_0$ that specifies a left-to-right evaluation order. We also adopt the standard convention of writing expressions that are known to be values as v, not e:

$$\frac{}{\mathsf{num}(n_1)\,\mathsf{value}} \quad \frac{e_1 \mapsto e_1'}{e_1 + e_2 \mapsto e_1' + e_2} \quad \frac{v_1\,\mathsf{value} \quad e_2 \mapsto e_2'}{v_1 + e_2 \mapsto v_1 + e_2'} \quad \frac{n_1 + n_2 = n_3}{\mathsf{num}(n_1) + \mathsf{num}(n_2) \mapsto \mathsf{num}(n_3)}$$

The extension of $\mathcal{L}_0$ with eager pairs ($e ::= \ldots \mid \pi_1 e \mid \pi_2 e \mid \langle e_1, e_2 \rangle$) is an example of a modular extension. We can take the following natural small-step SOS specification of eager pairs and concatenate it together with the $\mathcal{L}_0$ specification to form a coherent (and, with a suitable type system, provably type-safe) specification of a language that has both features.

$$\frac{v_1\,\mathsf{value} \quad v_2\,\mathsf{value}}{\langle v_1, v_2 \rangle\,\mathsf{value}} \quad \frac{e_1 \mapsto e_1'}{\langle e_1, e_2 \rangle \mapsto \langle e_1', e_2 \rangle} \quad \frac{v_1\,\mathsf{value} \quad e_2 \mapsto e_2'}{\langle v_1, e_2 \rangle \mapsto \langle v_1, e_2' \rangle}$$

[1] http://ollibot.hyperkind.org/examples-0.1/

$$\frac{e \mapsto e'}{\pi_1 e \mapsto \pi_1 e'} \quad \frac{\langle v_1, v_2\rangle \,\mathsf{value}}{\pi_1\langle v_1, v_2\rangle \mapsto v_1} \quad \frac{e \mapsto e'}{\pi_2 e \mapsto \pi_2 e'} \quad \frac{\langle v_1, v_2\rangle \,\mathsf{value}}{\pi_2\langle v_1, v_2\rangle \mapsto v_2}$$

In contrast, the extension of $\mathcal{L}_0$ with ML-style imperative references is an obvious example of a non-modular extension. The extended syntax is $(e ::= \ldots \mid \mathsf{ref}\, e \mid \mathsf{loc}[l] \mid\, !e \mid e_1 \,\mathsf{gets}\, e_2)$ — the expression $\mathsf{ref}\, e$ creates a reference, $!e$ dereferences a reference, $e_1 \,\mathsf{gets}\, e_2$ assigns the value of e_2 to the location represented by e_1, and $\mathsf{loc}[l]$ is a value, which only exists at runtime, referencing the abstract heap location l where some value is stored). In order to give a small-step SOS specification for the extended language, we must mention a store σ in every rule:

$$\frac{}{\mathsf{loc}[l]\,\mathsf{value}} \quad \frac{(\sigma, e) \mapsto (\sigma', e')}{(\sigma, \mathsf{ref}\, e) \mapsto (\sigma', \mathsf{ref}\, e')} \quad \frac{v\,\mathsf{value} \quad l \notin \mathrm{dom}(\sigma)}{(\sigma, \mathsf{ref}\, v) \mapsto (\sigma[l := v], \mathsf{loc}[l])}$$

$$\frac{(\sigma, e) \mapsto (\sigma', e')}{(\sigma, !e) \mapsto (\sigma', !e')} \quad \frac{\sigma(l) = v}{(\sigma, !\,\mathsf{loc}[l]) \mapsto (\sigma, v)}$$

$$\frac{(\sigma, e_1) \mapsto (\sigma', e_1')}{(\sigma, e_1 \,\mathsf{gets}\, e_2) \mapsto (\sigma', e_1' \,\mathsf{gets}\, e_2)} \quad \frac{v_1\,\mathsf{value} \quad (\sigma, e_2) \mapsto (\sigma', e_2')}{(\sigma, v_1 \,\mathsf{gets}\, e_2) \mapsto (\sigma', v_1 \,\mathsf{gets}\, e_2')}$$

$$\frac{v_2\,\mathsf{value}}{(\sigma, \mathsf{loc}[l] \,\mathsf{gets}\, v_2) \mapsto (\sigma[l := v_2], v_2)}$$

The non-modularity of this extension comes from the fact that it forces us to rewrite the rules describing addition as well:

$$\frac{(\sigma, e_1) \mapsto (\sigma', e_1')}{(\sigma, e_1 + e_2) \mapsto (\sigma', e_1' + e_2)}$$

$$\frac{v_1\,\mathsf{value} \quad (\sigma, e_2) \mapsto (\sigma, e_2')}{(\sigma, v_1 + e_2) \mapsto (\sigma', v_1 + e_2')} \quad \frac{n_1 + n_2 = n_3}{(\sigma, \mathsf{num}(n_1) + \mathsf{num}(n_2)) \mapsto (\sigma, \mathsf{num}(n_3))}$$

Modularity and specification styles Another interesting loss of modularity in SOS specifications appears when we introduce exceptions and exception handling $(e ::= \ldots \mid \mathsf{error} \mid \mathsf{try}\, e_1 \,\mathsf{ow}\, e_2)$.

$$\frac{e_1 \mapsto e_2}{\mathsf{try}\, e_1 \,\mathsf{ow}\, e_2 \mapsto \mathsf{try}\, e_1' \,\mathsf{ow}\, e_2} \quad \frac{v_1\,\mathsf{value}}{\mathsf{try}\, v_1 \,\mathsf{ow}\, e_2 \mapsto v_1} \quad \frac{}{\mathsf{try}\,\mathsf{error}\,\mathsf{ow}\, e_2 \mapsto e_2}$$

Our rules for addition do not need to be revised to be compatible with this extension, but in order to preserve type safety, we must provide a way for errors to "bubble up" through additions.

$$\frac{}{\mathsf{error} + e_2 \mapsto \mathsf{error}} \quad \frac{v_1\,\mathsf{value}}{v_1 + \mathsf{error} \mapsto \mathsf{error}}$$

We can avoid this particular form of non-modularity while technically staying within the SOS framework. This is possible if we use an *abstract machine* style of specification. Abstract machine-style specifications have an explicit control stack (or continuation), usually denoted k, that represents unfinished parts of the computation; a control stack is a list of *frames*, which we describe below. Basic abstract machine specifications have two kinds of states: an expression being evaluated on a stack $(k \rhd e)$ and a value being returned to the stack $(k \lhd v)$. We introduce two frames f in order to specify $\mathcal{L}_0$:

- $((-) + e_2)$ is a frame waiting on the first argument to be evaluated to a value so that the evaluation of e_2 can begin, and

- $(v_1 + (-))$ is a frame holding the already-evaluated value v_1 waiting on the second argument to be evaluated to a value.

The abstract machine specification of the operational semantics of $\mathcal{L}_0$ has four rules:

$$\begin{aligned}
k \rhd \mathsf{num}(n) &\mapsto k \lhd \mathsf{num}(n) \\
k \rhd (e_1 + e_2) &\mapsto (k, (-) + e_2) \rhd e_1 \\
(k, (-) + e_2) \lhd v_1 &\mapsto (k, v_1 + (-)) \rhd e_2 \\
(k, \mathsf{num}(n_1) + (-)) \lhd \mathsf{num}(n_2) &\mapsto k \lhd \mathsf{num}(n_3) \qquad (\text{if } n_1 + n_2 = n_3)
\end{aligned}$$

Given this specification, we can define the behavior of exceptions by adding a new frame $(\mathsf{try}\,(-)\,\mathsf{ow}\,e_2)$ and a new kind of state — $(k \blacktriangleleft)$, a stack dealing with an error.

$$\begin{aligned}
k \rhd \mathsf{try}\,e_1\,\mathsf{ow}\,e_2 &\mapsto (k, \mathsf{try}\,(-)\,\mathsf{ow}\,e_2) \rhd e_1 \\
(k, \mathsf{try}\,(-)\,\mathsf{ow}\,e_2) \lhd v_1 &\mapsto k \lhd v_1 \\
k \rhd \mathsf{error} &\mapsto k \blacktriangleleft \\
(k, \mathsf{try}\,(-)\,\mathsf{ow}\,e_2) \blacktriangleleft &\mapsto k \rhd e_2 \\
(k, f) \blacktriangleleft &\mapsto k \blacktriangleleft \qquad (\text{if } f \neq \mathsf{try}\,(-)\,\mathsf{ow}\,e_2)
\end{aligned}$$

When it comes to control features like exceptions, abstract machine specifications are more modular. We did not have to specifically consider or modify the stack frames for addition (or for multiplication, or function application...) in order to introduce exceptions and exception handling; one rule (the last one above) interacts modularly with essentially any "pure" language features. Both the SOS specification and the abstract machine specification were reasonable ways of specifying pure features like addition, but the abstract machine specification better anticipates the addition of control features.

Observation 1. *Different styles of specification can allow different varieties of programming language features to be specified in a modular style.*

Another way of stating this observation is that we must *precisely* anticipate the language features dealing with state and control if SOS-style specifications are our only option. When we pull back and look at a choice between these two specification styles, we only need to *generally* anticipate whether we might need control features like exceptions or first-class continuations; if so, we should use an abstract machine specification.

Modularity and ambient state Of course, an abstract-machine-style presentation does not solve the problem with mutable references that we discussed at first! Various attempts have been made to address this problem. The Definition of Standard ML used an ad-hoc convention that, applied to our example, would allow us to write the original $\mathcal{L}_0$ rules while actually meaning that we were writing the state-annotated version of the rules [MTHM97, p. 40]. In this case, the convention is less about modular specification and more about not writing hundreds and hundreds of basically-obvious symbols in the language definition.

Mosses's Modular Structural Operational Semantics (MSOS) attempts to formalize and standardize this sort of convention [Mos04]. A transition in MSOS is written as $(e_1 \xrightarrow{X} e_2)$, where X is an open-ended description of the ambient state. By annotating congruence rules with $X = \{\ldots\}$, the MSOS specification of $\mathcal{L}_0$ specifies that the congruence rules simply pass on whatever effects happen in subexpressions. The reduction rule is annotated with $X = \{-\}$ to indicate that it has no effect

on the ambient state.

$$\frac{v_1\ \mathsf{value} \quad e_2 \xrightarrow{\{\ldots\}} e_2'}{v_1 + e_2 \xrightarrow{\{\ldots\}} v_1 + e_2'} \qquad \frac{n_1 + n_2 = n_3}{\mathsf{num}(n_1) + \mathsf{num}(n_2) \xrightarrow{\{-\}} \mathsf{num}(n_3)}$$

Reduction rules that actually access some part of the ambient state, such as reduction rules for referencing the store, can mention and manipulate the store as necessary:

$$\frac{\sigma(l) = v}{!\,\mathsf{loc}(l) \xrightarrow{\{\sigma,\ -\}} v} \qquad \frac{v_2\ \mathsf{value}}{\mathsf{loc}(l)\ \mathsf{gets}\ v_2 \xrightarrow{\{\sigma,\ \sigma' = \sigma[l\ :=\ v_2],\ -\}} v_2}$$

There is a good bit of related work along the lines of MSOS; for example, Implicitly-Modular Structural Operational Semantics (I-MSOS) is an extension of MSOS that partially removes the need for the explicit annotations X [MN09]. In all cases, however, the goal is to provide a regular notion of ambient state that allows each part of a language specification to leave irrelevant parts of the state implicit and open-ended.

Observation 2. *A framework that provides an open-ended notion of ambient state assists in writing modular programming language specifications.*

Modularity and the LF context Logical frameworks in the LF family (such as LF [HHP93], LLF [CP02], OLF [Pol01], CLF [WCPW02], and HLF [Ree09a]) have an inherent notion of ambient information provided by the framework's *context*. As an example, consider the following LF specification of the static semantics of $\mathcal{L}_0$ (we use the usual convention in which capital letters are understood to be universally quantified):

$\mathsf{of} : \mathsf{exp} \to \mathsf{tp} \to \mathsf{type}.$
$\mathsf{of/num} : \mathsf{of}\ (\mathsf{num}\ \mathsf{N})\ \mathsf{int}.$
$\mathsf{of/plus} : \mathsf{of}\ \mathsf{E}_1\ \mathsf{int} \to \mathsf{of}\ \mathsf{E}_2\ \mathsf{int} \to \mathsf{of}\ (\mathsf{plus}\ \mathsf{E}_1\ \mathsf{E}_2)\ \mathsf{int}.$

The contents of the LF context are specified independently from the specification of the inference rules. If we declare the LF context to be empty, our specification corresponds to an "on-paper" specification of the language's static semantics that looks like this:

$$\frac{}{\mathsf{num}(N) : \mathsf{int}} \qquad \frac{e_1 : \mathsf{int} \quad e_2 : \mathsf{int}}{e_1 + e_2 : \mathsf{int}}$$

This assumption that there is nothing interesting in the context, frequently called a *closed world assumption*, is not the only option. Another option is to declare that the LF context may include free expression variables x in tandem with an assumption that $x{:}\tau$ for some type τ. In this case, the LF specification written above corresponds to the following on-paper specification:

$$\frac{}{\Gamma, x{:}\tau \vdash x : \tau} \qquad \frac{}{\Gamma \vdash \mathsf{num}(n) : \mathsf{int}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}}$$

Language features that incorporate binding require a variable context in the specification of their static semantics; as a result, our original on-paper static semantics could not be composed with an on-paper static semantics of language features like let-expressions and first-class functions. But because the contents of the LF context are open-ended, we *can* combine the LF specifications of the static semantics. The specification we wrote down in LF is open-ended in precisely the way we need it to be, even though the on-paper version is not.

Observation 3. *The LF context provides an open-ended notion of ambient information that enhances the modularity of certain specifications.*

Unfortunately, the open-endedness we describe here is useful for capturing ambient *information* in a static semantics; it is not useful for capturing ambient *state* in a dynamic semantics. The LF context only supports the addition of new assumptions that are persistent and unchanging, but capturing the dynamic semantics of stateful features like mutable references requires adding, modifying, and removing information about the state of the specified system. The solution is, in principle, well understood: logical frameworks based on substructural logics (especially linear logic) can capture stateful change — it is possible to remove facts from a linear logic context. Therefore, assumptions in a linear logic context can be thought of as representing *ephemeral* facts about the state of a system, facts which might change, as opposed to persistent assumptions which can never be removed or modified. Examples of logical frameworks that provide an open-ended notion of ambient state through the use of a linear context include Forum [Mil96], Lolli [HM94], LLF [CP02], and CLF [WCPW02]. Furthermore, all of these frameworks have been given an operational interpretation as logic programming languages, so specifications are, at least in theory, executable.

Previewing SSOS specifications These observations motivate the design of substructural operational semantics and, in turn, the design of the logical framework. We can give a basic picture of SSOS specifications at this point by thinking describing them as collections of string rewriting rules where, for instance, the rewrite rule $\mathsf{a} \cdot \mathsf{b} \twoheadrightarrow \mathsf{c}$ allows the string $\mathsf{a} \cdot \mathsf{a} \cdot \mathsf{b} \cdot \mathsf{b}$ to transition to the string $\mathsf{a} \cdot \mathsf{c} \cdot \mathsf{b}$.

SSOS specifications resemble the abstract machine-style specifications above, but instead of capturing the entire control stack k as a single piece of syntax, each frame f is captured by a token $\mathsf{cont}(f)$ in the string. The token $\mathsf{eval}(e)$ represents an evaluating expression e, and is analogous to the state $(k \rhd e)$ in the abstract machine specification, whereas the token $\mathsf{retn}(v)$ represents a returned value and is analogous to the state $(k \lhd v)$ in the abstract machine specification. The resulting rules are quite similar to the abstract machine rules:

$$\begin{aligned}
\mathsf{eval}(\mathsf{num}(n)) &\twoheadrightarrow \mathsf{retn}(\mathsf{num}(n)) \\
\mathsf{eval}(e_1 + e_2) &\twoheadrightarrow \mathsf{cont}((-) + e_2) \cdot \mathsf{eval}(e_1) \\
\mathsf{cont}((-) + e_2) \cdot \mathsf{retn}(v_1) &\twoheadrightarrow \mathsf{cont}(v_1 + (-)) \cdot \mathsf{eval}(e_2) \\
\mathsf{cont}(\mathsf{num}(n_1) + (-)) \cdot \mathsf{retn}(\mathsf{num}(n_2)) &\twoheadrightarrow \mathsf{retn}(\mathsf{num}(n_3)) \qquad (\text{if } n_1 + n_2 = n_3)
\end{aligned}$$

We can then give modular specifications of exceptions and exception handling much as we did in the abstract machine for control, and the non-local nature of the specifications also means that we can add state. To give a very simple example that does not generalize well, we can modularly extend the language with a single counter; evaluating the expression get gets the current value of the counter, and evaluating the expression inc gets the current value of the counter and then increments it by one. The value of the counter is stored in a token $\mathsf{counter}(n)$ that appears to the right of the $\mathsf{eval}(e)$ or $\mathsf{retn}(v)$ tokens (whereas the stack grows off to the left).

$$\begin{aligned}
\mathsf{eval}(\mathsf{get}) \cdot \mathsf{counter}(n) &\twoheadrightarrow \mathsf{retn}(\mathsf{num}(n)) \cdot \mathsf{counter}(n) \\
\mathsf{eval}(\mathsf{inc}) \cdot \mathsf{counter}(n) &\twoheadrightarrow \mathsf{retn}(\mathsf{num}(n)) \cdot \mathsf{counter}(n+1)
\end{aligned}$$

Just as the rules need only reference the parts of the control stack that they modify, the rules only need to mention the counter if they access or modify that counter in some way. Again, this is a terrifically non-generalizable way of extending the system with state, because the use of ambient state is not "open-ended" — where would we put the *second* counter? Nevertheless, it is a specification of a stateful feature that can be composed modularly with the specification of addition.

After we motivate and present the logical framework we plan to use in the next two sections, we will return to this style of specification and see how it can be used to give modular specifications of both exceptions and parallel evaluation. A more general way of introducing state in the form of ML-style references is also possible, as discussed in Appendix A.1.

Rewriting logic The reader familiar with rewriting logic will surely notice that the specifications above are can be essentially understood as string rewriting rules. Before we proceed, it is worth discussing this connection further; more detail is considered in the conclusion.

In this proposal, SSOS specifications are closely linked to the particular ordered logical framework that I will outline in the next two sections. However, SSOS specifications need not be so closely tied to a particular framework; the specifications above are representable in either an implementation of forward chaining in ordered logic (like Ollibot) or in a rewriting logic engine (like Maude). The reason we are not exploring SSOS specifications in rewriting logic is that essential elements of our approach cannot be specified in rewriting logic — in particular, rewriting logic as it is usually understood does not support higher-order abstract syntax, nor does it support LF-style specifications of a programming language's typing judgments as presented above. Designing a common framework that unifies and generalizes both rewriting logic and the logical framework described in this proposal is a desirable goal, but it is a goal that is likely outside the scope of this thesis.

There is also a line of work on representing the semantics of programming languages within rewriting logic; the two most notable projects along these lines are the rewriting logic semantics project [MR07], and the K framework for language specifications [ŞR10]. Modulo particulars of syntax and aforementioned issues like the absence of higher-order abstract syntax, specifications in the K framework bear a resemblance to SSOS specifications, and both projects have a similar goal: using an ambient notion of state to assist in the formalization of stateful and concurrent programming language features. However, beyond that similarity the approaches differ substantially in emphasis, and I believe they can best be seen as compatible efforts: it is likely that the model checking capabilities developed in the rewriting logic semantics project can be productively applied to SSOS specifications, and likewise the techniques for logically transforming SSOS specifications and mechanically verifying their properties that I am proposing to develop should be applicable to the rewriting logic semantics project.

3 Logics of deduction and transition

The aforementioned logical frameworks that provide an open-ended notion of ambient state through the use of substructural logics — Forum, CLF, etc. — can be difficult to use for certain types of reasoning about properties of systems specified in the framework. One source of difficulty is that many properties of specified systems are naturally described as properties of the *partial* proofs that arise during proof search, but in all of these frameworks it is more natural to reason about properties of *complete* proofs. In this section, we discuss a solution to this mismatch: we define linear logic as a state transition system, at which point it is natural to talk about sequences of transitions (which do, in fact, correspond to partial proofs in a sequent calculus) as first-class entities. The only other solutions that I am aware of are based on Ludics, a presentation of logic that treats proofs with missing "leaves" as objects of consideration [Gir01].

Underlying logical frameworks like LF and LLF is a particular justification of logic, dating back to Martin-Löf's 1983 Siena Lectures [ML96], which holds that the meaning of a proposition is captured by what counts as a verification of that proposition. In Section 3.1 I briefly revisit Martin-Löf's verificationist meaning-theory and how it relates to the use of canonical forms to adequately capture deductive systems.

There is another presentation of logic — first outlined by Pfenning [Pfe08] and developed further in Section 3.2 — which underlies a line of work on using forward-chaining in substructural logics for logic programming and logical specification [SP08, SP09, PS09]. Substructural contexts are treated as descriptions of the ephemeral state of a system, and the meaning of a proposition is captured by the state transitions it generates.

The integration of these two ways of understanding logic requires care, as a naïve approach for combining the two paradigms will destroy desirable properties of both the canonical-forms-based framework and the state-transition-based framework. In Section 3.3, I explain my proposed approach, which is based on the observations of adjoint logic [Ree09b].

3.1 A canonical-forms-based view of logic

Existing work on the proof theory of logical frameworks, including the LF family of logical frameworks [HHP93], is based in part on a *verificationist* philosophy described in Martin-Löf's Siena Lectures [ML96]. A verificationist meaning-theory provides that the meaning of a proposition is precisely captured by its introduction rules — that is, by the ways in which it may be verified. A proposition $A \wedge B$ is true, then, precisely if A is true and B is true, and a proposition $A \supset B$ is true precisely if we can verify B given an additional assumption that A is true.

The elimination rules, on the other hand, must be *justified* with respect to the introduction rules. We justify the elimination rules for a connective in part by checking their *local soundness* and *local completeness* [PD01]. Local soundness ensures that the elimination rules are not too strong: if an introduction rule is immediately followed by an elimination rule, the premise gives us all the necessary evidence for the conclusion.

$$\dfrac{\dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ A\ true & B\ true\end{array}}{A \wedge B\ true}\ \wedge I}{A\ true}\ \wedge E_1 \qquad \Rightarrow_R \qquad \begin{array}{c}\mathcal{D}_1 \\ A\ true\end{array}$$

Local completeness, on the other hand, ensures that the elimination rules are not too weak: given a derivation of the compound connective, we can use elimination rules to get all the evidence necessary to reapply the introduction rule:

$$\begin{array}{c}\mathcal{D} \\ A \wedge B\ true\end{array} \qquad \Rightarrow_R \qquad \dfrac{\dfrac{\begin{array}{c}\mathcal{D} \\ A \wedge B\ true\end{array}}{A\ true}\ \wedge E_1 \quad \dfrac{\begin{array}{c}\mathcal{D} \\ A \wedge B\ true\end{array}}{B\ true}\ \wedge E_2}{A \wedge B\ true}\ \wedge I$$

3.1.1 Verifications and canonical forms

An examination of the requirement that introduction rules *precisely* define the connectives leads to a restricted set of proofs called *verifications*. If there is a proof of the truth of $A \wedge B$, then we don't know a great deal about its proof. Maybe it looks like this!

$$\dfrac{\dfrac{\dfrac{\vdots}{D \wedge (A \wedge B)\ true}}{A \wedge B\ true}\ \wedge E_2 \quad \begin{array}{c}\vdots \\ C\ true\end{array}}{(A \wedge B) \wedge C\ true}\ \wedge I}{A \wedge B\ true}\ \wedge E_1$$

On the other hand, if we have a *verification* of $A \wedge B$ then we know that the last step in its proof combines two (smaller) verifications, one of which is a verification of A and the other of which is a verification of B. We consider the verifications to be the canonical proofs of a proposition — when we introduce proof terms, verifications correspond to a restricted set of proofs called the *canonical forms*.

3.1.2 Definitions, atomic propositions, and adequacy

The canonical forms are interesting from the point of view of logical frameworks because they allow us to capture a representative of a derivation as a proof term within the logic. To use a standard example, we can introduce a new atomic proposition ($\mathsf{add\,N\,M\,P}$) that is defined by the two constants $\mathsf{add/z}$ and $\mathsf{add/s}$. We continue to use the convention that identifiers beginning with capital letters represent variables that are implicitly universally quantified.

$\mathsf{z} : \mathsf{nat}.$
$\mathsf{s} : \mathsf{nat} \to \mathsf{nat}.$

$\mathsf{add} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat} \to \mathsf{type}.$
$\mathsf{add/z} : \mathsf{add\ z\ N\ N}.$
$\mathsf{add/s} : \mathsf{add\ N\ M\ P} \to \mathsf{add\ (s\ N)\ M\ (s\ P)}.$

Given this signature, we would like to say that we also know what counts as a verification of the proposition $\mathsf{add\,(s\,z)\,(s\,z)\,(s(s\,z))}$ — in particular, that only one thing will do: a use of $\mathsf{add/s}$ and a verification of $\mathsf{add\,z\,(s\,z)\,(s\,z)}$. But this is not the case! We can also verify a proposition $\mathsf{add\,(s\,z)\,(s\,z)\,(s(s\,z))}$ by using an *assumption* of the same proposition, or by using an assumption of the form $\mathsf{add\,N\,M\,P} \to \mathsf{add\,N\,M\,P}$ and a verification of $\mathsf{add\,(s\,z)\,(s\,z)\,(s(s\,z))}$, or by using an assumption of the form $\mathsf{add\,N\,(s\,M)\,P} \to \mathsf{add\,N\,M\,P}$ and a verification of $\mathsf{add\,(s\,z)\,(s(s\,z))\,(s(s\,z))}$... The point is that the very open-endedness of the LF context that we pointed out in Observation 3 is preventing us from making the claim that we know something specific about what counts as a verification of the proposition $\mathsf{add\,(s\,z)\,(s\,z)\,(s(s\,z))}$.

However, it is intuitively reasonable enough to require that our current assumptions include neither assumptions of the form $\mathsf{add\,N\,M\,P}$ nor assumptions that can be immediately used to prove $\mathsf{add\,N\,M\,P}$ (which is effectively a *closed world assumption*). Under this requirement, our intuition for what counts as a verification of the proposition $\mathsf{add\,(s\,z)\,(s\,z)\,(s(s\,z))}$ is correct. Considerations such as these lead to a formal notion of *adequate encodings*, which in this case means that there is a bijection between terms of the type $\mathsf{add\,N\,M\,P}$ and derivations of $\mathsf{N} + \mathsf{M} = \mathsf{P}$ as defined by the following deductive system:

$$\frac{\mathsf{N} + \mathsf{M} = \mathsf{P}}{\mathsf{s\,N} + \mathsf{M} = \mathsf{s\,P}}\ \mathsf{add/s} \qquad \frac{}{\mathsf{z} + \mathsf{N} = \mathsf{N}}\ \mathsf{add/z}$$

There is, as the example of a static semantics from Section 2 illustrates, tension between considerations of modularity (which dictate that the form of the context should be as open-ended as possible) and adequacy (which dictate that the form of the context should be as fully specified as possible). However, experience shows that, by incorporating concepts such as subordination [Vir99], this tension is unproblematic in practice.

The above discussion is informal, and adequacy is given a much more careful treatment elsewhere [HHP93, HL07]. The point of the preceding discussion is to introduce the use of canonical forms for specifying inductively defined systems and to underscore the necessity of imposing some sort of regularity constraint on the context.

3.2 A state-transition-based view of logic

The presentation of logic as verifications and canonical forms treats atomic propositions and their proofs as the primary objects of consideration. We are interested in treating the *state of a system* and *transitions between states* as the primary objects of consideration. The result has strong similarities to sequent calculus presentations of linear logic, which is unsurprising: previous work has universally started with a sequent presentation of logic and then "read off" a state transition system. Cervesato and Scedrov's multiset rewriting language ω is a prime example of this process; an introduction to ω and a review of related work in the area can be found in their article [CS09].

We will treat the state of a specified system as being defined by a set of *ephemeral facts* about that state; we think of these ephemeral facts as *resources*. Therefore, states are multisets of ephemeral facts ($A\ res$); the empty multiset is written as $(\cdot)$ and multiset union is written as (Δ_1, Δ_2). We define the meaning of propositions by the effect they have on resources. The proposition $A \,\&\, B$ represents a resource that can provide *either* a resource A or a resource B, and $\top$ is an unusable resource, so the resource $(\top\ res)$ will always stick around uselessly. The proposition $\forall x{:}\tau.A(x)$ represents a resource that can provide the resource $A(t)$ for any term t with type τ (we assume there is some signature Σ defining base types and constants).

$$A \,\&\, B\ res \rightsquigarrow A\ res \qquad (\&_{T1})$$

$$A \,\&\, B\ res \rightsquigarrow B\ res \qquad (\&_{T2})$$

$$(\textit{no rule for } \top) \qquad (\top_T)$$

$$\forall x{:}\tau.A(x)\ res \rightsquigarrow A(t)\ res \qquad \text{if} \qquad \Sigma \vdash t : \tau \qquad (\forall_T)$$

Linear implication represents a sort of test. It simplifies matters if, initially, we only consider linear implication $(Q \multimap A)$ where Q is an atomic proposition; it means that we can consume Q to produce A.

$$Q\ res,\ Q \multimap A\ res \rightsquigarrow A\ res \qquad (\multimap_{T1})$$

The transition relation is, in general, $\Delta_1 \rightsquigarrow \Delta_2$, and there is a congruence or frame property that says a transition can happen to any part of the state.

$$\frac{\Delta_1 \rightsquigarrow \Delta_1'}{\Delta_1, \Delta_2 \rightsquigarrow \Delta_1', \Delta_2}\ \textit{frame}$$

We also may allow the signature Σ to mention certain resources as being valid or unrestrictedly available; in this case we can spontaneously generate new copies of the resource:

$$\cdot \rightsquigarrow A\ res \qquad \text{if} \qquad (A\ valid) \in \Sigma \qquad (\textit{copy})$$

This is a very brief version of this story, and in its current telling it can be understood as a subset of Cervesato and Scedrov's ω multiset rewriting language [CS09]. However, that language and other similar languages take the notion of derivability in a linear logic sequent calculus as primary and derive a transition semantics from open proofs. The story we have told here starts with transitions and generalizes to all of first-order linear logic save for additive disjunction $(A \oplus B)$ and its unit $\mathbf{0}$.[2] I believe there is an inherent value in reducing the gap between the specification of linear logic and the specification of systems within that logic, and having transitions as a first-class notion within the logical framework appears to clarify many issues related to reasoning about specifications.

[2]Furthermore, I have made substantial, though unfinished, progress on incorporating these connectives into a transition-based presentations as well.

3.2.1 Canonical transitions

Our motivation for describing logic as a transition system is to make it more natural to use logic for the specification of transition systems. An example of a very simple transition system that we would hope to represent is the following bare-bones asynchronous pi-calculus represented by the usual structural congruences and the following reduction rule:

$$c(x).p \parallel c\langle v\rangle \mapsto p[v/x]$$

To represent this system in logic, we can show that the above transition system corresponds exactly to the following signature where there are two sorts of terms, channels chan and processes process. A process can be either asynchronous send along a channel (send C V, i.e. $c\langle v\rangle$) or a receive along a channel (recv C (λx. P x), i.e. $c(x).p$). The resource (proc P) represents the active process P.

$$\begin{array}{l}\mathsf{send} : \mathsf{chan} \to \mathsf{chan} \to \mathsf{process}.\\ \mathsf{recv} : \mathsf{chan} \to (\mathsf{chan} \to \mathsf{process}) \to \mathsf{process}.\\ \\ \mathsf{proc} : \mathsf{process} \to \mathsf{type}.\\ \\ \mathsf{synch} : \mathsf{proc}(\mathsf{recv}\ \mathsf{C}\ (\lambda x.\ \mathsf{P}\ x)) \multimap \mathsf{proc}(\mathsf{send}\ \mathsf{C}\ \mathsf{V}) \multimap \mathsf{proc}(\mathsf{P}\ \mathsf{V}).\end{array}$$

What do we mean by *corresponds exactly*? Perhaps something like weak bisimulation or strong bisimulation would do, but we really want something akin to the adequacy properties discussed earlier. Adequacy in this setting means that there should be a bijective correspondence between states in linear logic and states in the simple process calculus; furthermore, transitions in both systems must respect this correspondence — any single transition that is made in the logic can be mapped onto a single transition in the process calculus, and vice-versa.

This, certainly, is not the case in system we have described so far. Take the synchronization transition $(\mathsf{a}(x).x\langle x\rangle \parallel \mathsf{a}\langle \mathsf{b}\rangle) \mapsto \mathsf{b}\langle \mathsf{b}\rangle$ in the simple process calculus. We can say that $\mathsf{a}(x).x\langle x\rangle$ is represented as the atomic proposition $\mathsf{proc}(\mathsf{recv}\,\mathsf{a}\,\lambda x.\,\mathsf{send}\,x\,x)$ and that $\mathsf{a}\langle \mathsf{b}\rangle$ is represented as the atomic proposition proc(send a b). However, because the synch rule has two premises, fully applying it will take at least two transitions in linear logic (five if we count the three transitions necessary due to the implicit quantification of C, P, and V in the synch rule).

The analogue to canonical forms and verifications in this setting addresses this problem. A *focused* presentation of our linear logic allows us to restrict attention to transitions that atomically copy a proposition from the signature, instantiate all of its variables, and satisfy all of its premises. Then we can say that the synch rule defined in the signature corresponds to the following *synthetic transition*:

$$\mathsf{proc}(\mathsf{send}\,\mathsf{C}\,\mathsf{V})\ res,\ \mathsf{proc}(\mathsf{recv}\,\mathsf{C}\,\lambda x.\,\mathsf{P}\,x)\ res \rightsquigarrow \mathsf{proc}(\mathsf{P}\,\mathsf{V})\ res \qquad (\mathsf{synch})$$

The synthetic transition associated with synch precisely captures the transition $c(x).p \parallel c\langle v\rangle \mapsto p[v/x]$ in the process calculus, so transitions in the encoded system are adequately represented by transitions in the logical framework. In order to ensure that states are adequately represented in the framework, we again must impose a constraint on the regular structure of the context, namely that it only consists of atomic resources of the form (proc P).

3.3 Combining transitions and canonical forms

In Section 3.1, we saw that deductive systems, such as the addition of unary natural numbers, can be adequately represented in a logical framework based on verifications and canonical forms. In Section 3.2 we saw that transition systems can be characterized by an interpretation of linear

logic that treats the logic as a transition system. However, a transition system frequently needs to refer to some notion of inductive definition. As a slightly contrived example, consider a transition system where each resource contains a natural number and where two distinct numbers can be added together:

$$\mathsf{num}\,\mathsf{N}_1\ \mathit{res}, \mathsf{num}\,\mathsf{N}_2\ \mathit{res} \rightsquigarrow \mathsf{num}(\mathsf{N}_1 + \mathsf{N}_2)\ \mathit{res}$$

If natural numbers are defined using the unary notion given above, then addition $\mathsf{N}_1 + \mathsf{N}_2$ is not a primitive operation. Instead, we need to explain the addition part of this transition by using an inductive definition, writing a rule that looks something like this:

$$\mathsf{merge} : \mathsf{num}\ \mathsf{N}_1 \multimap \mathsf{num}\ \mathsf{N}_2 \multimap \mathsf{add}\ \mathsf{N}_1\ \mathsf{N}_2\ \mathsf{N}_3 \multimap \mathsf{num}\ \mathsf{N}_3.$$

There have been few satisfying solutions that allow the combination of a system based on canonical forms and a system based on forward chaining, and the culprit is the fact that verifications "pause" when they reach an atomic proposition. In a system that tries to straightforwardly combine transitions and canonical forms, transitions can occur during each one of these pauses. The result is a system that does a good job of representing neither transition systems nor canonical forms.

There are a number of exceptions. One is Miller et al.'s work on logics like $\mu\mathrm{LJ}^-$ that allow inductive definitions to be directly represented in the logic. Used in place of an atomic proposition, an inductive definition allows an arbitrarily long chain of reasoning to be captured as a single uninterrupted verification, removing any pause entirely [BM07, NM09]. A serious drawback of this approach is that inductive definitions cannot extend the context of assumptions with new information. Because representing typing derivations for programming languages requires this ability, this approach is not suitable for our purposes.

Another general solution for combining transitions and canonical forms comes from the concurrent logical framework CLF [WCPW02]. In CLF, transitions are captured by a lax monad that can effectively be used to segregate transitions and canonical forms. Our particular solution is very close to the design of CLF, particularly the *semantic effects* fragment of CLF that requires the canonical forms-based fragment of the language to forgo any reference to the state-transition-based fragment [DP09].

3.3.1 Introduction to adjoint logic

In the next section, we will take a fragment of LF — a canonical forms-based logic for representing syntax and derivations — with a transition-based presentation of ordered linear logic that is used to represent evolving systems. The basis for this combination is adopted from Reed's adjoint logic [Ree09b], in which two syntactically distinct logics can be connected through unary connectives U and F. In the logical framework presented in the next section, these two logics will be a fragment of LF and a transition-based ordered linear logic, but in this section, we will consider a logic that combines persistent and linear logic:

Persistent propositions	$P, Q, R ::= p \mid UA \mid P \supset Q$
Linear/ephemeral propositions	$A, B, C ::= a \mid FP \mid A \multimap B$

Following Reed (and in order to match up with the naturally sequent-calculus-oriented state transition presentation of logic) we will consider adjoint logic as a sequent calculus. Persistent contexts Γ hold persistent facts and linear contexts Δ hold ephemeral resources; the sequent $\Gamma \vdash P$ characterizes proofs of persistent propositions and the sequent $\Gamma; \Delta \vdash A$ characterizes proofs of linear (ephemeral) propositions. The linear goal FP is only true if there are no linear hypotheses and the

persistent proposition P is provable; the linear hypothesis FP can be replaced by the persistent hypothesis P.

$$\frac{}{\Gamma; a \vdash a} \quad \frac{\Gamma \vdash P}{\Gamma; \cdot \vdash FP} \quad \frac{\Gamma, P; \Delta \vdash C}{\Gamma; \Delta, FP \vdash C} \quad \frac{\Gamma; \Delta, A \vdash B}{\Gamma; \Delta \vdash A \multimap B} \quad \frac{\Gamma; \Delta_A \vdash A \quad \Gamma; \Delta, B \vdash C}{\Gamma; \Delta, \Delta_A, A \multimap B \vdash C}$$

The description of persistent connectives below has one quirk: there are two left rules for $(P \supset Q)$, one where the ultimate goal is to prove a persistent proposition and one where the ultimate goal is to prove a linear proposition. This is not a quirk that is unique to adjoint logic; it is the same issue that requires Pfenning and Davies' judgmental S4 to have two elimination rules for $\Box A$ [PD01], for instance.[3]

$$\frac{p \in \Gamma}{\Gamma \vdash p} \quad \frac{\Gamma; \cdot \vdash A}{\Gamma \vdash UA} \quad \frac{(UA) \in \Gamma \quad \Gamma; \Delta, A \vdash C}{\Gamma; \Delta \vdash C} \quad \frac{\Gamma, P \vdash Q}{\Gamma \vdash P \supset Q}$$

$$\frac{(P \supset Q) \in \Gamma \quad \Gamma \vdash P \quad \Gamma, Q \vdash R}{\Gamma \vdash R} \qquad \frac{(P \supset Q) \in \Gamma \quad \Gamma \vdash P \quad \Gamma, Q; \Delta \vdash C}{\Gamma; \Delta \vdash C}$$

As Reed notes, if we erase every persistent proposition save for UA, the resulting logic is equivalent to linear logic with exponentials, where $!A \equiv FUA$. Similarly, if we erase every linear proposition save for FP, the resulting logic is equivalent to the Pfenning-Davies reconstruction of lax logic where $\bigcirc P \equiv UFP$.

3.3.2 Combining transitions and canonical forms with adjoint logic

My approach in this thesis will be to use adjoint logic to present a framework that connects a persistent logical framework of canonical forms and a substructural logical framework of transitions. The result, as we have just seen, combines elements of substructural logic with elements of lax logic, so it is not surprising that the end result turns out to be very similar to the CLF logical framework that (in a rather different way) also segregates transitions and deductions using a combination of lax and substructural logic.[4]

The role of UA in the framework has a somewhat uncertain status. Practically, its presence in the logic allows canonical forms to contain stateful transitions in a controlled fashion: this is not a bad thing and is a mode of specification that CLF specifically supports. It is also precisely what the framework of semantic effects disallows. Because the framework of semantic effects is the fragment of CLF that I am most interested in representing and reasoning about, the framework I present in the next section has no way of incorporating transitions within canonical forms.[5] The connective F is then the only connection between the two systems, and since it retains much of the character of the linear logic exponential, it is written as $!P$ even though it actually acts only as the "first half" of the exponential.

[3] Another way to look at this "quirk" is that the *real* rule is parametric in the conclusion, able to prove either A *res* or P *valid*, and we just give the two instances of the real rule here.

[4] I believe that adjoint logic provides a better explanation than lax logic for how CLF is actually used and thought about, but I should re-emphasize that the critical difference between CLF and the framework presented in the next section — the reason I cannot use CLF directly — is that CLF does not treat individual state transitions and sequences of state transitions as objects of consideration.

[5] This is by no means a settled point! A good example of a language that challenges the exclusion of UA can be found in Section A.3 of the appendix, where I give an SSOS-like specification of the language from Davies' temporal-logic approach to binding-time analysis [Dav96].

4 A logical framework for evolving systems

In the previous section, we considered the use of logic to describe both deductive systems and state-transition systems, and our immediate goal is to use a logical framework based on these principles to specify the operational semantics of programming languages with state and concurrency. In this section we will present that logical framework, which based on a substructural logic that includes both ordered and linear propositions.

In addition to CLF, the design synthesizes ideas from Polakow et al.'s ordered logic framework [Pol01], Reed's adjoint logic [Ree09b], and Zeilberger's polarized higher-order focusing [Zei09]. There is no fundamental reason why the canonical forms fragment of the framework described here cannot be the full dependently typed logical framework LF [HHP93, HL07]; however, it is convenient for the purposes of this proposal to just consider a restricted fragment of LF.

4.1 Representing terms

We have already been using LF-like representations of terms: both natural numbers and process-calculus terms can be adequately represented as canonical forms in a simply-typed lambda calculus under the following signature.

$\mathsf{nat} : \mathsf{type}.$	$\mathsf{process} : \mathsf{type}.$
$\mathsf{z} : \mathsf{nat}.$	$\mathsf{chan} : \mathsf{type}.$
$\mathsf{s} : \mathsf{nat} \to \mathsf{nat}.$	$\mathsf{send} : \mathsf{chan} \to \mathsf{chan} \to \mathsf{process}.$
	$\mathsf{recv} : \mathsf{chan} \to (\mathsf{chan} \to \mathsf{process}) \to \mathsf{process}.$

A term type τ is either an atomic term type a defined by the signature (like nat, process, and chan) or an implication $\tau \to \tau$. The signature also defines term constants (like z, s, send, and recv) which are defined to have some type τ. Canonical terms are the η-long, β-normal terms of the simply-typed lambda calculus using constants drawn from the signature: the natural number 3 is represented as $\mathsf{s}(\mathsf{s}(\mathsf{s}\,\mathsf{z}))$ and the process $\mathsf{a}(x).x\langle x\rangle$ is represented as the canonical term $(\mathsf{recv}\,\mathsf{a}\,\lambda\mathsf{x}.\,\mathsf{send}\,\mathsf{x}\,\mathsf{x})$.

4.2 Judgments as types

An organizing principle of LF-like frameworks is the representation of *judgments as types* — in the case of the aforementioned example of addition, the three-place judgment $n_1 + n_2 = n_3$ is represented by the type family add of kind $\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat} \to \mathsf{type}$, and for any specific n_1, n_2, and n_3 we represent derivations of $\mathsf{n}_1 + \mathsf{n}_2 = \mathsf{n}_3$ as canonical proof terms of type $(\mathsf{add}\,\mathsf{n}_1\,\mathsf{n}_2\,\mathsf{n}_3)$ in the following signature:

$\mathsf{add} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat} \to \mathsf{type}.$
$\mathsf{add/z} : \Pi\mathsf{N}{:}\mathsf{nat}.\ \mathsf{add}\ \mathsf{z}\ \mathsf{N}\ \mathsf{N}.$
$\mathsf{add/s} : \Pi\mathsf{N}{:}\mathsf{nat}.\ \Pi\mathsf{M}{:}\mathsf{nat}.\ \Pi\mathsf{P}{:}\mathsf{nat}.\ \mathsf{add}\ \mathsf{N}\ \mathsf{M}\ \mathsf{P} \to \mathsf{add}\ (\mathsf{s}\ \mathsf{N})\ \mathsf{M}\ (\mathsf{s}\ \mathsf{P}).$

In this signature, the proof term $(\mathsf{add/s}\,\mathsf{z}\,(\mathsf{s}\,\mathsf{z})\,(\mathsf{s}\,\mathsf{z})\,(\mathsf{add/z}\,(\mathsf{s}\,\mathsf{z})))$ acts as a representative of this derivation:

$$\dfrac{\dfrac{}{\mathsf{z} + \mathsf{s}\,\mathsf{z} = \mathsf{s}\,\mathsf{z}}\ \mathsf{add/z}}{\mathsf{s}\,\mathsf{z} + \mathsf{s}\,\mathsf{z} = \mathsf{s}(\mathsf{s}\,\mathsf{z})}\ \mathsf{add/s}$$

More generally, type families are declared as $(a : \tau_1 \to \ldots \to \tau_n \to \mathsf{type})$ in the signature; atomic types in the type family a are written as $a\,t_1 \ldots t_n$, where each of the arguments t_i has type τ_i. Types A are either atomic types, implications $A \to B$, or universal quantifications over terms $\Pi x{:}\tau.A$. We usually leave universal quantification implicit in the signature, which allows us to write rules as we did earlier in this proposal:

$\mathsf{add/z} : \mathsf{add\ z\ N\ N}.$
$\mathsf{add/s} : \mathsf{add\ N\ M\ P} \to \mathsf{add\ (s\ N)\ M\ (s\ P)}.$

Furthermore, we leave the instantiation of implicit quantifiers implicit as well, which allows us to write the proof term corresponding to the above derivation more concisely as $(\mathsf{add/s\ add/z})$.

4.3 States as contexts

In the transition-based fragment of the logic, the states of the systems we intend to specify are represented as collections of atomic propositions. In sequent calculus-based logical frameworks, these collections correspond to *contexts*; we adopt this terminology.

Ephemeral contexts Linear logic and other substructural logics are useful for describing systems that change over time; part of the reason for this is that a linear resource is effectively an *ephemeral* fact about the state of a system — it describes something that is true in the current state of the system but which may not be true in a future state of the system. Our framework considers two kinds of ephemeral facts: *linear* facts (insensitive to their position relative to other facts) and *ordered* facts (sensitive to their position relative to other ordered facts).

We incorporate ephemeral facts into our framework by introducing two different kinds of type family: linear type families (where we write type_l — l for linear — instead of type in the type family declaration) and ordered type families (where we write type_o — o for ordered — instead of type in the type family declaration). This effectively enforces a syntactic separation between those persistent, linear, and ordered atomic propositions; the need for this syntactic separation has been noted in previous work [SP08, SP09] but was only given a logical interpretation in [PS09].

An ephemeral context (written Δ) is syntactically a list of linear and ordered types, but we consider contexts to be equivalent if they can be made syntactically equal by reordering linear propositions. As an example, if a_l and b_l are linear atomic propositions and c_o and d_o are ordered atomic propositions, then the contexts $(\mathsf{a}_l,\ \mathsf{b}_l,\ \mathsf{c}_o,\ \mathsf{d}_o)$, $(\mathsf{c}_o,\ \mathsf{d}_o,\ \mathsf{b}_l,\ \mathsf{a}_l)$, and $(\mathsf{c}_o,\ \mathsf{a}_l,\ \mathsf{d}_o,\ \mathsf{b}_l)$ are all equivalent, but they are not equivalent to $(\mathsf{a}_l,\ \mathsf{b}_l,\ \mathsf{d}_o,\ \mathsf{c}_o)$ because the ordered propositions c_o and d_o appear in a different order. We write this equivalence on contexts as $\Delta \approx \Delta'$.

We can use ordered resources to represent the *concrete* syntax of the language $\mathcal{L}_0$ from Section 2 by defining an ordered atomic proposition for each syntactic token:

$+ : \mathsf{type}_o.$
$\mathsf{num} : \mathsf{nat} \to \mathsf{type}_o.$

The proposition $+$ represents the addition token, and $\mathsf{num\ N}$ (where N is a term of type nat) represents a token for the natural number N. While we cannot yet actually describe parsing, we can intuitively say that the ephemeral context $(\mathsf{num(s\,z)}), +, (\mathsf{num\,z})$ corresponds to the $\mathcal{L}_0$ program $1 + 0$, the ephemeral context $(\mathsf{num(s\,z)}), +, (\mathsf{num\,z}), +, (\mathsf{num(s(s\,z))})$ is ambiguous and corresponds to either the $\mathcal{L}_0$ program $(1 + 0) + 2$ or the $\mathcal{L}_0$ program $1 + (0 + 2)$, and the ephemeral context $+, +, (\mathsf{num(s\,z)})$ is not syntactically well-formed and does not correspond to any $\mathcal{L}_0$ program.

As in sequent calculus presentations of a logic, we annotate the propositions in a context with unique variable names. Therefore, the context corresponding to the first $\mathcal{L}_0$ program above could also have been been written as $(x_1 : \mathsf{num(s\,z)}, x_2 : +, x_3 : \mathsf{num\,z})$. We further follow the convention of sequent calculus presentations of logic by often not mentioning these variable names when it is possible to reconstruct them.

Persistent contexts In addition to ephemeral state, is important to allow systems to have *persistent* state, which includes both dynamically introduced parameters and facts that, once true,

necessarily stay true.[6] We write free term parameters as $x : \tau$ and persistent assumptions as $x : A$, where A is a persistent type. A collection of these assumptions is a persistent context, written as Γ, and we treat all contexts containing the same terms and assumptions as equivalent, writing this equivalence as $\Gamma \approx \Gamma'$.

4.4 Substitutions

The connection point between contexts and terms is the definition of *substitutions*. A substitution σ is a term which gives evidence that, given the persistent facts Γ and ephemeral resources Δ, we can model or represent the state described by the persistent facts Γ' and ephemeral facts Δ'. Alternatively, we can think of Γ' and Δ' as "context-shaped holes," in which case a substitution is evidence that Γ and Δ "fit in the holes."

The way a substitution gives this evidence is by providing a resource (designated by its label y) for every for every resource in Δ', providing a proof term $\mathcal{D}$ that is defined in the context Γ for every fact $x : A$ in Γ', and providing a term t that is defined in the context Γ for every $x : \tau$ in Γ'.[7] Syntactically, a substitution is described by the following grammar:

$$\sigma ::= [] \mid \sigma, t/x \mid \sigma, \mathcal{D}/x \mid \sigma, y/x$$

The definition of substitution typing — $\Gamma; \Delta \vdash_\Sigma \sigma : \Gamma'; \Delta'$ — captures the judgment that σ is a witness to the fact that the state $\Gamma; \Delta$ can model or represent the state Γ', Δ' (under the signature Σ). The definition relies on two other judgments, $\Gamma \vdash_\Sigma t : \tau$ and $\Gamma \vdash_\Sigma \mathcal{D} : A$, which describe well-typed terms and well-typed proof terms, respectively. It also relies on the operation $A[\sigma]$, the application of a substitution to a type. Because we are using only a restricted fragment of LF, only the term components (t/x) of substitutions actually matter for this substitution.

$$\frac{}{\Gamma; \cdot \vdash_\Sigma [] : \cdot; \cdot} \qquad \frac{\Gamma; \cdot \vdash_\Sigma \sigma : \Gamma'; \cdot \qquad \Gamma \vdash_\Sigma t : \tau}{\Gamma; \cdot \vdash_\Sigma (\sigma, t/x) : (\Gamma', x : \tau); \cdot} \qquad \frac{\Gamma; \cdot \vdash_\Sigma \sigma : \Gamma'; \cdot \qquad \Gamma \vdash_\Sigma \mathcal{D} : A[\sigma]}{\Gamma; \cdot \vdash_\Sigma (\sigma, \mathcal{D}/x) : (\Gamma', x : A); \cdot}$$

$$\frac{\Delta \approx (\Delta'', y : Q') \qquad Q' = Q[\sigma] \qquad \Gamma; \Delta'' \vdash_\Sigma \sigma : \Gamma'; \Delta'}{\Gamma; \Delta \vdash_\Sigma (\sigma, y/x) : \Gamma'; (\Delta', x : Q)}$$

4.5 Positive types as patterns

In the discussion of a transition-based view of logic in Section 3.2, the only connectives we discussed were those that were defined in terms of their effect on the context; these are the connectives that make up the so-called *negative* propositions. Another family of connectives, which make up the so-called *positive* connectives, are more naturally defined by the contexts they describe. The pattern judgment — $\Gamma; \Delta \Vdash p :: A^+$ — expresses that p is the evidence that A^+ describes the context $\Gamma; \Delta$. The most natural way to think about Γ and Δ, which should appropriately be seen as an *output* of the judgment, is that it is describing a context-shaped hole that will be filled by a substitution in order to prove A^+.

Ordered conjunction ($A^+ \cdot B^+$) describes a context containing, to the left, the context described by A^+ and then, to the right, the context described by B^+.

$$\frac{\Gamma_1; \Delta_1 \Vdash p_1 :: A^+ \quad \Gamma_2; \Delta_2 \Vdash p_2 :: B^+}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \Vdash (p_1 \cdot p_2) :: (A^+ \cdot B^+)}$$

[6] Parameters can be used to model the channels in the process calculus from the previous section, and they also model abstract heap locations in the specification of ML-style references in Appendix A.1. We need to be able to dynamically introduce new parameters in order to represent producing a new channel or allocating a new location on the heap.

[7] In an appropriate dependently-typed setting, term types τ and types A are members of the same syntactic class, as are terms t and proof terms $\mathcal{D}$.

Existential quantification ($\exists x{:}\tau.A^+$) describes a context described by A^+, plus a variable $x : \tau$. If we are thinking of the output contexts as context-shaped holes, then x is a hole that must be filled by a term of type τ. The rule below should make it clear that pattern judgments are fundamentally different from typing judgments, where we would expect the x to appear in the premise, not in the conclusion.

$$\frac{\Gamma; \Delta \Vdash p :: A^+}{x : \tau, \Gamma; \Delta \Vdash (x.p) :: (\exists x{:}\tau.A^+)}$$

Finally, $\mathbf{1}$ describes a context containing no ephemeral propositions, and we treat atomic propositions as positive as well: Q (where Q is an ordered atomic proposition) describes a context with just one ordered atomic proposition Q, ¡Q (where Q is a linear atomic proposition) describes a context with just one linear atomic proposition Q, and $!A$ (where A is a persistent proposition) describes a context with no ephemeral propositions where A is true.

$$\frac{}{\cdot; \cdot \Vdash () :: \mathbf{1}} \qquad \frac{}{x : A; \cdot \Vdash x :: !A} \quad \frac{}{\cdot; x : Q \Vdash x :: ¡Q} \quad \frac{}{\cdot; x : Q \Vdash x :: Q}$$

Example. Because we may think of the pattern judgment $\Gamma; \Delta \Vdash p :: A^+$ as producing context-shape holes Γ and Δ, looking at a pattern (which produces context-shape holes) together with a substitution (which provides evidence that those holes can be filled) is illustrative. If we want to show that the ephemeral context $(x_1 : \mathsf{num}(\mathsf{s}\,\mathsf{z}), x_2 : +, x_3 : \mathsf{num}\,\mathsf{z})$ is described by the positive proposition $\exists n.\,(\mathsf{num}\,n \cdot +) \cdot (\mathsf{num}\,\mathsf{z} \cdot !\mathsf{add}\,n\,n\,(\mathsf{s}(\mathsf{s}\,\mathsf{z})))$, then we first find the pattern associated with that proposition (we write n instead of num for brevity):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot; y_1 : \mathsf{n}\,n \Vdash y_1 :: \mathsf{n}\,n} \quad \dfrac{}{\cdot; y_2 : + \Vdash y_2 :: +}}{\cdot; y_1 : \mathsf{n}\,n, y_2 : + \Vdash y_1 \cdot y_2 :: (\mathsf{n}\,n \cdot +)} \quad \dfrac{\dfrac{}{\cdot; y_3 : \mathsf{n}\,\mathsf{z} \Vdash y_3 :: \mathsf{n}\,\mathsf{z}} \quad \dfrac{}{y_4 : \mathsf{add}\,n\,n\,(\mathsf{s}(\mathsf{s}\,\mathsf{z})); \cdot \Vdash y_4 :: !\mathsf{add}\,n\,n\,(\mathsf{s}(\mathsf{s}\,\mathsf{z}))}}{y_4 : \mathsf{add}\,n\,n\,(\mathsf{s}(\mathsf{s}\,\mathsf{z})); y_3 : \mathsf{n}\,\mathsf{z} \Vdash (y_3 \cdot y_4) :: (\mathsf{n}\,\mathsf{z} \cdot !\mathsf{add}\,n\,n\,(\mathsf{s}(\mathsf{s}\,\mathsf{z})))}}{y_4 : \mathsf{add}\,n\,n\,(\mathsf{s}(\mathsf{s}\,\mathsf{z})); y_1 : \mathsf{n}\,n, y_2 : +, y_3 : \mathsf{n}\,\mathsf{z} \Vdash (y_1 \cdot y_2) \cdot (y_3 \cdot y_4) :: (\mathsf{n}\,n \cdot +) \cdot (\mathsf{n}\,\mathsf{z} \cdot !\mathsf{add}\,n\,n\,(\mathsf{s}(\mathsf{s}\,\mathsf{z})))}}{n : \mathsf{nat}, y_4 : \mathsf{add}\,n\,n\,(\mathsf{s}(\mathsf{s}\,\mathsf{z})); y_1 : \mathsf{n}\,n, y_2 : +, y_3 : \mathsf{n}\,\mathsf{z} \Vdash (n.(y_1 \cdot y_2) \cdot (y_3 \cdot y_4)) :: \exists n.\,(\mathsf{n}\,n \cdot +) \cdot (\mathsf{n}\,\mathsf{z} \cdot !\mathsf{add}\,n\,n\,(\mathsf{s}(\mathsf{s}\,\mathsf{z})))}$$

Then, we find a substitution σ such that we can derive the following:

$$\cdot; (x_1 : \mathsf{num}(\mathsf{s}\,\mathsf{z}), x_2 : +, x_3 : \mathsf{num}\,\mathsf{z}) \vdash \sigma : (n : \mathsf{nat}, y_4 : \mathsf{add}\,n\,n\,(\mathsf{s}(\mathsf{s}\,\mathsf{z}))); (y_1 : \mathsf{num}\,n, y_2 : +, y_3 : \mathsf{num}\,\mathsf{z})$$

The substitution $\sigma = [], (\mathsf{s}\,\mathsf{z})/n, (\mathsf{add}/\mathsf{s}\,\mathsf{add}/\mathsf{z})/y_4, x_1/y_1, x_2/y_2, x_3/y_3$ works in this case.

In future examples, we will use a derived notation of a *filled pattern*, a substitution applied to a pattern. For example, rather than first deriving a pattern and then a substitution in the preceding example, we will simply say that the filled pattern $((\mathsf{s}\,\mathsf{z}).\,(x_1 \cdot x_2) \cdot (x_3 \cdot (\mathsf{add}/\mathsf{s}\,\mathsf{add}/\mathsf{z})))$ is a proof term showing that the ephemeral context $(x_1 : \mathsf{num}(\mathsf{s}\,\mathsf{z}), x_2 : +, x_3 : \mathsf{num}\,\mathsf{z})$ is described by the positive proposition $\exists n.\,(\mathsf{num}\,n \cdot +) \cdot (\mathsf{num}\,\mathsf{z} \cdot !\mathsf{add}\,n\,n\,(\mathsf{s}(\mathsf{s}\,\mathsf{z})))$.

4.6 Negative types as transition rules

Negative types were introduced in Section 3.2 as describing ways in which contexts can change. The proposition $\mathsf{b} \cdot \mathsf{c} \twoheadrightarrow \uparrow\mathsf{e}$, for example, is essentially a rewriting rule that allows an ephemeral context such as $(\mathsf{a}\,\mathsf{b}\,\mathsf{c}\,\mathsf{d})$ to be rewritten as an ephemeral context $(\mathsf{a}\,\mathsf{e}\,\mathsf{d})$.

The judgment associated with negative propositions is $(\Gamma; \Delta \Vdash g :: A^- \gg \Gamma'; \Delta' \Vdash p')$. This judgment expresses that, if the premises of A^- — the context-shaped holes Γ and Δ — can be filled by consuming some number of ephemeral resources, then the consumed ephemeral resources can be replaced with the conclusion of A^- — the resources described by Δ' and the new facts described by Γ'. The *goal* g captures the premises, and the pattern p describes the conclusion; together, a goal g and a pattern p are the proof term associated with a negative proposition.

The simplest negative proposition is $\uparrow A^+$, which consumes no resources and produces the resources described by A^+:

$$\frac{\Gamma'; \Delta' \Vdash p' :: A^+}{\cdot; \cdot \Vdash () :: \uparrow A^+ \gg \Gamma'; \Delta' \Vdash p'}$$

The proposition $A^- \,\&\, B^-$ can represent either the transition described by A^- or the transition described by B^-:

$$\frac{\Gamma; \Delta \Vdash p :: A^- \gg \Gamma'; \Delta' \Vdash p'}{\Gamma; \Delta \Vdash (\pi_1 p) :: (A^- \,\&\, B^-) \gg \Gamma'; \Delta' \Vdash p'} \qquad \frac{\Gamma; \Delta \Vdash p :: B^- \gg \Gamma'; \Delta' \Vdash p'}{\Gamma; \Delta \Vdash (\pi_2 p) :: (A^- \,\&\, B^-) \gg \Gamma'; \Delta' \Vdash p'}$$

Transitions have to be located at a particular point in the ephemeral context; ordered implication $(A^+ \twoheadrightarrow B^-)$ is a transition that consumes a part of the context described by A^+ to the right of its initial position and then proceeds with the transition described by B^-.[8]

$$\frac{\Gamma_1; \Delta_1 \Vdash p :: A^+ \quad \Gamma_2; \Delta_2 \Vdash g :: B^- \gg \Gamma'; \Delta' \Vdash p'}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \Vdash (p; g) :: (A^+ \twoheadrightarrow B^-) \gg \Gamma'; \Delta' \Vdash p'}$$

Finally, universal quantification $\forall x{:}\tau.A^-$ can behave like $A^-[t/x]$ for any $t : \tau$; therefore, its pattern introduces a new

$$\frac{\Gamma; \Delta \Vdash g :: A^- \gg \Gamma'; \Delta' \Vdash p'}{x : \tau, \Gamma; \Delta \Vdash (x.g) :: (\forall x{:}\tau.A^-) \gg \Gamma'; \Delta' \Vdash p'}$$

When the placement of the shift operator $\uparrow$ can be inferred from the context, we will leave it implicit, writing $A^+ \twoheadrightarrow B^+$ instead of $A \twoheadrightarrow \uparrow B^+$.

4.7 Transitions and expressions

We now can actually describe parsing the concrete syntax of $\mathcal{L}_0$ using transition rules. First, we describe the abstract syntax of $\mathcal{L}_0$ as canonical terms of type exp; then we describe an ordered type family parsed E and two rewriting rules describing how the concrete syntax can be parsed.

```
exp : type.
n : nat → exp.
plus : exp → exp → exp.

parsed : exp → type_o.
parse/num : num N ↠ parsed(n N).
parse/plus : parsed E1 · + · parsed E2 ↠ parsed(plus E1 E2).
```

Intuitively, parse/num is a rewriting rule that allows us to transition from the ephemeral context $(\mathsf{num}(\mathsf{s}\,\mathsf{z})), +, (\mathsf{num}\,\mathsf{z})$ to the ephemeral context $(\mathsf{parsed}(\mathsf{n}(\mathsf{s}\,\mathsf{z}))), +, (\mathsf{num}\,\mathsf{z})$. To describe the proof term capturing this transition, we first discover the goal and pattern associated with the type of parse/num (note that the implicit quantification and the implicit shift operator $\uparrow$ are both made explicit):

$$\dfrac{\dfrac{\dfrac{}{(\cdot; y : \mathsf{num}\,n) \Vdash y :: \mathsf{num}\,n} \quad \dfrac{\dfrac{}{(\cdot; y_1 : \mathsf{parsed}(\mathsf{n}\,n)) \Vdash y_1 :: \mathsf{parsed}(\mathsf{n}\,n)}}{(\cdot; \cdot) \Vdash () ::\uparrow \mathsf{parsed}(\mathsf{n}\,n) \gg (\cdot; y_1 : \mathsf{parsed}(\mathsf{n}\,n)) \Vdash y_1}}{(\cdot; y : \mathsf{num}\,n) \Vdash (y; ()) :: (\mathsf{num}\,n \twoheadrightarrow\uparrow \mathsf{parsed}(\mathsf{n}\,n)) \gg (\cdot; y_1 : \mathsf{parsed}(\mathsf{n}\,n)) \Vdash y_1}}{(n : \mathsf{nat}; y : \mathsf{num}\,n) \Vdash (n.y; ()) :: (\forall n.\, \mathsf{num}\,n \twoheadrightarrow\uparrow \mathsf{parsed}(\mathsf{n}\,n)) \gg (\cdot; y_1 : \mathsf{parsed}(\mathsf{n}\,n)) \Vdash y_1}$$

[8] In ordered logic there are actually two distinct implications, right ordered implication $A^+ \twoheadrightarrow B^-$ and left ordered implication $A^+ \twoheadleftarrow B^-$. Either one or the other is usually sufficient for the logical fragment we are considering, and I conjecture that as long as the ephemeral context only contains atomic propositions, the addition of left ordered implication does not actually add any expressiveness.

If x_1 is the variable marking the ordered resource $(\mathsf{parsed}(\mathsf{n}(\mathsf{s}\,\mathsf{z})))$, we can fill this goal with the substitution $([], \mathsf{s}\,\mathsf{z}/n, x_1/y)$; the resulting filled pattern is $(\mathsf{s}\,\mathsf{z}.\, x_1; ())$. The proof term corresponding to the transition can be written in a CLF-like notation as the let-expression $(\mathsf{let}\ (y_1) = \mathsf{parse/num}(\mathsf{s}\,\mathsf{z}.\, x_1; ())\ \mathsf{in})$. However, I prefer a more "directional" notation for transitions that looks like this: $(\mathsf{parse/num}(\mathsf{s}\,\mathsf{z}.\, x_1; ()) \rhd (y_1))$. In addition, we will leave universal quantification implicit and omit the trailing () in a goal; the final result is a proof term for the aforementioned transition that looks like this:

$$\begin{aligned}&\mathsf{parse/num}(x_1) \rhd (y_1)\\ &\qquad : \ (x_1 : \mathsf{num}(\mathsf{s}\,\mathsf{z}), x_2 : +, x_3 : \mathsf{num}\,\mathsf{z}) \ \rightsquigarrow_{\Sigma\mathsf{parse}} \ (y_1 : \mathsf{parsed}(\mathsf{n}(\mathsf{s}\,\mathsf{z})), x_2 : +, x_3 : \mathsf{num}\,\mathsf{z})\end{aligned}$$

We'll return to the $\Sigma\mathsf{parse}$ annotation in the next section; in the meantime, the typing rule for a transition $\mathcal{T} : (\Gamma; \Delta \rightsquigarrow_\Sigma \Gamma'; \Delta')$ is as follows:

$$\frac{c : A^- \in \Sigma \qquad \Gamma_{in}; \Delta_{in} \Vdash g :: A^- \gg \Gamma_{out}; \Delta_{out} \Vdash p \qquad \Delta \approx \Delta_L, \Delta', \Delta_R \qquad \Gamma; \Delta' \vdash_\Sigma \sigma : \Gamma_{in}; \Delta_{in}}{c(g[\sigma]) \rhd p : \Gamma; \Delta \rightsquigarrow_\Sigma \Gamma, \Gamma_{out}; \Delta_L, \Delta_{out}, \Delta_R}$$

This transition can be read like this:

- Starting from the state $\Gamma; \Delta$,
- Pick a rule A^- from the signature Σ (this is the constant c),
- Determine the input context $\Gamma_{in}; \Delta_{in}$ (this is the goal g) and the output context $\Gamma_{out}; \Delta_{out}$ (this is the pattern p) associated with A^-,
- Split the ephemeral context Δ into three parts, Δ_L, Δ', and Δ_R,
- Show that, using the persistent facts Γ and the ephemeral resources Δ' you can fulfill the demands represented by the input pattern (this is the substitution σ), and
- Extend the persistent context with Γ_{out} and replace Δ' in the context with Δ_{out} to get the result of the transition.

4.8 Expressions

A sequence of one or more transitions $\mathcal{T}$ is an expression $\mathcal{E} : (\Gamma; \Delta \rightsquigarrow^*_\Sigma \Gamma'; \Delta')$. An expression is either a single transition $\mathcal{T}$, the sequential composition of two expressions $\mathcal{E}_1; \mathcal{E}_2$, or the empty expression $\diamond$. We treat sequential composition as an associative operation with unit $\diamond$.

The following expression represents a complete parse of an $\mathcal{L}_0$ program:

$$\begin{aligned}&\mathsf{parse/num}(x_1) \rhd (y_1);\ \ \mathsf{parse/num}(x_3) \rhd (y_3);\ \ \mathsf{parse/plus}(y_1 \cdot x_2 \cdot y_3) \rhd (y)\\ &\qquad : \ (x_1 : \mathsf{num}(\mathsf{s}\,\mathsf{z}), x_2 : +, x_3 : \mathsf{num}\,\mathsf{z}) \quad \rightsquigarrow^*_{\Sigma\mathsf{parse}} \quad (y : \mathsf{parsed}(\mathsf{plus}\,(\mathsf{n}(\mathsf{s}\,\mathsf{z}))\,(\mathsf{n}\,\mathsf{z})))\end{aligned}$$

The annotation of a signature $\Sigma\mathsf{parse}$ on the transition $\mathcal{T} : (\Gamma; \Delta \rightsquigarrow_\Sigma \Gamma'; \Delta')$ will be critical in practice. We frequently want to define multiple families of transitions that can act on a single proposition. For instance, consider the following rules which describe the direct calculation of a number from the concrete syntax of $\mathcal{L}_0$:

$$\begin{aligned}&\mathsf{calc} : \mathsf{nat} \to \mathsf{type}_o.\\ &\mathsf{calc/num} : \mathsf{num}\ \mathsf{N} \twoheadrightarrow \mathsf{calc}\ \mathsf{N}.\\ &\mathsf{calc/plus} : \ \mathsf{calc}\ \mathsf{N}_1 \cdot + \cdot \mathsf{calc}\ \mathsf{N}_2 \cdot !\mathsf{add}\ \mathsf{N}_1\ \mathsf{N}_2\ \mathsf{N}_3 \twoheadrightarrow \mathsf{calc}\ \mathsf{N}_3.\end{aligned}$$

Then we can talk about the transitions and expressions where we refer only to the transition rules beginning with calc.

$$\mathsf{calc/num}(x_1) \rhd (y_1) \;:\; (x_1 : \mathsf{num}(\mathsf{s}\,\mathsf{z}), x_2 : +, x_3 : \mathsf{num}\,\mathsf{z}) \rightsquigarrow_{\Sigma\mathsf{calc}} (y_1 : \mathsf{calc}(\mathsf{s}\,\mathsf{z}), x_2 : +, x_3 : \mathsf{num}\,\mathsf{z})$$

$$\mathsf{calc/num}(x_1) \rhd (y_1);\quad \mathsf{calc/num}(x_3) \rhd (y_3);\quad \mathsf{calc/plus}(y_1 \cdot x_2 \cdot y_3 \cdot (\mathsf{add/s}\,\mathsf{add/z})) \rhd (y) \\ :\; (x_1 : \mathsf{num}(\mathsf{s}\,\mathsf{z}), x_2 : +, x_3 : \mathsf{num}\,\mathsf{z}) \rightsquigarrow^*_{\Sigma\mathsf{calc}} (y : \mathsf{calc}(\mathsf{s}\,\mathsf{z}))$$

We use these annotations to specify theorems about the relationship between two different transitions or expressions. For instance, if $\Sigma\mathsf{e}$ describes the (not-yet-specified) SSOS specification of $\mathcal{L}_0$, then we can write the following conjecture about the relationship between the rules starting with $\mathsf{calc/}\ldots$ and the rules starting with $\mathsf{parse/}\ldots$

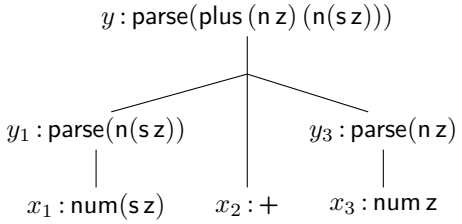
Conjecture 1. *If* $(\Delta \rightsquigarrow^*_{\Sigma\mathsf{calc}} y : \mathsf{calc}\,\mathsf{N})$ *and such that* $(\Delta \rightsquigarrow^*_{\Sigma\mathsf{parse}} y : \mathsf{parsed}\,\mathsf{E})$*, then* $(x : \mathsf{eval}\,\mathsf{E} \rightsquigarrow^*_{\Sigma\mathsf{e}} y : \mathsf{retn}(\mathsf{n}\,\mathsf{N}))$.

4.9 Concurrent equivalence

The two expressions below both represent a complete parse of the $\mathcal{L}_0$ program that we have been using in our example:

$$\mathsf{parse/num}(x_3) \rhd (y_3);\quad \mathsf{parse/num}(x_1) \rhd (y_1);\quad \mathsf{parse/plus}(y_1 \cdot x_2 \cdot y_3) \rhd (y)$$
$$\mathsf{parse/num}(x_1) \rhd (y_1);\quad \mathsf{parse/num}(x_3) \rhd (y_3);\quad \mathsf{parse/plus}(y_1 \cdot x_2 \cdot y_3) \rhd (y)$$

In fact, these two expressions represent the *same* parse if we look at the parse not as a list of transitions but as a tree of string rewrites:



The notion of concurrent (or permutative) equivalence $\mathcal{E}_1 \approx \mathcal{E}_2$ captures the notion that, while we represent expressions as sequences, they are perhaps more appropriately thought of as directed acyclic graphs. Concurrent equivalence is the least equivalence relation such that

$$(\mathcal{E}_1;\quad c(g[\sigma]) \rhd p;\quad c'(g'[\sigma']) \rhd p';\quad \mathcal{E}_2) \quad\approx\quad (\mathcal{E}_1;\quad c'(g'[\sigma']) \rhd p';\quad c(g[\sigma]) \rhd p;\quad \mathcal{E}_2)$$

whenever σ does not include any variables bound by p' and σ' likewise does not mention any variables bound by p.

4.10 Metatheory

In this section, I list some of the metatheoretic results that I expect to hold in the logical framework presented in this section. I mostly present decidability results; another important part of the metatheory of this framework is showing that the focused logic underlying this framework is complete with respect to an unfocused logic.

Even if taking for granted that there is a decidable syntactic equality in the presence of associative operations like concatenation of contexts and sequential composition of expressions, the more coarse equivalences we have discussed need to be shown to be decidable.

Conjecture 2 (Decidability of equivalences)**.** *The three different equivalences we consider are all decidable.*

- *For all persistent contexts* Γ *and* Γ'*, it is decidable whether* $\Gamma \approx \Gamma'$*.*
- *For all ephemeral contexts* Δ *and* Δ'*, it is decidable whether* $\Delta \approx \Delta'$*.*
- *For all expressions* $\mathcal{E}$ *and* $\mathcal{E}'$*, it is decidable whether* $\mathcal{E} \approx \mathcal{E}'$*.*

One way to approach this conjecture is to follow the proof of decidability for concurrent equivalence in CLF [WCPW02]. First, a syntax-directed version of the equivalence relation is given, and then this syntax-directed (and therefore obviously decidable) definition is shown to be reflexive, symmetric, and transitive.

The decidability of typing is another important component of the metatheory of this framework. I expect to prove the following statements about the finalized framework:

Conjecture 3 (Decidability of typing)**.** *Given a signature* Σ*,*

- *For all* Γ*,* Δ*,* Γ'*,* Δ'*, and* σ*, it is decidable whether* $\Gamma; \Delta \vdash_\Sigma \sigma : \Gamma'; \Delta'$*.*
- *For all* A^+ *and* p*, it is decidable whether there exist contexts* Γ *and* Δ *such that* $\Gamma; \Delta \Vdash p :: A^+$*.*
- *For all* A^-*,* g*, and* p*, it is decidable whether there exist contexts* Γ*,* Δ*,* Γ'*, and* Δ' *such that* $\Gamma; \Delta \Vdash g :: A^- \gg \Gamma'; \Delta' \Vdash p$*.*
- *For all transitions* $\mathcal{T}$ *and contexts* Γ*,* Δ*,* Γ'*, and* Δ'*, it is decidable whether* $\mathcal{T} : (\Gamma; \Delta \rightsquigarrow_\Sigma \Gamma'; \Delta')$*.*
- *For all expressions* $\mathcal{E}$ *and contexts* Γ*,* Δ*,* Γ'*, and* Δ'*, it is decidable whether* $\mathcal{E} : (\Gamma; \Delta \rightsquigarrow_\Sigma \Gamma'; \Delta')$*.*

Since we treat expressions as typed objects, we need to ensure that concurrent equivalence does not tamper with typing:

Conjecture 4 (Typed concurrent equivalence)**.** *If we treat persistent and ephemeral contexts up to context equivalence, then if* $\mathcal{E}_1 \approx \mathcal{E}_2$ *and* $\mathcal{E}_1 : (\Gamma; \Delta \rightsquigarrow^*_\Sigma \Gamma'; \Delta')$*, then* $\mathcal{E}_2 : (\Gamma; \Delta \rightsquigarrow^*_\Sigma \Gamma'; \Delta')$*.*

Because expressions do not include an end point, they have a frame-like monotonicity property; extra state can be freely added to both the beginning and the end of the expression.

Conjecture 5 (Frame property)**.**
If $\mathcal{T} : \Gamma_1; \Delta_1 \rightsquigarrow_\Sigma \Gamma_2; \Delta_2$*, then* $\mathcal{T} : \Gamma, \Gamma_1; \Delta_L, \Delta_1, \Delta_R \rightsquigarrow_\Sigma \Gamma, \Gamma_2; \Delta_L, \Delta_2, \Delta_R$*, and*
if $\mathcal{E} : \Gamma_1; \Delta_1 \rightsquigarrow^*_\Sigma \Gamma_2; \Delta_2$*, then* $\mathcal{E} : \Gamma, \Gamma_1; \Delta_L, \Delta_1, \Delta_R \rightsquigarrow^*_\Sigma \Gamma, \Gamma_2; \Delta_L, \Delta_2, \Delta_R$

4.11 Logic programming

I created the Ollibot logic programming language for the purpose of representing the examples from the LICS 2009 paper on substructural operational semantics in ordered logic.[9] It implements a committed-choice operational semantics for the forward-chaining portion of this language; in other words, the current incarnation of Ollibot can execute the parse/... rules as a logic program but not the calc/... rules — the logic programming semantics for the latter rules would require backward chaining to search for a proof of add N M X where N and M are known and X is unknown.

[9] http://ollibot.hyperkind.org/examples-0.1/

Ollibot may or may not form the core of the eventual tool for verifying safety properties of SSOS specifications, but I am interested in extending the expressiveness of Ollibot in a number of ways in the course of this thesis. In particular, adding a backward-chaining semantics for the deductive fragment of the logical framework will significantly improve the language's ability to act as an interpreter for the SSOS specifications such as the ones introduced in the next section.

I am also interested in improving this efficiency of the (not the least bit efficient) Ollibot, both by adapting compilation techniques from rewriting systems like Maude and by investigating the implementation of Ollibot in parallel programming languages like X10 [CGS+05].

4.12 Notational conventions

We have discussed a number of notational conventions, such as omitting the shift operator $\uparrow$ and simplifying the way that goals are written down. In discussions of SSOS in Section 5, we will use a more concise shorthand notation for describing transitions and expressions; this section will introduce this shorthand notation.

Consider the last transition in the parse of our example $\mathcal{L}_0$ program:

$$\begin{aligned}&\mathsf{parse/plus}(y_1 \cdot x_2 \cdot y_3) \rhd (y)\\ &\qquad : \quad (y_1 : \mathsf{parsed}(\mathsf{n}(\mathsf{s}\,\mathsf{z})), x_2 : +, y_3 : \mathsf{parsed}(\mathsf{n}\,\mathsf{z})) \quad \rightsquigarrow_{\Sigma_{\mathsf{parse}}} \quad (y : \mathsf{parsed}(\mathsf{plus}\,(\mathsf{n}(\mathsf{s}\,\mathsf{z}))\,(\mathsf{n}\,\mathsf{z})))\end{aligned}$$

The variables that describe what parts of the context changed are not strictly necessary here; we can convey the exact same information in a more concise form by omitting the mention of any variables:

$$\mathsf{parse/plus} \;:\; \mathsf{parsed}(\mathsf{n}(\mathsf{s}\,\mathsf{z})), +, \mathsf{parsed}(\mathsf{n}\,\mathsf{z}) \;\rightsquigarrow_{\Sigma_{\mathsf{parse}}}\; \mathsf{parsed}(\mathsf{plus}\,(\mathsf{n}(\mathsf{s}\,\mathsf{z}))\,(\mathsf{n}\,\mathsf{z}))$$

This notation must be used carefully: for instance, if we say that the complete parse is represented by the expression (parse/num; parse/num; parse/plus), it is unclear whether the first token or the last token was rewritten first (although in this case, both of the traces are concurrently equivalent!)

It is not always the case that the name of a rule alone is sufficient. The exception is when one of the premises to a rule is a canonical form, such as the last step in the calculation of our example $\mathcal{L}_0$ program:

$$\begin{aligned}&\mathsf{calc/plus}(y_1 \cdot x_2 \cdot y_3 \cdot (\mathsf{add/s\ add/z})) \rhd (y)\\ &\qquad : \quad (y_1 : \mathsf{calc}(\mathsf{s}\,\mathsf{z}), x_2 : +, y_3 : \mathsf{calc}\,\mathsf{z}) \quad \rightsquigarrow_{\Sigma_{\mathsf{calc}}} \quad (y : \mathsf{calc}(\mathsf{s}\,\mathsf{z}))\end{aligned}$$

The shorthand version of this rule only mentions the canonical form $(\mathsf{add/s}\,\mathsf{add/z} : \mathsf{add}\,(\mathsf{s}\,\mathsf{z})\,\mathsf{z}\,(\mathsf{s}\,\mathsf{z}))$ — unlike variable names, that part of the original transition conveys more information than just a position.

$$\mathsf{calc/plus(add/s\ add/z)} \;:\; \mathsf{calc}(\mathsf{s}\,\mathsf{z}), +, \mathsf{calc}\,\mathsf{z} \;\rightsquigarrow_{\Sigma_{\mathsf{calc}}}\; \mathsf{calc}(\mathsf{s}\,\mathsf{z})$$

Finally, in the following section I will refer to states $\Gamma; \Delta$ generally as $\mathcal{S}$, and will use the notation $\mathcal{S}[\Delta]$ to describe a state that includes somewhere inside of it the ephemeral context Δ — a "one-hole context" over contexts, in other words. For instance, we can write the following:

$$\begin{aligned}\mathsf{parse/num} \;&:\; \mathcal{S}[\mathsf{num}(\mathsf{s}(\mathsf{s}\,\mathsf{z}))] \;\rightsquigarrow_{\Sigma_{\mathsf{parse}}}\; \mathcal{S}[\mathsf{parsed}(\mathsf{s}(\mathsf{s}\,\mathsf{z}))]\\ \mathsf{calc/num} \;&:\; \mathcal{S}[\mathsf{num}(\mathsf{s}(\mathsf{s}\,\mathsf{z}))] \;\rightsquigarrow_{\Sigma_{\mathsf{calc}}}\; \mathcal{S}[\mathsf{calc}(\mathsf{s}(\mathsf{s}\,\mathsf{z}))]\\ \mathsf{calc/plus(plus/z)} \;&:\; \mathcal{S}[\mathsf{calc}\,\mathsf{z}, +, \mathsf{calc}(\mathsf{s}(\mathsf{s}\,\mathsf{z}))] \;\rightsquigarrow_{\Sigma_{\mathsf{calc}}}\; \mathcal{S}[\mathsf{calc}(\mathsf{s}(\mathsf{s}\,\mathsf{z}))]\end{aligned}$$

5 Substructural operational semantics

Now that we have discussed the outlines of our logical framework, we can finally return to the discussion of substructural operational semantics specifications that we motivated in the introduction and in Section 2. In this section, we consider a series of relatively simple substructural operational semantics specifications, mostly taken from [PS09]. Substructural operational semantics (or SSOS) is a style of specifying the operational semantics of programming languages as transitions in substructural logics. Its origin lies in a set of examples illustrating the expressiveness of the logical framework CLF [CPWW02], though elements were anticipated by Chirimar's Ph.D. thesis [Chi95] and by the continuation-based specification of ML with references in LLF [CP02]. A methodology of SSOS specifications was refined and generalized to include other substructural logics in subsequent work [Pfe04, PS09].

In this section, we will develop a base language $\mathcal{L}_1$ that extends $\mathcal{L}_0$ with functions and function application (Section 5.1) and then extend that language with parallel evaluation of pairs (Section 5.3) and with exceptions and exception handling (Section 5.4). The novel content of this section is a presentation of the *static semantics* of this language and proof of language safety via progress and preservation lemmas that it enables. We give a proof of safety for $\mathcal{L}_1$ in Section 5.2, and then briefly discuss how that safety proof can be extended when we consider the static semantics of parallel evaluation and exceptions.

Fundamental components of SSOS specifications Many of the elements of substructural operational semantics specifications are immediately familiar to anyone who has written or read traditional SOS-style operational semantics formalized in Twelf. The syntax, for instance, consists of expressions E, types T, stack frames F, and a judgment $(\mathsf{value}\,\mathsf{V})$ capturing the subsort of expressions that are also values (we continue to write V for those expressions which are known to be values).

$\mathsf{exp} : \mathsf{type}.$
$\mathsf{tp} : \mathsf{type}.$
$\mathsf{frame} : \mathsf{type}.$
$\mathsf{value} : \mathsf{exp} \to \mathsf{type}.$

Particular to SSOS specifications are three groups of propositions. The *active* propositions are those that can always eagerly participate in a transition; the prototypical active proposition is $(\mathsf{eval}\,\mathsf{E})$, an ordered atomic proposition containing an expression E that we are trying to evaluate to a value.

$\mathsf{eval} : \mathsf{exp} \to \mathsf{type}_o.$

The *latent* propositions represent suspended computations. The primary latent proposition we will consider is $(\mathsf{cont}\,\mathsf{F})$, which stores a part of a continuation (in the form of a stack frame) waiting for a value to be returned.[10]

$\mathsf{cont} : \mathsf{frame} \to \mathsf{type}_o.$

Finally, *passive* propositions are those that do not drive transitions on their own, but which may take part in transitions when combined with latent propositions. The only passive proposition in the SSOS specification of an effect-free languages is $(\mathsf{retn}\,\mathsf{V})$, which holds a value being returned from the evaluation of an expression. On its own, it is passive, representing a completed computation. If there is a latent stack frame to its left, a transition should occur by returning the value to the stack frame.

[10] Latent propositions can also be seen as artifacts of kind of defunctionalization that is performed on "higher-order" SSOS specifications to make them more amenable to extension and analysis. (See Section 6.4 for a further discussion.)

$\mathsf{retn} : \mathsf{exp} \to \mathsf{type}_o.$

5.1 Specifying $\mathcal{L}_1$

We will introduce SSOS specifications by encoding the operational semantics of the very simple language $\mathcal{L}_0$ that has been used as running example; we extend $\mathcal{L}_0$ with call-by-value functions to make the language less boring, and call the result $\mathcal{L}_1$.

5.1.1 Syntax

The syntax of expressions can be given by the following BNF specification:

$$e ::= x \mid \lambda x.e \mid e_1(e_2) \mid n \mid e_1 + e_2$$

The encoding of that BNF is standard, including the use of higher-order abstract syntax to represent binding. The judgments v/n and v/lam indicate that functions and numbers are values.

$\mathsf{lam} : (\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp}.$
$\mathsf{app} : \mathsf{exp} \to \mathsf{exp} \to \mathsf{exp}.$
$\mathsf{n} : \mathsf{nat} \to \mathsf{exp}.$
$\mathsf{plus} : \mathsf{exp} \to \mathsf{exp} \to \mathsf{exp}.$
$\mathsf{v/lam} : \mathsf{value}\ (\mathsf{lam}\ \lambda \mathsf{x}.\ \mathsf{E}\ \mathsf{x}).$
$\mathsf{v/n} : \mathsf{value}\ (\mathsf{n}\ \mathsf{N}).$

We also need to define the syntax of frames.

$$f ::= (-)(e_2) \mid v_1(-) \mid (-) + e \mid v + (-)$$

$\mathsf{app}_1 : \mathsf{exp} \to \mathsf{frame}.$ — *this is* $(-)(e_2)$
$\mathsf{app}_2 : \mathsf{exp} \to \mathsf{frame}.$ — *this is* $v_1(-)$
$\mathsf{plus}_1 : \mathsf{exp} \to \mathsf{frame}.$ — *this is* $(-) + e_2$
$\mathsf{plus}_2 : \mathsf{exp} \to \mathsf{frame}.$ — *this is* $v_1 + (-)$

5.1.2 Dynamic semantics

The dynamic semantics for $\mathcal{L}_1$ is straightforward by analogy with an abstract machine for control. A function is a value, and therefore it is always returned immediately, and a function application first evaluates the function part to a value, then evaluates the argument part to a value, and then finally substitutes the argument into the function. In the usual style of higher-order abstract syntax, substitution is performed by application in the rule $\mathsf{e/app}_2$.

$\mathsf{e/lam} :\ \mathsf{eval}(\mathsf{lam}\ \lambda \mathsf{x}.\ \mathsf{E}\ \mathsf{x}) \twoheadrightarrow \mathsf{retn}(\mathsf{lam}\ \lambda \mathsf{x}.\ \mathsf{E}\ \mathsf{x}).$
$\mathsf{e/app} :\ \mathsf{eval}(\mathsf{app}\ \mathsf{E}_1\ \mathsf{E}_2) \twoheadrightarrow \mathsf{cont}(\mathsf{app}_1\ \mathsf{E}_2) \cdot \mathsf{eval}\ \mathsf{E}_1.$
$\mathsf{e/app}_1 :\ \mathsf{cont}(\mathsf{app}_1\ \mathsf{E}_2) \cdot \mathsf{retn}\ \mathsf{V}_1 \twoheadrightarrow \mathsf{cont}(\mathsf{app}_2\ \mathsf{V}_1) \cdot \mathsf{eval}\ \mathsf{E}_2.$
$\mathsf{e/app}_2 :\ \mathsf{cont}(\mathsf{app}_2\ (\mathsf{lam}\ \lambda \mathsf{x}.\ \mathsf{E}_0\ \mathsf{x})) \cdot \mathsf{retn}\ \mathsf{V}_2 \twoheadrightarrow \mathsf{eval}(\mathsf{E}_0\ \mathsf{V}_2).$

With one exception, the dynamic semantics of numbers and addition are just as straightforward. As mentioned before, when both subterms have been evaluated to numerical values, there must be a primitive operation that actually adds the two numbers together. As discussed in the context of the calc/... rules in the previous section, the solution is to rely on the persistent type family

(add N M P) that adequately encodes the inductive definition of addition for natural numbers. Given this definition, the dynamic semantics of addition are also straightforward: a number is a value, and to evaluate $\mathsf{plus}\,\mathsf{E}_1\,\mathsf{E}_2$ we evaluate E_1, then evaluate E_2, and then add the resulting numbers.

$$\mathsf{e/n}: \quad \mathsf{eval(nat\ N)} \twoheadrightarrow \mathsf{retn(nat\ N)}.$$
$$\mathsf{e/plus}: \quad \mathsf{eval(plus\ E_1\ E_2)} \twoheadrightarrow \mathsf{cont(plus_1\ E_2)} \cdot \mathsf{eval\ E_1}.$$
$$\mathsf{e/plus_1}: \quad \mathsf{cont(plus_1\ E_2)} \cdot \mathsf{retn(V_1)} \twoheadrightarrow \mathsf{cont(plus_2\ V_1)} \cdot \mathsf{eval\ E_2}.$$
$$\mathsf{e/plus_2}: \quad \mathsf{cont(plus_2\ (n\ N_1))} \cdot \mathsf{retn(n\ N_2)} \cdot \mathsf{!add\ N_1\ N_2\ N_3} \twoheadrightarrow \mathsf{retn(n\ N_3)}.$$

5.1.3 Example trace

As an example, Figure 1 shows a complete evaluation of an $\mathcal{L}_1$ expression that we would write on paper as $((\lambda x.\, 2 + x)(1 + 5))$. The expression represented by the series of transitions on the right has the following type:

$$\mathsf{eval(app\,(lam}(\lambda x.\mathsf{plus\,(n(s(s\,z)))}\,x))\,\mathsf{(plus\,(n(s\,z))\,(n(s(s(s(s\,z))))))})$$
$$\leadsto^*_{\Sigma_\mathsf{e}} \quad \mathsf{retn(n(s(s(s(s(s(s(s(s(s\,z))))))))))}$$

For brevity and clarity, the on-paper notation is used in the figure.

	$\mathsf{eval}((\lambda x.\, 2 + x)(1 + 5))$		
$\leadsto_{\Sigma_\mathsf{e}}$	$\mathsf{cont}((-)(1 + 5)),\ \mathsf{eval}(\lambda x.\, 2 + x)$		by e/app
$\leadsto_{\Sigma_\mathsf{e}}$	$\mathsf{cont}((-)(1 + 5)),\ \mathsf{retn}(\lambda x.\, 2 + x)$		by e/lam
$\leadsto_{\Sigma_\mathsf{e}}$	$\mathsf{cont}((\lambda x.\, 2 + x)(-)),\ \mathsf{eval}(1 + 5)$		by $\mathsf{e/app_1}$
$\leadsto_{\Sigma_\mathsf{e}}$	$\mathsf{cont}((\lambda x.\, 2 + x)(-)),\ \mathsf{cont}((-) + 5),\ \mathsf{eval}(1)$		by e/plus
$\leadsto_{\Sigma_\mathsf{e}}$	$\mathsf{cont}((\lambda x.\, 2 + x)(-)),\ \mathsf{cont}((-) + 5),\ \mathsf{retn}(1)$		by e/n
$\leadsto_{\Sigma_\mathsf{e}}$	$\mathsf{cont}((\lambda x.\, 2 + x)(-)),\ \mathsf{cont}(1 + (-)),\ \mathsf{eval}(5)$		by $\mathsf{e/plus_1}$
$\leadsto_{\Sigma_\mathsf{e}}$	$\mathsf{cont}((\lambda x.\, 2 + x)(-)),\ \mathsf{cont}(1 + (-)),\ \mathsf{retn}(5)$		by e/n
$\leadsto_{\Sigma_\mathsf{e}}$	$\mathsf{cont}((\lambda x.\, 2 + x)(-)),\ \mathsf{retn}(6)$		by $\mathsf{e/plus_2}$(add/s add/z)
$\leadsto_{\Sigma_\mathsf{e}}$	$\mathsf{eval}(2 + 6)$		by $\mathsf{e/app_2}$
$\leadsto_{\Sigma_\mathsf{e}}$	$\mathsf{cont}((-) + 6),\ \mathsf{eval}(2)$		by e/plus
$\leadsto_{\Sigma_\mathsf{e}}$	$\mathsf{cont}((-) + 6),\ \mathsf{retn}(2)$		by e/n
$\leadsto_{\Sigma_\mathsf{e}}$	$\mathsf{cont}(2 + (-)),\ \mathsf{eval}(6)$		by $\mathsf{e/plus_1}$
$\leadsto_{\Sigma_\mathsf{e}}$	$\mathsf{cont}(2 + (-)),\ \mathsf{retn}(6)$		by e/n
$\leadsto_{\Sigma_\mathsf{e}}$	$\mathsf{retn}(8)$		by $\mathsf{e/plus_2}$(add/s(add/s add/z))

Figure 1: Example trace in $\mathcal{L}_1$.

5.2 Static semantics, progress, and preservation

The description of the dynamic semantics of $\mathcal{L}_1$ is not enough on its own; there's nothing we have said so far that prevents us from attempting to evaluate $5 + \lambda x.x$ or $3(\lambda x.x + 2)$, both of which would get *stuck*. A state is stuck if it has not evaluated all the way to the single atomic proposition

($\mathsf{retn}\,\mathsf{E}$) but which nevertheless cannot take a step. A safe expression is one that cannot lead to a stuck state:

Definition 1 (Safety)**.** *The expression* E *is* **safe** *if* ($\mathsf{eval}\,\mathsf{E} \rightsquigarrow^*_{\Sigma\mathsf{e}} \mathcal{S}$) *implies that either* $\mathcal{S} \rightsquigarrow_{\Sigma\mathsf{e}} \mathcal{S}'$ *for some* $\mathcal{S}'$ *or else* $\mathcal{S} = \mathsf{retn}\,\mathsf{V}$ *for some* V*.*

While this definition of safety is appropriate for our current purposes, it is not necessarily the only definition of safety we might want to consider. For instance, a system that might deadlock is usually considered to be safe, but would not be safe according to this definition.

Integral to establishing safety will be the static semantics, a new set of rewriting rules annotated with $\mathsf{t}/\ldots$ (so we use $\Sigma\mathsf{t}$ to refer to them collectively). The goal of these rewriting rules is to rewrite the entire state $\mathcal{S}$ to a single atomic proposition $\mathsf{abs}\,\mathsf{T}$. The atomic proposition $\mathsf{abs}\,\mathsf{T}$ is intended to represent an abstraction of states that will produce a value of type T if they produce any value at all.[11] These rewriting rules, which we will refer to as the static semantics, rely on the not-yet-defined judgments $\mathsf{of}\,\mathsf{E}\,\mathsf{T}$ (the expression E has type T) and $\mathsf{off}\,\mathsf{F}\,\mathsf{T}\,\mathsf{T}'$ (the frame F takes values of type T to states of type T'). The rules themselves are straightforward:

$\mathsf{abs} : \mathsf{tp} \to \mathsf{type}_o.$

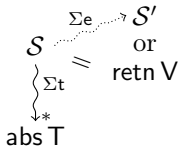
$\mathsf{t/eval}: \quad \mathsf{eval}\ \mathsf{E} \cdot !\mathsf{of}\ \mathsf{E}\ \mathsf{T} \twoheadrightarrow \mathsf{abs}\ \mathsf{T}.$

$\mathsf{t/retn}: \quad \mathsf{retn}\ \mathsf{E} \cdot !\mathsf{value}\ \mathsf{E} \cdot !\mathsf{of}\ \mathsf{E}\ \mathsf{T} \twoheadrightarrow \mathsf{abs}\ \mathsf{T}.$

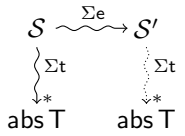
$\mathsf{t/cont}: \quad \mathsf{cont}\ \mathsf{F} \cdot \mathsf{abs}\ \mathsf{T} \cdot !\mathsf{off}\ \mathsf{F}\ \mathsf{T}\ \mathsf{T}' \twoheadrightarrow \mathsf{abs}\ \mathsf{T}'.$

These static semantics (together with the not-yet-specified typing rules) allow us to prove progress and preservation theorems, which in turn allow us to establish type safety for the language.

Progress If $\mathcal{S} \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\,\mathsf{T}$ and $\mathcal{S}$ contains no resources $\mathsf{abs}\,\mathsf{T}$, then either $\mathcal{S} \rightsquigarrow_{\Sigma\mathsf{e}} \mathcal{S}'$ for some $\mathcal{S}'$ or else $\mathcal{S} = \mathsf{retn}\,\mathsf{V}$.



Preservation If $\mathcal{S} \rightsquigarrow_{\Sigma\mathsf{e}} \mathcal{S}'$ and $\mathcal{S} \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\,\mathsf{T}$, then $\mathcal{S}' \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\,\mathsf{T}$.

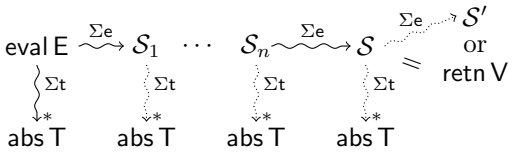


[11] This characterization of $\mathsf{abs}\,\mathsf{T}$ is actually a corollary of the preservation lemma.

Given progress and preservation as stated above, we can prove the desired type safety theorem:

Theorem 1. *If* $\vdash \mathsf{of}\,\mathsf{E}\,\mathsf{T}$*, then* E *is safe.*

Proof. The overall picture for safety is this one:



We are given an expression ($\mathsf{eval}\,\mathsf{E} \rightsquigarrow^*_{\Sigma\mathsf{e}} \mathcal{S}$), and we must show that either $\mathcal{S} \rightsquigarrow_{\Sigma\mathsf{e}} \mathcal{S}'$ for some $\mathcal{S}'$ or else $\mathcal{S} = \mathsf{retn}\,\mathsf{V}$ for some V. By rule $\mathsf{t/eval}$ and the premise, ($\mathsf{eval}\,\mathsf{E} \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\,\mathsf{T}$), and by induction on the length of the trace and progress lemma, ($\mathcal{S} \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\,\mathsf{T}$). By a similar induction on the trace and the observation that $\mathsf{abs}\,\mathsf{T}$ appears nowhere in any of the rules of the form $\mathsf{e}/\ldots$, $\mathcal{S}$ contains no resources $\mathsf{abs}\,\mathsf{T}$. The result then follows by the progress lemma. □

Note that both the proof of type safety and the statements of the progress and preservation theorems talk about individual transition steps and about taking a series of transitions and extending them with an additional transition. Both of these notions are naturally represented by the transitions and expressions of our logical framework, but neither of these are easy to represent in the CLF framework. The CLF framework has proof terms that express a completed derivations that has ended by proving something, but a series of transitions in our framework translates to a *partial* derivation in CLF, not a complete derivation. It is for this reason that I previously described the focus on a state-transition-based presentation of logic as the most critical difference between CLF and the framework presented in this proposal.

In the remainder of this section, we will present the typing rules and static semantics for $\mathcal{L}_1$, and then give proofs of the progress and preservation lemmas above.

5.2.1 Static semantics

Typing rules are completely conventional. The syntax of types can be written as a BNF specification as $\tau ::= \mathsf{nat} \mid \tau \to \tau$, and the signature for types is the following:

```
tnat : tp.
arrow : tp → tp → tp.
```

The typing rules are the familiar ones:

```
of : exp → tp → type.
of/lam :  (Πx. of x T → of (E x) T') → of (lam λx. E x) (arrow T T')
of/app :  of E1 (arrow T' T) → of E2 T' → of (app E1 E2) T
of/n :    of (nat N) tnat.
of/plus :  of E1 tnat → of E2 tnat → of (plus E1 E2) tnat
```

Frame typing rules are also straightforward:

```
off/app1 :   of E2 T' → off (app1 E2) (arrow T' T) T.
off/app2 :   value E1 → of E1 (arrow T' T) → off (app2 E2) T' T.
off/plus1 :  of E2 tnat → off (plus1 E2) tnat tnat.
off/plus2 :  value E1 → of E1 tnat → off (plus2 E1) tnat tnat.
```

5.2.2 Preservation

The preservation lemma establishes that our purported invariant is actually an invariant. The form of the preservation lemma should be very familiar: if the invariant described by the static semantics holds of a state, the invariant still holds of the state after any transition made under the dynamic semantics.

Lemma 1 (Preservation)**.** *If* $\mathcal{S} \rightsquigarrow_{\Sigma\mathsf{e}} \mathcal{S}'$ *and* $\mathcal{S} \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\,\mathsf{T_f}$*, then* $\mathcal{S}' \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\,\mathsf{T_f}$*.*

Preservation proofs tend to rely on typing inversion lemmas, and we use traditional typing inversion lemmas as needed without proof. However, there is another lemma that is critical for preservation proofs for SSOS specifications. It captures the intuition that if $\mathcal{S} \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\,\mathsf{T}$, either $\mathcal{S}$ is already a substructural context containing a single resource ($\mathsf{abs}\,\mathsf{T}$) or else everything in $\mathcal{S}$ must eventually be consumed in the service of producing a single resource ($\mathsf{abs}\,\mathsf{T}$). We call this the *consumption lemma* because it represents the obligation that the static semantics consume everything in the context. Recall that $\mathcal{S}[\Delta]$ is notation representing a state $\mathcal{S}$ containing the ephemeral state Δ inside of it.

Lemma 2 (Consumption)**.**

- *If* $\mathcal{E} : \mathcal{S}[\mathsf{eval}\,\mathsf{E}] \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\,\mathsf{T_f}$*,*
  then $\mathcal{E} \approx (\mathsf{t/eval}(\mathcal{D}_t);\ \mathcal{E}')$*, where* $\mathcal{D}_t : \mathsf{of}\,\mathsf{E}\,\mathsf{T}$ *and* $\mathcal{E}' : \mathcal{S}[\mathsf{abs}\,\mathsf{T}] \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\,\mathsf{T_f}$ *for some* T*.*
- *If* $\mathcal{E} : \mathcal{S}[\mathsf{retn}\,\mathsf{V}] \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\,\mathsf{T_f}$*,*
  then $\mathcal{E} \approx (\mathsf{t/retn}(\mathcal{D}_v \cdot \mathcal{D}_t);\ \mathcal{E}')$*, where* $\mathcal{D}_v : \mathsf{value}\,\mathsf{V}$*,* $\mathcal{D}_t : \mathsf{of}\,\mathsf{V}\,\mathsf{T}$*, and* $\mathcal{E}' : \mathcal{S}[\mathsf{abs}\,\mathsf{T}] \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\,\mathsf{T_f}$ *for some* T*.*
- *If* $\mathcal{E} : \mathcal{S}[\mathsf{cont}\,\mathsf{F}, \mathsf{abs}\,\mathsf{T}] \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\,\mathsf{T_f}$*,*
  then $\mathcal{E} \approx (\mathsf{t/cont}(\mathcal{D}_f);\ \mathcal{E}')$*, where* $\mathcal{D}_f : \mathsf{off}\,\mathsf{F}\,\mathsf{T}\,\mathsf{T}'$ *and* $\mathcal{E}' : \mathcal{S}[\mathsf{abs}\,\mathsf{T}'] \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\,\mathsf{T_f}$ *for some* T'*.*

Proof. Each of the statements can be proved independently by induction on the structure of $\mathcal{E}$. The eval and retn cases are quite similar to each other and are both simpler versions of the cont case, which we give below.

Case: $\mathcal{E} = \diamond$

This case cannot occur, as there is no way for $\mathcal{S}[\mathsf{cont}\,\mathsf{F}, \mathsf{abs}\,\mathsf{T}]$ to be equivalent to $(\mathsf{abs}\,\mathsf{T_f})$.

Case: $\mathcal{E} = \mathsf{t/eval}(\mathcal{D}'_t);\ \mathcal{E}'$
$\mathcal{D}'_t : \mathsf{of}\,\mathsf{E}\,\mathsf{T_e}$,
$\mathcal{E}' : (\mathcal{S}'[\mathsf{abs}\,\mathsf{T_e}] \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\,\mathsf{T_f})$, and
$\mathcal{S}[\mathsf{cont}\,\mathsf{F}, \mathsf{abs}\,\mathsf{T}] \approx \mathcal{S}'[\mathsf{eval}\,\mathsf{E}]$

Because $\mathcal{S}[\mathsf{cont}\,\mathsf{F}, \mathsf{abs}\,\mathsf{T}] \approx \mathcal{S}'[\mathsf{eval}\,\mathsf{E}]$, it must be the case that both of these are also equivalent to $\mathcal{S}''[\mathsf{cont}\,\mathsf{F}, \mathsf{abs}\,\mathsf{T}][\mathsf{eval}\,\mathsf{E}]$ for some $\mathcal{S}'$ and $\mathcal{E}' : (\mathcal{S}''[\mathsf{cont}\,\mathsf{F}, \mathsf{abs}\,\mathsf{T}][\mathsf{abs}\,\mathsf{T_e}] \rightsquigarrow^*_{\Sigma_t} \mathsf{abs}\,\mathsf{T_f})$.

$\mathcal{E}' \approx (\mathsf{t/cont}(\mathcal{D}_f);\ \mathcal{E}'')$	By i.h.
$\mathcal{D}_f : \mathsf{off}\,\mathsf{E}\,\mathsf{T}\,\mathsf{T}'$	”
$\mathcal{E}'' : (\mathcal{S}''[\mathsf{abs}\,\mathsf{T}'][\mathsf{abs}\,\mathsf{T_e}] \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\,\mathsf{T_f})$	”
$\mathcal{E} \approx (\mathsf{t/eval}(\mathcal{D}'_t);\ \mathsf{t/cont}(\mathcal{D}_f);\ \mathcal{E}'')$	By construction
$\mathcal{E} \approx (\mathsf{t/cont}(\mathcal{D}_f);\ \mathsf{t/eval}(\mathcal{D}'_t);\ \mathcal{E}'')$	By concurrent equivalence

Case: $\mathcal{E} = \mathsf{t/retn}(\mathcal{D}'_v \cdot \mathcal{D}'_t);\ \mathcal{E}'$

Similar to previous case.

Case: $\mathcal{E} = \mathsf{t/cont}(\mathcal{D}'_f);\ \mathcal{E}'$
$\mathcal{D}'_f : \mathsf{off}\ \mathsf{F}_1\ \mathsf{T}_1\ \mathsf{T}'_1$,
$\mathcal{E}' : (\mathcal{S}'[\mathsf{abs}\ \mathsf{T}'_1] \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\ \mathsf{T}_\mathsf{f})$, and
$\mathcal{S}[\mathsf{cont}\ \mathsf{F}, \mathsf{abs}\ \mathsf{T}] \approx \mathcal{S}'[\mathsf{cont}\ \mathsf{F}_1, \mathsf{abs}\ \mathsf{T}_1]$

Because $\mathcal{S}[\mathsf{cont}\ \mathsf{F}, \mathsf{abs}\ \mathsf{T}] \approx \mathcal{S}'[\mathsf{cont}\ \mathsf{F}_1, \mathsf{abs}\ \mathsf{T}_1]$, there are two possibilities. The first possibility is that $\mathcal{S}[\mathsf{abs}\ \mathsf{T}'] \approx \mathcal{S}'[\mathsf{abs}\ \mathsf{T}']$, $\mathsf{F} = \mathsf{F}_1$, and $\mathsf{T} = \mathsf{T}_1$, in which case we are done.

The other possibility is $\mathcal{S}[\mathsf{cont}\ \mathsf{F}, \mathsf{abs}\ \mathsf{T}] \approx \mathcal{S}'[\mathsf{cont}\ \mathsf{F}_1, \mathsf{abs}\ \mathsf{T}_1] \approx \mathcal{S}''[\mathsf{cont}\ \mathsf{F}, \mathsf{abs}\ \mathsf{T}][\mathsf{cont}\ \mathsf{F}_1, \mathsf{abs}\ \mathsf{T}_1]$ for some $\mathcal{S}''$. In that case, $\mathcal{E}' : (\mathcal{S}''[\mathsf{cont}\ \mathsf{F}, \mathsf{abs}\ \mathsf{T}][\mathsf{abs}\ \mathsf{T}'_1])$ and we proceed with the following reasoning:

$\mathcal{E}' \approx (\mathsf{t/cont}(\mathcal{D}_f); \mathcal{E}'')$	By i.h.
$\mathcal{D}_f : \mathsf{off}\ \mathsf{E}\ \mathsf{T}\ \mathsf{T}'$	”
$\mathcal{E}'' \approx (\mathcal{S}''[\mathsf{abs}\ \mathsf{T}'][\mathsf{abs}\ \mathsf{T}'_1])$	”
$\mathcal{E} \approx (\mathsf{t/cont}(\mathcal{D}'_f);\ \mathsf{t/cont}(\mathcal{D}_f);\ \mathcal{E}'')$	By construction
$\mathcal{E} \approx (\mathsf{t/cont}(\mathcal{D}_f);\ \mathsf{t/cont}(\mathcal{D}'_f);\ \mathcal{E}'')$	By concurrent equivalence

This completes the proof. □

It's worth emphasizing what may be obvious: the proof of the consumption lemma is a bit more general than necessary. In the static semantics we have defined so far, $\mathcal{S} \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\ \mathsf{T}$ means that $\mathcal{S}$ contains *precisely one* atomic proposition of the form $(\mathsf{eval}\ \mathsf{E})$, $(\mathsf{retn}\ \mathsf{E})$, or $(\mathsf{abs}\ \mathsf{T})$, which means that most of the cases we consider are actually impossible. However, this impossibility is *also* a fact that must be proven by induction on the structure of expressions. The proof we have given is both straightforward in its own right and allows the proof of preservation to "scale" to specifications with parallel evaluation (see Section 5.3). Furthermore, the theorem *statement* is not at all complicated as a result of handling the general case. If we wanted to write a version of the consumption lemma for the special case of sequential SSOS specifications, the only difference is that we would write "If $\mathcal{E} : (\mathcal{S}, \mathsf{eval}\ \mathsf{E} \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\ \mathsf{T}_\mathsf{f})$, then..." instead of "If $\mathcal{E} : (\mathcal{S}[\mathsf{eval}\ \mathsf{E}] \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\ \mathsf{T}_\mathsf{f})$, then..." because we know that $\mathsf{eval}\ \mathsf{E}$ appears only on the right-hand edge of the ephemeral context.

Now we can consider the actual proof of preservation:

Proof of the Preservation Lemma (Lemma 1). The proof proceeds by case analysis on the transition $\mathcal{T} : (\mathcal{S} \rightsquigarrow_{\Sigma\mathsf{e}} \mathcal{S}')$, followed in each case by applying the consumption lemma to the expression $\mathcal{E} : (\mathcal{S} \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\ \mathsf{T}_\mathsf{f})$.

Case: $\mathcal{T} = \mathsf{e/lam}\ :\ \mathcal{S}[\mathsf{eval}(\mathsf{lam}\ \lambda\mathsf{x}.\ \mathsf{E}\,\mathsf{x})] \rightsquigarrow_{\Sigma\mathsf{e}} \mathcal{S}[\mathsf{retn}(\mathsf{lam}\ \lambda\mathsf{x}.\ \mathsf{E}\,\mathsf{x})]$

$\mathcal{E} \approx (\mathsf{t/eval}(\mathcal{D}_t);\ \mathcal{E}')$	By consumption lemma
$\mathcal{D}_t : \mathsf{of}\ (\mathsf{lam}\ \lambda\mathsf{x}.\ \mathsf{E}\,\mathsf{x})\ \mathsf{T}$	”
$\mathcal{E}' : (\mathcal{S}[\mathsf{abs}\ \mathsf{T}] \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\ \mathsf{T}_\mathsf{f})$	”
$(\mathsf{t/retn}(\mathsf{v/lam} \cdot \mathcal{D}_t);\ \mathcal{E}') : (\mathcal{S}[\mathsf{retn}(\mathsf{lam}\ \lambda\mathsf{x}.\ \mathsf{E}\,\mathsf{x})] \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\ \mathsf{T}_\mathsf{f})$	By construction

Case: $\mathcal{T} = \mathsf{e/app}\ :\ \mathcal{S}[\mathsf{eval}(\mathsf{app}\,\mathsf{E}_1\,\mathsf{E}_2)] \rightsquigarrow_{\Sigma\mathsf{e}} \mathcal{S}[\mathsf{cont}(\mathsf{app}_1\,\mathsf{E}_2), \mathsf{eval}\,\mathsf{E}_1]$

$\mathcal{E} \approx (\mathsf{t/eval}(\mathcal{D}_t);\ \mathcal{E}')$	By consumption lemma
$\mathcal{D}_t : \mathsf{of}\ (\mathsf{app}\,\mathsf{E}_1\,\mathsf{E}_2)\ \mathsf{T}$	”
$\mathcal{E}' : \mathcal{S}[\mathsf{abs}\ \mathsf{T}] \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\ \mathsf{T}_\mathsf{f}$	”
$\mathcal{D}_t = \mathsf{of/app}\,\mathcal{D}_1\,\mathcal{D}_2$	By inversion on $\mathcal{D}_t$
$\mathcal{D}_1 : \mathsf{of}\ \mathsf{E}_1\ (\mathsf{arrow}\ \mathsf{T}'\ \mathsf{T})$	”

$\mathcal{D}_2 : \mathsf{of}\ \mathsf{E}_2\ \mathsf{T}'$ ”
$(\mathsf{t/cont}(\mathsf{off/app}_1\ \mathcal{D}_2);\ \mathcal{E}') : (\mathcal{S}[\mathsf{cont}(\mathsf{app}_1\ \mathsf{E}_2), \mathsf{abs}(\mathsf{arrow}\ \mathsf{T}'\ \mathsf{T})] \rightsquigarrow^*_{\Sigma\mathsf{t}}\ \mathsf{abs}\ \mathsf{T}_\mathsf{f})$ By construction
$(\mathsf{t/eval}(\mathcal{D}_1);\ \mathsf{t/cont}(\mathsf{off/app}_1\ \mathcal{D}_2);\ \mathcal{E}') : (\mathcal{S}[\mathsf{cont}(\mathsf{app}_1\ \mathsf{E}_2), \mathsf{eval}\ \mathsf{E}_1] \rightsquigarrow^*_{\Sigma\mathsf{t}}\ \mathsf{abs}\ \mathsf{T}_\mathsf{f})$ By construction

Case: $\mathcal{T} = \mathsf{e/app}_1\ :\ \mathcal{S}[\mathsf{cont}(\mathsf{app}_1\ \mathsf{E}_2), \mathsf{retn}\ \mathsf{V}_1]\ \rightsquigarrow_{\Sigma\mathsf{e}}\ \mathcal{S}[\mathsf{cont}(\mathsf{app}_2\ \mathsf{V}_1), \mathsf{eval}\ \mathsf{E}_2]$

$\mathcal{E} \approx (\mathsf{t/retn}(\mathcal{D}_v \cdot \mathcal{D}_t);\ \mathcal{E}')$ By consumption lemma
$\mathcal{D}_v : \mathsf{value}\ \mathsf{V}_1$ ”
$\mathcal{D}_t : \mathsf{of}\ \mathsf{V}_1\ \mathsf{T}$ ”
$\mathcal{E}' : (\mathcal{S}[\mathsf{cont}(\mathsf{app}_1\ \mathsf{E}_2), \mathsf{abs}\ \mathsf{T}] \rightsquigarrow^*_{\Sigma\mathsf{t}}\ \mathsf{abs}\ \mathsf{T}_\mathsf{f})$ ”
$\mathcal{E}' \approx \mathsf{t/cont}(\mathcal{D}_f);\ \mathcal{E}''$ By consumption lemma
$\mathcal{D}_f : \mathsf{off}\ (\mathsf{app}_1\ \mathsf{E}_2)\ \mathsf{T}\ \mathsf{T}'$ ”
$\mathcal{E}'' : (\mathcal{S}[\mathsf{abs}\ \mathsf{T}'] \rightsquigarrow^*_{\Sigma\mathsf{t}}\ \mathsf{abs}\ \mathsf{T}_\mathsf{f})$ ”
$\mathcal{D}_f = \mathsf{off/app}_1(\mathcal{D}'_t)$ By inversion on $\mathcal{D}_f$
$\mathsf{T} = \mathsf{arrow}\ \mathsf{T}_0\ \mathsf{T}'$ ”
$\mathcal{D}'_t : \mathsf{of}\ \mathsf{E}_2\ \mathsf{T}_0$ ”
$(\mathsf{t/cont}(\mathsf{off/app}_2\ \mathcal{D}_v\ \mathcal{D}_t);\ \mathcal{E}'') : (\mathcal{S}[\mathsf{cont}(\mathsf{app}_2\ \mathsf{V}_1), \mathsf{abs}\ \mathsf{T}_0] \rightsquigarrow^*_{\Sigma\mathsf{t}}\ \mathsf{abs}\ \mathsf{T}_\mathsf{f})$ By construction
$(\mathsf{t/eval}(\mathcal{D}'_t);\ \mathsf{t/cont}(\mathsf{off/app}_2\ \mathcal{D}_v\ \mathcal{D}_t);\ \mathcal{E}'') : (\mathcal{S}[\mathsf{cont}(\mathsf{app}_2\ \mathsf{V}_1), \mathsf{eval}\ \mathsf{E}_2] \rightsquigarrow^*_{\Sigma\mathsf{t}}\ \mathsf{abs}\ \mathsf{T}_\mathsf{f})$ By construction

Case: $\mathcal{T} = \mathsf{e/app}_2\ :\ \mathcal{S}[\mathsf{cont}(\mathsf{app}_2(\mathsf{lam}\ \lambda\mathsf{x}.\ \mathsf{E}\,\mathsf{x})), \mathsf{retn}\ \mathsf{V}_2]\ \rightsquigarrow_{\Sigma\mathsf{e}}\ \mathcal{S}[\mathsf{eval}(\mathsf{E}\ \mathsf{V}_2)]$

$\mathcal{E} \approx \mathsf{t/retn}(\mathcal{D}_v \cdot \mathcal{D}_t);\ \mathcal{E}'$ By consumption lemma
$\mathcal{D}_t : \mathsf{of}\ \mathsf{V}_2\ \mathsf{T}$ ”
$\mathcal{E}' : (\mathcal{S}[\mathsf{cont}(\mathsf{app}_2(\mathsf{lam}\ \lambda\mathsf{x}.\ \mathsf{E}\,\mathsf{x})), \mathsf{abs}\ \mathsf{T}] \rightsquigarrow^*_{\Sigma\mathsf{t}}\ \mathsf{abs}\ \mathsf{T}_\mathsf{f})$ ”
$\mathcal{E}' \approx \mathsf{t/cont}(\mathcal{D}_f);\ \mathcal{E}''$ By consumption lemma
$\mathcal{D}_f : \mathsf{off}\ (\mathsf{app}_2(\mathsf{lam}\ \lambda\mathsf{x}.\ \mathsf{E}\,\mathsf{x}))\ \mathsf{T}\ \mathsf{T}'$ ”
$\mathcal{E}'' : (\mathcal{S}[\mathsf{abs}\ \mathsf{T}'] \rightsquigarrow^*_{\Sigma\mathsf{t}}\ \mathsf{abs}\ \mathsf{T}_\mathsf{f})$ ”
$\mathcal{D}_f = \mathsf{off/app}_2\ \mathcal{D}'_v\ (\lambda x.\ \lambda d.\ \mathcal{D}'_t\ x\ d)$ By inversion on $\mathcal{D}_f$
$(\lambda x.\ \lambda d.\ \mathcal{D}'_t\ x\ d) : \Pi x.\ \mathsf{of}\ x\ \mathsf{T} \to \mathsf{of}\ (\mathsf{E}\,x)\ \mathsf{T}'$ ”
$\mathcal{D}'_t\ \mathsf{V}_2\ \mathcal{D}_t : \mathsf{of}\ (\mathsf{E}\ \mathsf{V}_2)\ \mathsf{T}'$ By substitution
$(\mathsf{t/eval}(\mathcal{D}'_t\ \mathsf{V}_2\ \mathcal{D}_t);\ \mathcal{E}'') : (\mathcal{S}[\mathsf{eval}(\mathsf{E}\ \mathsf{V}_2)] \rightsquigarrow^*_{\Sigma\mathsf{t}}\ \mathsf{abs}\ \mathsf{T}_\mathsf{f})$ By construction

The remaining cases for $\mathsf{e/nat}$, $\mathsf{e/plus}$, $\mathsf{e/plus}_1$, and $\mathsf{e/plus}_2$ are similar. □

5.2.3 Progress

Preservation theorems, and in particular big-step preservation theorems, have traditionally been the primary correctness criteria for language specifications in substructural logics [CP02]. However, preservation alone is insufficient to ensure language safety as specified by Definition 1. The preservation lemma ensures that the invariant is maintained; the progress lemma ensures that the invariant actually establishes safety.

The critical lemma for progress, the analogue to the consumption lemma in the preservation lemma, is actually quite a bit more general. It relies on a general property of the static semantics, namely that it is *contractive*, because each rule has at most one conclusion.

Definition 2 (Contractive signature)**.** *If every rule in Σ has at most one atomic proposition in the conclusion, and that conclusion is an ordered or linear atomic proposition, then Σ is contractive.*

When a signature is contractive, then different pieces of the context at the end of an expression can be traced backwards to different pieces at the beginning of the expression.

Lemma 3 (Splitting)**.** *If Σ is contractive and $\mathcal{E} : (\mathcal{S} \rightsquigarrow^*_\Sigma \mathcal{S}'_1, \mathcal{S}'_2)$, then $\mathcal{S} \approx \mathcal{S}_1, \mathcal{S}_2$, and there exist expressions $\mathcal{E}_1$ and $\mathcal{E}_2$ such that $\mathcal{E}_1 : (\mathcal{S}_1 \rightsquigarrow^*_\Sigma \mathcal{S}'_1)$ and $\mathcal{E}_2 : (\mathcal{S}_2 \rightsquigarrow^*_\Sigma \mathcal{S}'_2)$, where $\mathcal{E}$ has the same number of transitions as $\mathcal{E}_1$ and $\mathcal{E}_2$ combined.*

Proof. By induction on the structure of $\mathcal{E}$. If $\mathcal{E} = \diamond$, the result is immediate, and if $\mathcal{E} = (\mathcal{E}';\ \mathcal{T})$ then we apply the induction hypothesis on $\mathcal{E}'$ and add $\mathcal{T}$ to the end of the appropriate smaller expression: $\mathcal{E}_1$ if the conclusion of $\mathcal{T}$ was bound in $\mathcal{S}_1$, $\mathcal{E}_2$ otherwise. □

A stronger version of the splitting lemma would also establish that $\mathcal{E} \approx (\mathcal{E}_1;\ \mathcal{E}_2) \approx (\mathcal{E}_2;\ \mathcal{E}_1)$, but this lemma is sufficient to allow us to prove the progress lemma.

Lemma 4 (Progress)**.** *If $\mathcal{S} \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\,\mathsf{T}$ and $\mathcal{S}$ contains no resources $\mathsf{abs}\,\mathsf{T}$, then either $\mathcal{S} \rightsquigarrow_{\Sigma\mathsf{e}} \mathcal{S}'$ for some $\mathcal{S}'$ or else $\mathcal{S} = \mathsf{retn}\,\mathsf{V}$ for some value V with type T.*

At a high level, this proof works by showing that any state $\mathcal{S}$ that satisfies the typing invariant must be one of the following:

- $\mathcal{S} = \mathsf{retn}\,\mathsf{V}$, a safe state that takes no step,
- $\mathcal{S} = \mathsf{eval}\,\mathsf{E}$, in which case we can always take a step because $\mathsf{eval}\,\mathsf{E}$ is an active proposition,
- $\mathcal{S} = (\mathsf{comp}\,\mathsf{F}, \mathcal{S}')$ where $\mathcal{S}' \rightsquigarrow_{\Sigma\mathsf{e}} \mathcal{S}''$; in this case, the larger state takes a step as well, or
- $\mathcal{S} = (\mathsf{comp}\,\mathsf{F}, \mathsf{retn}\,\mathsf{V})$, in which case we use canonical forms lemmas (which are standard, and which we therefore use without proof) to ensure that we can perform a reduction.

Proof. By induction on the number of transitions in $\mathcal{E}$. We note that $\mathcal{E} \neq \diamond$, because that would mean that $\mathcal{S} = \mathsf{abs}\,\mathsf{T}$, contradicting the premise that $\mathcal{S}$ does not contain any atomic propositions of the form $\mathsf{abs}\,\mathsf{T}$. So we can assume $\mathcal{E} = (\mathcal{E}';\ \mathcal{T})$ and do case analysis on the form of the transition $\mathcal{T}$.

Case: $\mathcal{T} = \mathsf{t}/\mathsf{retn}(\mathcal{D}_v \cdot \mathcal{D}_t)$ and $\mathcal{E}' : \mathcal{S} \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{retn}\,\mathsf{V}$
where $\mathcal{D}_v : \mathsf{value}\,\mathsf{V}$ and $\mathcal{D}_t : \mathsf{of}\,\mathsf{E}\,\mathsf{T}$.

There are no transitions in $\Sigma\mathsf{t}$ that can appear as the last transition in $\mathcal{E}'$, so $\mathcal{E}' = \diamond$ and $\mathcal{S} = \mathsf{retn}\,\mathsf{V}$; therefore, we are done.

Case: $\mathcal{T} = \mathsf{t}/\mathsf{eval}(\mathcal{D}_t)$ and $\mathcal{E}' : \mathcal{S} \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{eval}\,\mathsf{E}$.

There are no transitions in $\Sigma\mathsf{t}$ that can appear as the last transition in $\mathcal{E}'$, so $\mathcal{E}' = \diamond$ and $\mathcal{S} = \mathsf{eval}\,\mathsf{E}$. We proceed by case analysis on the structure of E.

Subcase: $\mathsf{E} = \mathsf{lam}\,\lambda\mathsf{x}.\,\mathsf{E}\,\mathsf{x}$ — $\mathsf{e}/\mathsf{lam} : \mathsf{eval}(\mathsf{lam}\,\lambda\mathsf{x}.\,\mathsf{E}\,\mathsf{x}) \rightsquigarrow_{\Sigma\mathsf{e}} \mathsf{retn}(\mathsf{lam}\,\lambda\mathsf{x}.\,\mathsf{E}\,\mathsf{x})$

Subcase: $\mathsf{E} = \mathsf{app}\,\mathsf{E}_1\,\mathsf{E}_2$ — $\mathsf{e}/\mathsf{app} : \mathsf{eval}(\mathsf{app}\,\mathsf{E}_1\,\mathsf{E}_2) \rightsquigarrow_{\Sigma\mathsf{e}} (\mathsf{cont}(\mathsf{app}_1\,\mathsf{E}_2), \mathsf{eval}\,\mathsf{E}_1)$

Subcase: $\mathsf{E} = \mathsf{n}\,\mathsf{N}$ — $\mathsf{e}/\mathsf{n} : \mathsf{eval}(\mathsf{n}\,\mathsf{N}) \rightsquigarrow_{\Sigma\mathsf{e}} \mathsf{retn}\,(\mathsf{n}\,\mathsf{N})$

Subcase: $\mathsf{E} = \mathsf{plus}\,\mathsf{E}_1\,\mathsf{E}_2$ — $\mathsf{e}/\mathsf{plus} : \mathsf{eval}(\mathsf{plus}\,\mathsf{E}_1\,\mathsf{E}_2) \rightsquigarrow_{\Sigma\mathsf{e}} (\mathsf{cont}(\mathsf{plus}_1\,\mathsf{E}_2), y_2 : \mathsf{eval}\,\mathsf{E}_1)$

Case: $\mathcal{T} = \mathsf{t}/\mathsf{cont}(\mathcal{D}_f)$ and $\mathcal{E}' : (\mathcal{S} \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{cont}\,\mathsf{F}, \mathsf{abs}\,\mathsf{T}')$, where $\mathcal{D}_f : \mathsf{off}\,\mathsf{F}\,\mathsf{T}'\,\mathsf{T}$.

By the splitting lemma, $\mathcal{S} \approx \mathcal{S}_1, \mathcal{S}_2$ and there exist two expressions $\mathcal{E}_1 : (\mathcal{S}_1 \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{cont}\,\mathsf{F})$ and $\mathcal{E}_2 : (\mathcal{S}_2 \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\,\mathsf{T}')$. There are no transitions in $\Sigma\mathsf{t}$ that can appear as the last transition in $\mathcal{E}_1$, so $\mathcal{E}_1 = \diamond$ and $\mathcal{S}_1 = \mathsf{cont}\,\mathsf{F}$.

We then apply the induction hypothesis on $\mathcal{E}_2$ (which has one transition less than $\mathcal{E}$, justifying the call to the induction hypothesis). If $\mathcal{T} : \mathcal{S}_2 \rightsquigarrow_{\Sigma \mathsf{e}} \mathcal{S}_2'$, then $\mathcal{T} : (\mathsf{cont}\,\mathsf{F},\ \mathcal{S}_2 \rightsquigarrow_{\Sigma \mathsf{e}} \mathsf{cont}\,\mathsf{F},\ \mathcal{S}_2')$ by the frame property (Conjecture 5), and we are done.

Otherwise, we have $\mathcal{S}_2 = \mathsf{retn}\,\mathsf{V}$, $\mathcal{D}_v : \mathsf{value}\,\mathsf{V}$, and $\mathcal{D}_t : \mathsf{of}\,\mathsf{V}\,\mathsf{T}'$. We proceed by case analysis on the proof term $\mathcal{D}_f$ establishing that the frame F is well typed.

Subcase: $\mathcal{D}_f = \mathsf{off}/\mathsf{app}_1\,\mathcal{D}_{t2}$, so $\mathsf{F} = \mathsf{app}_1\,\mathsf{E}_2$.
$\mathsf{e}/\mathsf{app}_1\ :\ (\mathsf{cont}(\mathsf{app}_1\,\mathsf{E}_2),\ \mathsf{retn}\,\mathsf{V}) \rightsquigarrow_{\Sigma \mathsf{e}} (\mathsf{cont}(\mathsf{app}_2\,\mathsf{V}),\ \mathsf{eval}\,\mathsf{E}_2)$.

Subcase: $\mathcal{D}_f = \mathsf{off}/\mathsf{app}_2\,\mathcal{D}_{v1}\,\mathcal{D}_{t1}$, so $\mathsf{F} = \mathsf{app}_2\,\mathsf{V}_1$,
$\mathcal{D}_{v1} : \mathsf{value}\,\mathsf{V}_1$, and $\mathcal{D}_{t1} : \mathsf{of}\,\mathsf{E}_1\,(\mathsf{arrow}\,\mathsf{T}'\,\mathsf{T})$.
By the canonical forms lemma on $\mathcal{D}_{v1}$ and $\mathcal{D}_{t1}$, $\mathsf{V}_1 = \mathsf{lam}\,\lambda \mathsf{x}.\,\mathsf{E}_0\,\mathsf{x}$.
$\mathsf{e}/\mathsf{app}_2\ :\ (\mathsf{cont}(\mathsf{app}_2(\mathsf{lam}\,\lambda \mathsf{x}.\,\mathsf{E}_0\,\mathsf{x})),\ \mathsf{retn}\,\mathsf{V}) \rightsquigarrow_{\Sigma \mathsf{e}} \mathsf{eval}(\mathsf{E}_0\,\mathsf{V})$.

Subcase: $\mathcal{D}_f = \mathsf{off}/\mathsf{plus}_1\,\mathcal{D}_{t2}$, so $\mathsf{F} = \mathsf{plus}_1\,\mathsf{E}_2$.
$\mathsf{e}/\mathsf{plus}_1\ :\ (\mathsf{cont}(\mathsf{plus}_1\,\mathsf{E}_2),\ \mathsf{retn}\,\mathsf{V})) \rightsquigarrow_{\Sigma \mathsf{e}} (\mathsf{cont}(\mathsf{plus}_2\,\mathsf{V}),\ \mathsf{eval}\,\mathsf{E}_2)$.

Subcase: $\mathcal{D}_f = \mathsf{off}/\mathsf{plus}_2\,\mathcal{D}_{v1}\,\mathcal{D}_{t1}$, so $\mathsf{F} = \mathsf{plus}_2\,\mathsf{V}_1$ and $\mathsf{T} = \mathsf{T}' = \mathsf{tnat}$
$\mathcal{D}_{v1} : \mathsf{value}\,\mathsf{E}_1$, and $\mathcal{D}_{t1} : \mathsf{of}\,\mathsf{E}_1\,\mathsf{tnat}$.
By the canonical forms lemma on $\mathcal{D}_{v1}$ and $\mathcal{D}_{t1}$, $\mathsf{V}_1 = \mathsf{n}\,\mathsf{N}_1$.
By the canonical forms lemma on $\mathcal{D}_v$ and $\mathcal{D}_t$, $\mathsf{V} = \mathsf{n}\,\mathsf{N}_2$.
By the effectiveness of addition on N_1 and N_2, there exists a natural number N_3 and a proof term $\mathcal{D}_a : \mathsf{add}\,\mathsf{N}_1\,\mathsf{N}_2\,\mathsf{N}_3$.
$\mathsf{e}/\mathsf{plus}_2(\mathcal{D}_a)\ :\ (\mathsf{cont}(\mathsf{app}_2(\mathsf{n}\,\mathsf{N}_1)),\ \mathsf{retn}(\mathsf{n}\,\mathsf{N}_2)) \rightsquigarrow_{\Sigma \mathsf{e}} \mathsf{retn}(\mathsf{n}\,\mathsf{N}_3)$.

This completes the proof; we have assumed standard canonical forms lemmas and the effectiveness of addition, provable by induction on the structure of N_1. □

5.3 Parallel evaluation

One way in which parallel evaluation has been incorporated into functional programming languages is by allowing pairs (or, more generally, tuples) to evaluate in parallel [FRRS08]. In this section we will consider $\mathcal{L}_2$, the first modular extension to the language $\mathcal{L}_1$ with parallel pairs. The syntax and typing rules pairs are a straightforward and standard addition:

$\mathsf{pair} : \mathsf{exp} \to \mathsf{exp} \to \mathsf{exp}$.
$\mathsf{fst} : \mathsf{exp} \to \mathsf{exp}$.
$\mathsf{snd} : \mathsf{exp} \to \mathsf{exp}$.

$\mathsf{v/pair} : \mathsf{value}\ \mathsf{E}_1 \to \mathsf{value}\ \mathsf{E}_2 \to \mathsf{value}\ (\mathsf{pair}\ \mathsf{E}_1\ \mathsf{E}_2)$.

$\mathsf{pairtp} : \mathsf{tp} \to \mathsf{tp} \to \mathsf{tp}$.

$\mathsf{of/pair} : \mathsf{of}\ \mathsf{E}_1\ \mathsf{T}_1 \to \mathsf{of}\ \mathsf{E}_2\ \mathsf{T}_2 \to \mathsf{of}\ (\mathsf{pair}\ \mathsf{E}_1\ \mathsf{E}_2)\ (\mathsf{pairtp}\ \mathsf{T}_1\ \mathsf{T}_2)$.

$\mathsf{of/fst} : \mathsf{of}\ \mathsf{E}\ (\mathsf{pairtp}\ \mathsf{T}_1\ \mathsf{T}_2) \to \mathsf{of}\ (\mathsf{fst}\ \mathsf{E})\ \mathsf{T}_1$.

$\mathsf{of/snd} : \mathsf{of}\ \mathsf{E}\ (\mathsf{pairtp}\ \mathsf{T}_1\ \mathsf{T}_2) \to \mathsf{of}\ (\mathsf{snd}\ \mathsf{E})\ \mathsf{T}_2$.

5.3.1 Dynamic semantics

Nothing in the previous signature indicates that pairs are any more interesting than the language features we have already presented. It is in the specification of the dynamic semantics that we add a new capability to the language: *parallel evaluation*. Our dynamic semantics will evaluate both parts of a pair in parallel. First, we add a new type of latent ordered proposition, $(\mathsf{cont}_2\,\mathsf{F})$, representing a frame waiting on *two* values to be returned to it.

cont_2 : frame $\to$ type_o.

fst_0 : frame.
snd_0 : frame.
pair_0 : frame.

e/fst : eval(fst E) $\twoheadrightarrow$ cont fst_0 · eval E

e/fst_0 : cont fst_0 · retn(pair V_1 V_2) $\twoheadrightarrow$ retn V_1

e/snd : eval(snd E) $\twoheadrightarrow$ cont snd_0 · eval E

e/snd_0 : cont snd_0 · retn(pair V_1 V_2) $\twoheadrightarrow$ retn V_2

e/pair : eval(pair E_1 E_2) $\twoheadrightarrow$ cont_2 pair_0 · eval E_1 · eval E_2.

e/pair_0 : cont_2 pair_0 · retn V_1 · retn V_2 $\twoheadrightarrow$ retn(pair V_1 V_2).

Our specification no longer corresponds to an on-paper description of a stack machine with one expression evaluating on top of the stack: with this change, we have made the substructural context a treelike structure where multiple independent groups of propositions may be able to transition at any given time.

5.3.2 Static semantics

Beyond the typing rules for pairs given at the beginning of this section, we need a new typing judgment for frames waiting on two values:

off2 : frame $\to$ tp $\to$ tp $\to$ tp $\to$ type.

The fst_0 and snd_0 frames lead to new cases for the regular frame typing rule, and we give a parallel frame typing rule for the pair_0 frame:

off/fst_0 : off fst_0 (pairtp T_1 T_2) T_1.

off/snd_0 : off fst_0 (pairtp T_1 T_2) T_2.

off2/pair_0 : off2 pair_0 T_1 T_2 (pairtp T_1 T_2).

And a rule in the static semantics explaining how parallel frames interact with (abs T):

t/cont_2 : cont_2 F · abs T_1 · abs T_2 · !off2 F T_1 T_2 T'
$\twoheadrightarrow$ abs T'.

Those three declarations entirely describe the static semantics of parallel evaluation. The typing rule off2/pair_0 and the rewriting rule t/cont_2 could easily be merged, but by separating them it is possible to incorporate additional parallel features into the language in a modular way.

5.3.3 Safety

We have just seen that it is possible to modularly extend both the static and dynamic semantics of $\mathcal{L}_1$ to obtain the language $\mathcal{L}_2$ with parallel evaluation of pairs. We are also able to straightforwardly extend the safety proof of $\mathcal{L}_1$ to incorporate parallel evaluation. We have to do the following things to extend the existing proof:

- Add a new consumption lemma:
  If $\mathcal{E} : \mathcal{S}[\mathsf{cont}_2\,\mathsf{F}, \mathsf{abs}\,\mathsf{T}_1, \mathsf{abs}\,\mathsf{T}_2] \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\,\mathsf{T}_\mathsf{f}$,
  then $\mathcal{E} \approx (\mathsf{t}/\mathsf{cont}(\mathcal{D}_t);\ \mathcal{E}')$, *where* $\mathcal{D}_f$: $\mathsf{off2}\,\mathsf{F}\,\mathsf{T}_1\,\mathsf{T}_2\,\mathsf{T}'$ *and* $\mathcal{E}' : \mathcal{S}[\mathsf{abs}\,\mathsf{T}'] \rightsquigarrow^*_{\Sigma\mathsf{t}} \mathsf{abs}\,\mathsf{T}_\mathsf{f}$ *for some* T'.

- Add a case for $\mathsf{t/cont_2}$ to the proof of each of the other consumption lemmas.
- Add cases for $\mathsf{e/pair}$ and $\mathsf{e/pair_0}$, $\mathsf{e/fst}$, etc. to the proof of preservation lemma.
- Add new subcases for $\mathsf{E = pair\,E_1\,E_2}$, $\mathsf{E = fst\,E}$, and $\mathsf{E = snd\,E}$ to the second case of the progress lemma where it is the case that $\mathcal{S} = \mathsf{eval\,E}$.
- Add new subcases for $\mathcal{D}_f = \mathsf{off/fst_0}$ and $\mathcal{D}_f = \mathsf{off/snd_0}$ to the third case of the progress lemma, both of which will appeal to a canonical forms lemma.
- Add a case for $\mathcal{T} = \mathsf{t/cont_2}(\mathcal{D}_f)$ to the proof of the progress lemma.

The essential structure of the safety proof is preserved under the extension of the language with parallel pairs. This is significant because, with the exception of the proof of the consumption lemma, the safety proof for $\mathcal{L}_1$ did not need to explicitly prepare for the possibility of parallel evaluation.

5.4 Exceptions

Exceptions and exception handling are another example of a feature that we can add to $\mathcal{L}_1$ or $\mathcal{L}_2$ in a modular fashion, though there is an important caveat. While we can add parallel evaluation *or* exceptions to $\mathcal{L}_1$ without reconsidering any aspects of the original specification, we cannot extend the base language with *both* exceptions and parallelism without considering their interaction. This should not be seen as a troublesome lack of modularity, however — rather, the SSOS specification in this section illustrates that exceptions and parallel evaluation are not truly orthogonal language features. Seen in this light, it is natural that we must clearly describe how the features interact.

The syntax of exceptions includes error, which raises an exception, and trycatch, which includes a primary expression E and a secondary expression E′ that is evaluated only if the evaluation of E returns an error. It would not be much more difficult to allow exceptions to carry a value, but we do not do so here.

error : exp.
trycatch : exp → exp → exp.

of/error : of error T.

of/trycatch : of E T → of E' T → of (trycatch E E') T.

5.4.1 Dynamic semantics

To give the dynamic semantics of exceptions and exception handling, we introduce two new ordered atomic propositions. The first, $\mathsf{handle\,E'}$, represents an exception-handling stack frame. The second, raise, represents an uncaught exception.

raise : type_o.
handle : exp → type_o.

e/raise : eval(error) ↠ raise.

e/trycatch : eval(trycatch E E') ↠ handle E' · eval E.

e/vhandle : handle E' · retn V ↠ retn V.

e/xhandle : handle E' · raise ↠ eval E'.

e/xcont : cont F · raise ↠ raise.

As mentioned before, we must deal explicitly with the interaction of parallel evaluation and exception handling by explaining how exceptions interact with cont_2 frames. The following is one possibility in which both sub-computations must terminate before an error is returned.

$$\mathsf{e/xvcont} : \mathsf{cont}_2\,\mathsf{F} \cdot \mathsf{raise} \cdot \mathsf{retn}\,\mathsf{V} \twoheadrightarrow \mathsf{raise}.$$

$$\mathsf{e/vxcont} : \mathsf{cont}_2\,\mathsf{F} \cdot \mathsf{retn}\,\mathsf{V} \cdot \mathsf{raise} \twoheadrightarrow \mathsf{raise}.$$

$$\mathsf{e/xxcont} : \mathsf{cont}_2\,\mathsf{F} \cdot \mathsf{raise} \cdot \mathsf{raise} \twoheadrightarrow \mathsf{raise}.$$

Another obvious candidate for the interaction between parallel evaluation of pairs and exceptions would be for a pair to immediately raise an exception if either of its components raises an exception. However, it is not obvious how to gracefully implement this in the ordered SSOS style we have presented so far. This is a symptom of a broader problem, namely that ordered SSOS specifications don't, in general, handle non-local transfer of control particularly well. This is a limitation of the specification style and not the framework, a point which is discussed further in Section 6.2.

5.4.2 Static semantics

The static semantics of exception handling are extremely simple; there are just two rules. An uncaught exception has any type, and an exception handler must have the same type as its sub-computation.

$$\mathsf{t/raise} : \quad \mathsf{raise} \twoheadrightarrow \mathsf{abs}\,\mathsf{T}.$$

$$\mathsf{t/handle} : \mathsf{handle}\,\mathsf{E} \cdot \mathsf{abs}\,\mathsf{T} \cdot \mathsf{!of}\,\mathsf{E}\,\mathsf{T} \twoheadrightarrow \mathsf{abs}\,\mathsf{T}.$$

5.4.3 Safety

The existing structure of the safety proof also extends straightforwardly to handle the addition of exceptions to the language; the most significant change is to the definition of safety itself, as a safe state either steps, is a returned value $\mathsf{retn}\,\mathsf{V}$, or is an unhandled exception raise.

5.5 Discussion: modular proofs and their verification

So far in this section we have explored the basics of substructural operational semantics, presented a strategy for giving static semantics to these languages, and shown how invariants established by the static semantics can be used in proofs of language safety. Not only is it possible to straightforwardly extend the dynamic and static semantics by inserting more rules, it is also possible to straightforwardly extend safety proofs by adding more cases. For this reason, I hope to be able to show in my thesis work that SSOS specification supports not only modular specification of a programming language's semantics but also a modular specification of a programming language's safety theorem.

There is, however, still work to be done before this argument can be made with complete confidence. Both of the extensions in this section can be generally described as adding control features to the language — language extensions that do not add new control features (such as inductive types, sums, fixed points, etc...) should in general be even easier to add. The other stated goal of SSOS specification is to allow modular specification of stateful features. The specification of mutable state, presented in Appendix A.1, extends the dynamic and static semantics presented so far in a modular way, but the safety proof presented in this section needs to be modified to prove safety for the extended semantics.

Another challenge is the consumption lemma. Every time we added a new relevant feature to the language, we needed to add both a new consumption lemma and a new case to all the existing consumption lemmas. On paper, this kind of behavior is easy to sweep under the rug with language

such as "all the other cases are similar." When considering mechanically-verified proofs, however, this quadratic "proof complexity" can be an obstacle to the formal verification and modular extension of proofs. The obvious solution is to provide tactics that can automatically verify proofs of theorems like the consumption lemma with a constant or linear amount of work by the human author.[12] Insofar as it is possible, routine proofs with super-linear proof complexity should be left to computers.

The above considerations have to do with constructing proofs. The most significant obstacle to mechanically verifying the proofs of the safety properties in this section is probably exemplified by the following reasoning, extracted from one of the cases of the consumption lemma:

> *Given:*
> $\mathcal{S}[\mathsf{cont}\,\mathsf{F}, \mathsf{abs}\,\mathsf{T}] \approx \mathcal{S}'[\mathsf{cont}\,\mathsf{F}_1, \mathsf{abs}\,\mathsf{T}_1]$*, there are two possibilities.*
> *The first possibility is that* $\mathcal{S}[\mathsf{abs}\,\mathsf{T}'] \approx \mathcal{S}'[\mathsf{abs}\,\mathsf{T}']$, $\mathsf{F} = \mathsf{F}_1$, *and* $\mathsf{T} = \mathsf{T}_1$.
> *The other possibility is that, for some* $\mathcal{S}''$,
> $\mathcal{S}[\mathsf{cont}\,\mathsf{F}, \mathsf{abs}\,\mathsf{T}] \approx \mathcal{S}'[\mathsf{cont}\,\mathsf{F}_1, \mathsf{abs}\,\mathsf{T}_1] \approx \mathcal{S}''[\mathsf{cont}\,\mathsf{F}, \mathsf{abs}\,\mathsf{T}][\mathsf{cont}\,\mathsf{F}_1, \mathsf{abs}\,\mathsf{T}_1]$.

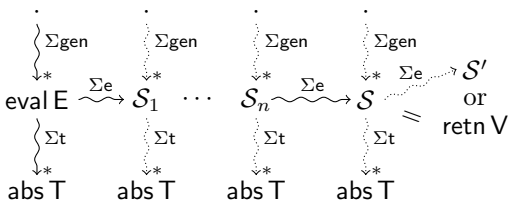
Implicit in this reasoning is a kind of *exhaustiveness* or *coverage* checking: the two cases that we claim capture all possibilities do, in fact, capture all possibilities. Similar issues arose in the mechanical verification of HLF specifications, but the solution was somewhat ad-hoc [Ree09a]. One of the primary goals of my proposed work is to develop general solutions to coverage checking problems that involve algebraic properties of expressions and ephemeral contexts, but I expect this to be a significant challenge.

5.6 Regular worlds and context generators

In the discussion of safety in Section 5.2, we needed a special "context invariant" lemma saying that the dynamic semantics preserves the property that a substructural context contains no ($\mathsf{abs}\,\mathsf{T}$) propositions. Another way of specifying this context invariant is in terms of a transition system that *generates* the context. The degenerate "rewriting rules" that capture the context invariant for parallel evaluation are as follows:

$\mathsf{gen/eval}$: $\forall \mathsf{E}.\ \mathsf{eval}\,\mathsf{E}.$
$\mathsf{gen/retn}$: $\forall \mathsf{E}.\ \mathsf{retn}\,\mathsf{E}.$
$\mathsf{gen/cont}$: $\forall \mathsf{F}.\ \mathsf{cont}\,\mathsf{F}.$
$\mathsf{gen/cont}_2$: $\forall \mathsf{F}.\ \mathsf{cont}_2\,\mathsf{F}.$

This changes the picture of safety presented in Section 5.2 to the following picture, which shows the two invariants given by the context generating rules and the static semantics. The context generating rules construct the context, and the static semantics then analyze it.



[12] An obvious analogy is cut elimination. Coverage checking a cut elimination argument for a logic with n inference rules generally requires a proof assistant to consider $O(n^2)$ cases, but an assistant like Twelf or Agda can verify the theorem given at most three cases for each inference rule (a principal case, a left commutative case, and a right commutative case).

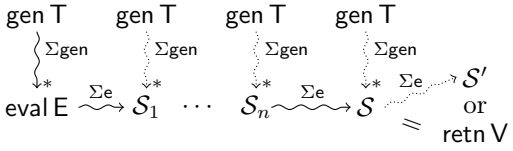
This account of context generating rules bears a strong resemblance to the *regular worlds* specifications mentioned in Section 3 and implemented in the LF/Twelf metalogical framework [PS99]; I am interested in exploring the use of transition rules as a logical justification for regular worlds specifications. In this setting, however, there is no reason why we need to limit ourselves to regular worlds-like context generators. Another possibility is the use of context generators that capture the tree-like structure of the ordered context:

$$\begin{array}{ll}
\mathsf{gen/eval}: & \forall \mathsf{E}.\ \mathsf{gen} \twoheadrightarrow \mathsf{eval}\ \mathsf{E}. \\
\mathsf{gen/retn}: & \forall \mathsf{E}.\ \mathsf{gen} \twoheadrightarrow \mathsf{retn}\ \mathsf{E}. \\
\mathsf{gen/cont}: & \forall \mathsf{F}.\ \mathsf{gen} \twoheadrightarrow \mathsf{cont}\ \mathsf{F} \cdot \mathsf{gen}. \\
\mathsf{gen/cont}_2: & \forall \mathsf{F}.\ \mathsf{gen} \twoheadrightarrow \mathsf{cont}_2\ \mathsf{F} \cdot \mathsf{gen} \cdot \mathsf{gen}.
\end{array}$$

But what if we don't stop there? We could also have the gen atomic proposition carry a type, and then only allow the generation of well-typed tree structures (we return to leaving quantification implicit):

$$\begin{array}{ll}
\mathsf{gen/eval}: & \mathsf{gen}\ \mathsf{T} \cdot !\mathsf{of}\ \mathsf{E}\ \mathsf{T} \twoheadrightarrow \mathsf{eval}\ \mathsf{E}. \\
\mathsf{gen/retn}: & \mathsf{gen}\ \mathsf{T} \cdot !\mathsf{value}\ \mathsf{E} \cdot !\mathsf{of}\ \mathsf{E}\ \mathsf{T} \twoheadrightarrow \mathsf{retn}\ \mathsf{E}. \\
\mathsf{gen/cont}: & \mathsf{gen}\ \mathsf{T} \cdot !\mathsf{off}\ \mathsf{F}\ \mathsf{T'}\ \mathsf{T} \twoheadrightarrow \mathsf{cont}\ \mathsf{F} \cdot \mathsf{gen}\ \mathsf{T'}. \\
\mathsf{gen/cont}_2: & \mathsf{gen}\ \mathsf{T} \cdot !\mathsf{off}_2\ \mathsf{F}\ \mathsf{T}_1\ \mathsf{T}_2\ \mathsf{T} \twoheadrightarrow \mathsf{cont}_2\ \mathsf{F} \cdot \mathsf{gen}\ \mathsf{T}_1 \cdot \mathsf{gen}\ \mathsf{T}_2.
\end{array}$$

The static semantics of the language now have been captured entirely within the generation rules, seemingly removing any need for a second invariant that analyzes the state. The statement of the safety theorem would remain as it was in Theorem 1, but its proof would look like this:



The perspective suggested by this picture is quite different from our usual intuitions about static semantics. Usually, static semantics are seen as a way of *analyzing* or *abstractly evaluating* a state in order to generate an approximation (the type) that is invariant under evaluation and sufficient to ensure safety; here, the static semantics is more of a template for generating safe states. We have not explored this style of analysis in depth, but it does seem to address certain significant complications that have been encountered in the process of proving safety for the linear-destination-passing-style specifications introduced in Section 6.2.

5.7 Beyond language safety

The discussion in this section has centered on specifying substructural operational semantics in ordered logic and reasoning about type language safety. Type safety is an interesting and critical property, and it is one that seems to have been overlooked by many related approaches to modular specification of stateful and concurrent programming languages. However, type safety is just one of many interesting properties of such programming languages!

I also plan to explore the use of other techniques for reasoning about concurrent systems, particularly those that can be used to statically preclude race conditions and deadlocks (perhaps along the lines of Abadi et. al [AFF06]). Methods I plan to consider formalizing include simulation/bisimulation-based reasoning, session types, fractional permissions, and information flow type systems.

6 Transformations of SSOS specifications

The logical basis of our framework gives us a number of potentially powerful tools for transforming specifications in ways that either preserve or approximate the meaning of the original specification; in this section, I discuss a variety of interesting transformations on SSOS specifications. This section is a change of pace from the previous sections, which were primarily motivated by the problem of giving modular SSOS specifications and proving them type safe. That said, the investigations in this section represent other strategies for reasoning about SSOS specifications.

6.1 Transformations for approximation

In the paper "Linear logical algorithms," coauthored with Frank Pfenning, we observed that program analyses such as control flow analysis and alias analysis could be derived directly from SSOS specifications in linear logic [SP09]. In a submitted journal paper, this approach was extended to SSOS specifications in ordered logic [SP10]. I will give a very brief overview of this approach here.

A control flow analysis can be derived from the functions-only fragment of $\mathcal{L}_1$ by taking the four rules describing function application and passing them through an automatic, meaning-preserving transformation. The ordering constraints on the context are represented by adding two arguments to every atomic proposition: these extra arguments chain together an ordered sequence of linear propositions in the manner of a doubly-linked list.

$$\begin{aligned}
&\mathsf{el/lam}: && \mathsf{eval}\ (\mathsf{lam}\ \lambda x.\ \mathsf{E}\ x)\ \mathsf{D}\ \mathsf{D'} \multimap \mathsf{retn}\,(\mathsf{lam}\ \lambda x.\ \mathsf{E}\ x)\ \mathsf{D}\ \mathsf{D'}.\\
&\mathsf{el/app}: && \mathsf{eval}\ (\mathsf{app}\ \mathsf{E}_1\ \mathsf{E}_2)\ \mathsf{D}\ \mathsf{D'}\\
&&& \quad \multimap \exists \mathsf{d}_1.\ \mathsf{cont}\ (\mathsf{app}_1\ \mathsf{E}_2)\ \mathsf{D}\ \mathsf{d}_1 \otimes \mathsf{eval}\ \mathsf{E}_1\ \mathsf{d}_1\ \mathsf{D'}.\\
&\mathsf{el/app}_1: && \mathsf{cont}\ (\mathsf{app}_1\ \mathsf{E}_2)\ \mathsf{D}\ \mathsf{D}_1 \otimes \mathsf{retn}\ \mathsf{V}_1\ \mathsf{D}_1\ \mathsf{D'}\\
&&& \quad \multimap \exists \mathsf{d}_2.\ \mathsf{cont}\ (\mathsf{app}_2\ \mathsf{V}_1)\ \mathsf{D}\ \mathsf{d}_2 \otimes \mathsf{eval}\ \mathsf{E}_2\ \mathsf{d}_2\ \mathsf{D'}.\\
&\mathsf{el/app}_2: && \mathsf{cont}\ (\mathsf{app}_2\ (\mathsf{lam}\ \lambda x.\ \mathsf{E}_0\ x))\ \mathsf{D}\ \mathsf{D}_2 \otimes \mathsf{retn}\ \mathsf{V}_2\ \mathsf{D}_2\ \mathsf{D'}\\
&&& \quad \multimap \mathsf{eval}\ (\mathsf{E}_0\ \mathsf{V}_2)\ \mathsf{D}\ \mathsf{D'}.
\end{aligned}$$

The existential quantifier in the conclusion of a rule creates new parameters (called *destinations*) that maintain the ordering invariant that was previously maintained by the structure of the ordered context. When SSOS specifications are passed through this transformation, the result is similar to the *linear destination-passing style* that was used in original SSOS specifications. The only essential difference is that the transformation adds a somewhat useless second argument to eval and retn propositions; these vestigial destinations are grayed-out in the specification above. This first transformation into linear logic does not change provability or the structure of proofs, and can therefore be seen as a way of representing forward-chaining ordered logical specifications in a linear logical framework like CLF.

Destinations, and parameters more generally, are quite useful in SSOS specifications — while the examples in Section 5 did not use them, they are used in many of the examples in the appendix. For instance, destinations can be used to represent an environment semantics where a parameter is substituted into an expression rather than substituting the value itself; a new persistent proposition is then introduced which permanently associates the fresh parameter with the value. The specification above can be modified to demonstrate this; the modified rule $\mathsf{el/app}_2$ and the new rule el/bind are the only major changes, though the vestigial D' argument was removed throughout:

$$\begin{aligned}
&\mathsf{el/lam}: && \mathsf{eval}\ (\mathsf{lam}\ \lambda x.\ \mathsf{E}\ x)\ \mathsf{D} \multimap \mathsf{retn}\,(\mathsf{lam}\ \lambda x.\ \mathsf{E}\ x)\ \mathsf{D}.\\
&\mathsf{el/app}: && \mathsf{eval}\ (\mathsf{app}\ \mathsf{E}_1\ \mathsf{E}_2)\ \mathsf{D}\\
&&& \quad \multimap \exists \mathsf{d}_1.\ \mathsf{cont}\ (\mathsf{app}_1\ \mathsf{E}_2)\ \mathsf{D}\ \mathsf{d}_1 \otimes \mathsf{eval}\ \mathsf{E}_1\ \mathsf{d}_1.
\end{aligned}$$

$$\mathsf{el/app_1}: \ \mathsf{cont}\ (\mathsf{app}_1\ \mathsf{E}_2)\ \mathsf{D}\ \mathsf{D}_1 \otimes \mathsf{retn}\ \mathsf{V}_1\ \mathsf{D}_1 \multimap \exists \mathsf{d}_2.\ \mathsf{cont}\ (\mathsf{app}_2\ \mathsf{V}_1)\ \mathsf{D}\ \mathsf{d}_2 \otimes \mathsf{eval}\ \mathsf{E}_2\ \mathsf{d}_2.$$

$$\mathsf{el/app_2}: \ \mathsf{cont}\ (\mathsf{app}_2\ (\mathsf{lam}\ \lambda \mathsf{x}.\ \mathsf{E}_0\ \mathsf{x}))\ \mathsf{D}\ \mathsf{D}_2 \otimes \mathsf{retn}\ \mathsf{V}_2\ \mathsf{D}_2 \multimap \exists \mathsf{x}.\ \mathsf{eval}\ (\mathsf{E}_0\ (\mathsf{var}\ \mathsf{x}))\ \mathsf{D} \otimes !\mathsf{bind}\ \mathsf{x}\ \mathsf{V}_2.$$

$$\mathsf{el/bind}: \ \mathsf{eval}\ (\mathsf{var}\ \mathsf{X})\ \mathsf{D} \otimes !\mathsf{bind}\ \mathsf{X}\ \mathsf{V} \multimap \mathsf{retn}\ \mathsf{V}\ \mathsf{D}.$$

Once we have obtained a linear logic specification, we are in a setting where program approximation can be considered. By making all the linear propositions *persistent*, a sort of collecting semantics arises. If we then use equality constraints to unify introduced parameters and existing terms, it is possible to get a logical specification that, when run as a saturating, forward-chaining logic program, terminates and can be interpreted as a control flow analysis. The following rules, for instance, describe a basic control flow analysis for the lambda calculus when run as a saturating logic program. In particular, if the function $\mathsf{lam}\,\lambda \mathsf{x}.\,\mathsf{E}\,\mathsf{x}$ may be called at runtime from the call site represented by the expression E_1 in the actual semantics, then the atomic $\mathsf{retn}\,(\mathsf{lam}\,\lambda \mathsf{x}.\,\mathsf{E}\,\mathsf{x})\,\mathsf{E}_1$ will appear in the saturated database produced by the control flow analysis.

$$\mathsf{ea/lam}: \ \mathsf{eval}\ (\mathsf{lam}\ \lambda \mathsf{x}.\ \mathsf{E}\ \mathsf{x})\ \mathsf{D} \rightarrow \mathsf{retn}\,(\mathsf{lam}\ \lambda \mathsf{x}.\ \mathsf{E}\ \mathsf{x})\ \mathsf{D}\ .$$

$$\mathsf{ea/app}: \ \mathsf{eval}\ (\mathsf{app}\ \mathsf{E}_1\ \mathsf{E}_2)\ \mathsf{D} \rightarrow \exists \mathsf{d}_1.\ \mathsf{d}_1 = \mathsf{E}_1,\ \mathsf{cont}\ (\mathsf{app}_1\ \mathsf{E}_2)\ \mathsf{D}\ \mathsf{d}_1,\ \mathsf{eval}\ \mathsf{E}_1\ \mathsf{d}_1.$$

$$\mathsf{ea/app_1}: \ \mathsf{cont}\ (\mathsf{app}_1\ \mathsf{E}_2)\ \mathsf{D}\ \mathsf{D}_1,\ \mathsf{retn}\ \mathsf{V}_1\ \mathsf{D}_1 \rightarrow \exists \mathsf{d}_2.\ \mathsf{d}_2 = \mathsf{E}_2,\ \mathsf{cont}\ (\mathsf{app}_2\ \mathsf{V}_1)\ \mathsf{D}\ \mathsf{D}_2,\ \mathsf{eval}\ \mathsf{E}_2\ \mathsf{d}_2.$$

$$\mathsf{ea/app_2}: \ \mathsf{cont}\ (\mathsf{app}_2\ (\mathsf{lam}\ \lambda \mathsf{x}.\ \mathsf{E}_0\ \mathsf{x}))\ \mathsf{D}\ \mathsf{D}_2,\ \mathsf{retn}\ \mathsf{V}_2\ \mathsf{D}_2 \rightarrow \exists \mathsf{x}.\ \mathsf{x} = (\mathsf{lam}\ \lambda \mathsf{x}.\ \mathsf{E}_0\ \mathsf{x}),\ \mathsf{eval}\ (\mathsf{E}_0\ (\mathsf{var}\ \mathsf{x}))\ \mathsf{D},\ \mathsf{bind}\ \mathsf{x}\ \mathsf{V}_2.$$

$$\mathsf{ea/bind}: \ \mathsf{eval}\ (\mathsf{var}\ \mathsf{X})\ \mathsf{D},\ \mathsf{bind}\ \mathsf{X}\ \mathsf{V}_2 \rightarrow \mathsf{retn}\ \mathsf{E}_2\ \mathsf{D}.$$

This is an extremely brief overview; more detail can be found in [SP09, SP10].

6.2 Transformations for modularity

There is one known and significant failing of SSOS specifications in ordered logic: they are unable to capture the expected semantics of first-class continuations, because the context representing the entire control stack cannot necessarily be reified as a term. There is a way to give an SSOS specification of first-class continuations in linear logic, however! If stack frames are all persistent, then a destination can be used to uniquely identify a continuation. A value can then be "thrown" to a different continuation simply by returning the value to a different destination, as shown in the $\mathsf{el/throw_2}$ rule below:

$$\mathsf{el/letcc}: \ \mathsf{eval}\ (\mathsf{letcc}\ \lambda \mathsf{x}.\ \mathsf{E}\ \mathsf{x})\ \mathsf{D} \multimap \mathsf{eval}\ (\mathsf{E}\ (\mathsf{continue}\ \mathsf{D}))\ \mathsf{D}.$$

$$\mathsf{el/continue}: \ \mathsf{eval}\ (\mathsf{continue}\ \mathsf{D}_{\mathsf{cont}})\ \mathsf{D} \multimap \mathsf{retn}\ (\mathsf{continue}\ \mathsf{D}_{\mathsf{cont}})\ \mathsf{D}.$$

$$\mathsf{el/throw}: \ \mathsf{eval}\ (\mathsf{throw}\ \mathsf{E}_1\ \mathsf{E}_2)\ \mathsf{D} \multimap \exists \mathsf{d}_1.\ !\mathsf{cont}\ (\mathsf{throw}_1\ \mathsf{E}_2)\ \mathsf{D}\ \mathsf{d}_1 \otimes \mathsf{eval}\ \mathsf{E}_1\ \mathsf{d}_1.$$

$$\mathsf{el/throw_1}: \ !\mathsf{cont}\ (\mathsf{throw}_1\ \mathsf{E}_2)\ \mathsf{D}\ \mathsf{D}_1 \otimes \mathsf{retn}\ \mathsf{V}_1\ \mathsf{D}_1 \multimap \exists \mathsf{d}_2.\ !\mathsf{cont}\ (\mathsf{throw}_2\ \mathsf{V}_1)\ \mathsf{D}\ \mathsf{d}_2 \otimes \mathsf{eval}\ \mathsf{E}_2\ \mathsf{d}_2.$$

$$\mathsf{el/throw_2}: \ !\mathsf{cont}\ (\mathsf{throw}_2\ (\mathsf{continue}\ \mathsf{D}_{\mathsf{cont}}))\ \mathsf{D}\ \mathsf{D}_2 \otimes \mathsf{retn}\ \mathsf{V}_2\ \mathsf{D}_2 \multimap \mathsf{retn}\ \mathsf{V}_2\ \mathsf{D}_{\mathsf{cont}}.$$

The original motivation for introducing ordered logic into SSOS specifications was to represent control stacks without the bureaucracy of destinations, but the trade-off is that this style of specification cannot easily capture first-class continuations and other kinds of non-local transfer of control. The only real hope of modularly composing ordered SSOS specifications and specifications of first-class continuations may be to extend the meaning-preserving transformations described above and in [SP10] to allow ordered SSOS specifications to be simply and gracefully transformed into a form that is compatible with linear-destination-passing style SSOS specifications. Even if this is possible, it may well be too challenging or unpleasant to allow proofs of language safety to be compatible across these transformations, meaning that the modularity of proofs may be lost even if the modularity of specifications can be restored.

6.3 Transformation for implementation

Logical transformations can also be used to give alternate logic programming interpretations to SSOS specifications. The transformation of ordered SSOS specifications to linear SSOS specifications described in Section 6.1, for instance, outputs specifications that can be run by logic programming languages based on CLF like Lollimon [LPPW05] and Celf [SNS08].

Additionally, the output of the translation into linear logic can be further transformed into an LLF specification. Any linear transition rule $A^+ \multimap B^+$ can be transformed into the LLF logical framework by introducing a new atomic proposition rhs and rewriting every rule as $(B^+ \multimap \mathsf{rhs}) \multimap (A^+ \multimap \mathsf{rhs})$.[13] At this point, a backward-chaining logic programming language like Lolli or HLF/Twelf could be used as an implementation of the logic programming semantics of the framework. In theory, the metatheoretic capabilities HLF/Twelf could even be used to reason directly about properties of ordered SSOS specifications that had been transformed into LLF. However, as this transformation adds complexity and destroys concurrent equivalence, this does not necessarily seem like a promising approach.

Yet another interesting use of program transformations for implementation goes back to the parsing rules for $\mathcal{L}_0$ first introduced in Section 4.7. If we take that specification, apply the transformation into linear logic, and then apply the approximation transformation into persistent logic, we get the following specification, which can be run as a saturating logic program:

$\mathsf{parsea/num} : (\mathsf{num\ N\ D_1\ D_2}) \to \mathsf{parsed\ (n\ N)\ D_1\ D_2}.$
$\mathsf{parsea/plus} : (\mathsf{parsed\ E_1\ D_1\ D_2}), (\mathsf{+\ D_2\ D_3}), (\mathsf{parsed\ E_2\ D_3\ D_4}) \to \mathsf{parsed\ (plus\ E_1\ E_2)\ D_1\ D_4}.$

The most interesting aspect of this transformed specification is that it encodes the CKY parsing algorithm when run as a saturating logic program. This means that the transformation did not produce an approximation of the original logic program in this instance, it produced a precise version of the original program that is amenable to a very different execution strategy.

6.4 Defunctionalization and refunctionalization

As previously mentioned, atomic propositions in SSOS specifications can be categorized as active (like eval), passive (like retn and raise) or latent (like cont and handle). The first SSOS specifications had only active and passive atomic propositions. To see why, we can look at an SSOS specification for parallel pairs that uses a style similar to original SSOS specifications (the difference is that the original specifications were linear, not ordered):

[13] LLF does not support positive propositions, but because both A^+ and B^+ appear on the left of an arrow, they can be mechanically Curried into acceptable LLF specifications.

$$\begin{aligned}&\mathsf{e/pair\text{-}alt} : \mathsf{eval}\ (\mathsf{pair}\ \mathsf{E}_1\ \mathsf{E}_2)\\ &\qquad\qquad \twoheadrightarrow\ (\forall \mathsf{V}_1,\, \mathsf{V}_2.\ \mathsf{retn}\ \mathsf{V}_1 \cdot \mathsf{retn}\ \mathsf{V}_2 \twoheadrightarrow \mathsf{retn}\ (\mathsf{pair}\ \mathsf{V}_1\ \mathsf{V}_2))\\ &\qquad\qquad\quad \cdot\ \mathsf{eval}\ \mathsf{E}_1 \cdot \mathsf{eval}\ \mathsf{E}_2.\end{aligned}$$

This specification is not supported by the logical framework we have presented because the conclusion of the rule contains a non-atomic negative proposition; however, it would not be conceptually difficult to add this to the framework. There are two reasons that we do not want to use this style of specification. First, it interferes with our ability to write a static semantics, and second, it interferes with our ability to write modular language extensions. The latter problem is easier to see: if we wanted to incorporate exceptions into the original-style SSOS specification of parallel pairs, it would have to look something like this:

$$\begin{aligned}&\mathsf{e/pair\text{-}alt\text{-}raise} :\\ &\quad \mathsf{eval}\ (\mathsf{pair}\ \mathsf{E}_1\ \mathsf{E}_2)\\ &\quad\quad \twoheadrightarrow\ ((\forall \mathsf{V}_1,\, \mathsf{V}_2.\ \mathsf{retn}\ \mathsf{V}_1 \cdot \mathsf{retn}\ \mathsf{V}_2\ \twoheadrightarrow\ \mathsf{retn}\ (\mathsf{pair}\ \mathsf{V}_1\ \mathsf{V}_2))\ \&\\ &\quad\quad\quad (\forall \mathsf{V}_1.\ \mathsf{retn}\ \mathsf{V}_1 \cdot \mathsf{raise}\ \twoheadrightarrow\ \mathsf{raise})\ \&\\ &\quad\quad\quad (\forall \mathsf{V}_2.\ \mathsf{raise} \cdot \mathsf{retn}\ \mathsf{V}_1\ \twoheadrightarrow\ \mathsf{raise})\ \&\\ &\quad\quad\quad (\mathsf{raise} \cdot \mathsf{raise}\ \twoheadrightarrow\ \mathsf{raise}))\\ &\quad\quad\quad\ \cdot\ \mathsf{eval}\ \mathsf{E}_1 \cdot \mathsf{eval}\ \mathsf{E}_2.\end{aligned}$$

This shows that the original style of SSOS specification makes modular extension much more difficult. Furthermore, no expressiveness is lost when we disallow negative atomic propositions from appearing in the conclusion of transition rules. In a the signature for $\mathcal{L}_2$, an atomic proposition $(\mathsf{cont}_2\ \mathsf{pair}_0)$ behaves in precisely the same way the negative atomic proposition $(\forall \mathsf{V}_1,\, \mathsf{V}_2.\ \mathsf{retn}\ \mathsf{V}_1 \cdot \mathsf{retn}\ \mathsf{V}_2 \twoheadrightarrow \mathsf{retn}\ (\mathsf{pair}\ \mathsf{V}_1\ \mathsf{V}_2))$ would behave. There is a separate question as to whether $\mathsf{e/pair\text{-}alt}$ is a cleaner specification than $\mathsf{e/pair}$ combined with $\mathsf{e/pair}_0$; I am not convinced that that this is the case.

There is obviously a sort of transformation that connects the two specifications, but the "defunctionalized" style first presented in [Pfe04] and developed in [PS09] and Section 5 seems to be clearly superior as a basis for modular specification of programming languages features. It will be interesting to consider if there is a formal connection between this kind of defunctionalization and the defunctionalization that appears in Danvy's rational reconstructions of abstract machines [Dan08].

7 Conclusion

In this proposal, I have presented an logical framework based on a state-transition-based view of ordered linear logic and presented preliminary results indicating that the framework allows for both modular specification of programming languages and formal reasoning about their safety properties. I have also discussed how transformations of specifications in ordered logic can be used to explore program approximations and alternative evaluation strategies. In this conclusion, I will return to a discussion of related work, give an outline of the thesis I am proposing, and discuss my plan for getting there.

7.1 Related work

We have discussed most of the relevant related work throughout this proposal, but I want to emphasize two particular strands of related work. From the logical framework perspective, the most closely related work is the LF/LLF/OLF/CLF/HLF family of logical frameworks, though the focus that the framework presented here gives to state transitions exists elsewhere only in the CLF framework, and even there state transitions are second-class citizens. The only published work on proving properties

of SSOS specifications in CLF is Schack-Nielsen's proof of the equivalence of a big-step operational semantics and an original-style SSOS specification [SN08].

The notion that ideas at the intersection of abstract machine specifications and linear logic can capture stateful and concurrent programming language features in a modular way is one with a fairly long history [Chi95, CP02, CPWW02, Pfe04, Mil09, PS09]. However, while the power of these frameworks for specification has long been recognized, formal reasoning about properties of these specifications has typically concentrated on properties like approximation [Mil08, SP09] and equivalence [SN08] rather than on familiar properties like progress and preservation that I demonstrate in the transition-based framework. I am aware of two exceptions to this pattern, though in both cases only preservation lemmas were formalized, not progress lemmas. As discussed in Section 6.3, Cervesato's LLF specification of *MLR* can be seen as a transition-based specification that has been flipped around and turned into a backward-chaining specification, and Reed's HLF is able to mechanically verify Cervesato's preservation theorem [Ree09a]. Felty and Momigliano used a similar style of specification in their work on Hybrid to encode abstract machines in ordered logic and reason about subject reduction in either Isabelle/HOL or Coq [FM08].

While the theoretical basis of this proposal is found in the study of logical frameworks, the most similar project in terms of goals and strategies is the rewriting logic semantics project [MR07], and in particular the K framework for language specifications [ŞR10, RŞ10]. Based on the Maude rewriting framework, these two projects have proven successful in specifying, model-checking, and efficiently executing operational semantics of stateful and concurrent programming languages, including a number of large formalizations of object oriented [HR07] and functional [MHR07] programming languages.

Many specifications in the K framework bear a strong resemblance to SSOS specifications, and the two approaches to language formalization seem to share a great deal of fundamental structure, even if they differ substantially in emphasis. I am only aware of one discussion of safety via progress and preservation for a K specification [EŞR08]; there was no discussion of formalizing that proof. The primary limitation of K relative to our approach is a lack of LF-like canonical forms and higher-order abstract syntax, both of which were critical to our specifications and to reasoning about their type safety. The primary limitation of our approach relative to K is the absence of any mechanism for capturing arbitrary sets of propositions at the rule level — in K there is no distinction between terms and propositions, so it is as if we could write this rule:

$$\mathsf{CONT} \cdot \mathsf{eval}(\mathsf{callcc}(\lambda \mathsf{x}.\, \mathsf{E}\, \mathsf{x})) \twoheadrightarrow \mathsf{CONT} \cdot \mathsf{eval}(\mathsf{E}(\mathsf{continue}\, \mathsf{CONT}))$$

where CONT captures *all* the propositions $\mathsf{cont}\, \mathsf{F}$ to the left of $\mathsf{eval}(\mathsf{callcc}(\lambda \mathsf{x}.\, \mathsf{E}\, \mathsf{x}))$. This makes K's approach to the modular specification of continuations infeasible in our framework. In addition, the existing interpreters for fragments of the framework described in this proposal (i.e. Ollibot, Lollimon, and Celf) are nowhere near competitive with the performance of the fine-tuned and extensively engineered Maude interpreter for rewriting logic.

7.2 Outline

While the ultimate structure of my thesis will obviously depend significantly on the outcome of the research I am proposing to do, I would currently predict that my dissertation will have the following outline:

A transition-based view of logic. I will start by presenting all of ordered linear logic (save for, probably, additive disjunction) based around the organizing principle, presented in Section 3.2 of this proposal, that logic can be treated as a state transition system. I will discuss notions of local soundness and completeness for this view of logic, give a focused presentation of the logic, and prove the completeness of the focused presentation.

A logical framework for evolving systems. By adding proof terms to a focused fragment of logic presented in the preceding section, I will present a logical framework along the lines of the framework in Section 4.

Substructural operational semantics. An extension of Section 5 in the proposal. Following [PS09], I will present SSOS specifications of a number of language features — at minimum, I expect to present parallel evaluation, mutable storage, and exceptions in addition to presenting function evaluation with a substitution semantics, an environment semantics, and a call-by-need semantics. In addition, this section will present a static semantics for each of these specifications and a proof of language safety via progress and preservation proofs.

Implementing and reasoning about SSOS specifications. This section will consider three levels of implementation. First, an implementation of the framework, which involves (mostly well-understood) issues such as reconstructing implicitly quantified parameters. Second, a logic programming language based on the logical framework that gives a backwards-chaining interpretation to the canonical forms fragment and a forward-chaining interpretation to the transition fragment. Third, a constructive framework for formally reasoning about specifications.

Transforming SSOS specifications An extension of Section 6 in this proposal. By detailing a transformation on SSOS specifications that naturally gives rise to linear destination-passing style, I present the SSOS specification and type safety proofs of a language with first-class continuations. I will also discuss how transformations on SSOS specifications can be used to derive program approximations (as presented in [SP10]). Ideally, the tools described in the previous section should provide a degree of automated support for applying these transformations.

Another kind of program transformation is a compiler transformation; in this section, I will consider how the logical framework can specify and relate different internal languages within a compiler.

Reasoning beyond safety. The SSOS specifications considered up to this point in the thesis will all have a notion of safety that does not allow deadlocks — a well-typed machine state must either step or be a returned value. In this section, I will consider the specification of systems that can communicate and potentially deadlock, and will consider the application of coinductive notions such as bisimulation that are used to reason about such communicating processes; this will also be where I discuss other techniques for reasoning about properties of SSOS specifications along the lines of those discussed in Section 5.7.

7.3 Goals and plan

I plan to concentrate first on the development of the logical framework and its properties while also writing on-paper specifications and safety proofs for a variety of programming language features. There are quite a number of potential framework designs that are close to the design of the logical framework presented in Section 4. While the logical framework in that section is not *necessarily* the final word with respect to my thesis, it is *most definitely* not the final word on substructural specification frameworks. My thesis statement addresses a class of logical frameworks, but my proposal is to gather evidence about a particular logical framework. I do want to generalize the framework as much as possible, but I anticipate placing a higher priority on keeping the process of formal reasoning simple and on presenting a wide variety of examples.

After I have a larger collection of example safety proofs and have settled on the framework's design, I will consider the coverage checking problem in more detail. This investigation will inform

the design of a tool for formally verifying safety properties. I am still considering a range of options for implementing the framework for formally verifying these properties. For example, the framework could be embedded in the Agda programming language [Nor07] in the manner of Licata's thesis work [Lic08] or it could be developed as a standalone language utilizing the infrastructure of Twelf, Celf, and/or Ollibot.

While I propose to describe principles for formally reasoning about specifications in the framework I present, I anticipate that these principles, and any mechanical verification tools I build, will be somewhat specific to the problem of verifying properties of programming languages. In particular, I very much do *not* expect to develop a system that exhibits the kind of uniformity between specification and reasoning about specifications that is seen in metalogical frameworks such as LF/Twelf and HLF/Twelf. Instead, I expect that the tool for formally reasoning about specifications will more closely resemble metatheoretic frameworks for reasoning about specifications (such as $\mathcal{M}_\omega^+$ [Sch00] and $\mathcal{L}_\omega^+$ [MS08]) and languages for functional programming with LF terms (such as Delphin [Pos08] and Beluga [PD10]).

I do not attach artificial timelines to the specific goals in the next section. However, I generally anticipate spending less than a year designing the framework, establishing its metatheory, and specifying and proving properties of SSOS specifications; I then expect to spend about a year developing and using a tool for reasoning about specifications. I would like to graduate in 2012.

7.3.1 Specific goals

Goals are classified as **primary**, **open-ended**, and **secondary**. The primary goals are the major expected original components of this thesis proposal and deserve the most attention. Open-ended goals are intentionally somewhat unspecified; they are areas where I hope to make some progress but also expect to leave significant future work. Secondary goals are mostly related topics that would strengthen the thesis but which may become future work.

A concurrent ordered logical framework

- **Primary: The logical framework.**
  I intend to settle on a specific version of the framework in Section 4 that I will use as the basis of my thesis work. This framework will need to have a full specification, and I will formally establish the standard metatheory of the framework.

  Regardless of the nature of the specified framework, my implementation efforts may concentrate on a fragment without full dependent types.

- **Secondary:** Extend the transition-based account of linear logic in Section 3.2 into a complete account of intuitionistic linear logic.

- **Secondary: Nominal Quantification.**
  Miller et al.'s ∇ quantifier and the nominal existential quantifier in Cervesato and Scedrov's ω rewriting language seem able to capture inequality checks on dynamically-generated parameters. If reasoning about the properties of these quantifiers proves to be tractable, they should be included in the framework.

Properties of programming languages

- **Primary:** Provide progress and preservation proofs for the programming language features (including parallel evaluation, exceptions, mutable state, and call-by-need evaluation) described in the LICS paper on substructural operational semantics [PS09].

- **Open-ended:** Present SSOS specifications and establish safety proofs for larger or more complicated language specifications (such as Harper's Modernized Algol [Har10] or Roşu and Şerbănuţă's Challenge [RŞ10]), or more realistic languages.
- **Open-ended:** Reason about properties like deadlock-avoidance, race-freeness, and secure information flow in SSOS specifications (Section 5.7).
- **Secondary:** Modularly extend ordered SSOS specifications with first-class continuations via program transformation (Section 6.2) and establish safety for the resulting language.

Executable specifications

- **Primary:** Extend the Ollibot forward-chaining logic programming language to implement the full logical framework.
- **Open-ended:** Explore compilation and efficient execution of specifications, particularly through the use of distributed programming languages.
- **Secondary:** Generate program analyses by approximating specifications of programming languages (Section 6.1).

Formal verification of language properties

- **Primary: Coverage checking.**
  The largest theoretical barrier to mechanically verifying properties of SSOS specifications may be reasoning about coverage checking problems in the presence of the equivalence relations on expressions and contexts. This is an area where I expect to dedicate a good bit of effort.
- **Primary:** I plan to write a number of proofs about safety properties of stateful and concurrent programming languages; I also intend to mechanically verify many or all of these proofs using a framework that is appropriate for that task.
- **Open-ended: Context generators.**
  As discussed in Section 5.6, the use of context generators naturally captures and extends the language of regular worlds used in the Twelf metalogical framework. Extending the language of context generators beyond the regular worlds fragment while preserving automatic verification is an interesting, though open-ended, topic.
- **Secondary: Modular proofs.**
  A primary goal is that the *code* of specifications should be modular; furthermore, proofs should be *conceptually* modular — that is, extending a safety proof to handle modular language extensions should be a straightforward exercise. However, depending on the specific strategy used for formalizing proofs, the code of proofs may not be as modular as the specification language.

References

[AFF06] Martin Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for java. *ACM Transactions on Programming Languages and Systems*, 28(2):207–255, 2006.

[BM07] David Baelde and Dale Miller. Least and greatest fixed points in linear logic. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 92–106. Springer LNCS 4790, 2007.

[CGS+05] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538. ACM, 2005.

[Chi95] Jawahar Lal Chirimar. *Proof Theoretic Approach To Specification Languages*. PhD thesis, University of Pennsylvania, 1995.

[CP02] Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information and Computation*, 179(1):19–75, 2002.

[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.

[CS09] Iliano Cervesato and Andre Scedrov. Relating state-based and process-based concurrency through linear logic. *Information and Computation*, 207:1044–1077, 2009.

[Dan08] Olivier Danvy. Defunctionalized interpreters for programming languages. In *Proceedings of the International Conference on Functional Programming*, pages 131–142. ACM, 2008.

[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.

[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. Informal Proceedings, 2009.

[EŞR08] Charles M. Ellison, Traian Florin Şerbănuţă, and Grigore Roşu. A rewriting logic approach to type inference. Technical Report UIUCDCS-R-2008-2934, University of Illinois at Urbana-Champaign, 2008.

[FM08] Amy Felty and Alberto Momigliano. Hybrid: A definitional two level approach to reasoning with higher-order abstract syntax. Submitted for publication, available online: http://www.site.uottawa.ca/~afelty/bib.html, September 2008.

[FRRS08] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly-threaded parallelism in Manticore. In *Proceedings of the International Conference on Functional Programming*, pages 119–130. ACM, 2008.

[Gir01] Jean-Yves Girard. Locus Solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(3):301–506, 2001.

[Har10] Robert Harper. Practical foundations for programming languages. Working draft, available online: http://www.cs.cmu.edu/~rwh/plbook/book.pdf, 2010.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[HL07] Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17(4-5):613–673, 2007.

[HM94] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.

[HR07] Mark Hills and Grigore Roşu. KOOL: An application of rewriting logic to language prototyping and analysis. In *Rewriting Techniques and Applications*, pages 246–256. Springer LNCS 4533, 2007.

[Lic08] Daniel R. Licata. Dependently typed programming with domain-specific logics. Thesis proposal at Carnegie Mellon University, October 2008.

[LPPW05] Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In *Principles and Practice of Declarative Programming*, pages 35–46. ACM, 2005.

[MHR07] Patrick Meredith, Mark Hills, and Grigore Roşu. A K definition of Scheme. Technical Report UIUCDCS-R-2007-2907, University of Illinois at Urbana-Champaign, 2007.

[Mil96] Dale Miller. A multiple-conclusion meta-logic. *Theoretical Computer Science*, 165(1):201–232, 1996.

[Mil08] Dale Miller. A proof-theoretic approach to the static analysis of logic programs. In Christoph Benzmueller, Chad E. Brown, Jörg Siekmann, and Richard Statman, editors, *Reasoning in Simple Type Theory: Festschrift in honour of Peter B. Andrews on his 70th birthday. Studies in Logic*, volume 17. College Publications, 2008.

[Mil09] Dale Miller. Formalizing operational semantic specifications in logic. *Electronic Notes in Theoretical Computer Science*, 246:147–165, 2009. Proceedings of WFLP 2008.

[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.

[MN09] Peter D. Mosses and Mark J. New. Implicit propagation in structural operational semantics. *Electronic Notes in Theoretical Computer Science*, 229(4):49–66, 2009. Proceedings of SOS 2008.

[Mos04] Peter D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:195–228, 2004.

[MR07] José Meseguer and Grigore Roşu. The rewriting logic semantics project. *Theoretical Computer Science*, 373:213–237, 2007.

[MS08] Andrew McCreight and Carsten Schürmann. A meta linear logical framework. *Electronic Notes in Theoretical Computer Science*, 199:129–147, 2008. Proceedings of LFM 2004.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML: Revised*. MIT Press, Cambridge, Massachusetts, 1997.

[NM09] Vivek Nigam and Dale Miller. Algorithmic specifications in linear logic with subexponentials. In *Principles and Practice of Declarative Programming*, pages 129–140. ACM, 2009.

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[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.

[PD10] Brigitte Pientka and Joshua Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR)*, 2010. To appear.

[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.

[Pfe08] Frank Pfenning. On linear inference. Unpublished note, available online: http://www.cs.cmu.edu/~fp/papers/lininf08.pdf, February 2008.

[Pol01] Jeff Polakow. *Ordered Linear Logic and Applications*. PhD thesis, Carnegie Mellon University, 2001.

[Pos08] Adam Brett Poswolsky. *Functional Programming with Logical Frameworks*. PhD thesis, Yale University, 2008.

[PS99] 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.

[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.

[Ree09a] Jason Reed. *A Hybrid Logical Framework*. PhD thesis, Carnegie Mellon University, 2009.

[Ree09b] Jason Reed. A judgmental deconstruction of modal logic. Submitted for publication, available online: http://www.cs.cmu.edu/~jcreed/papers/jdml.pdf, May 2009.

[RŞ10] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 2010. Submitted for publication, available online:
http://fsl.cs.uiuc.edu/index.php/An_Overview_of_the_K_Semantic_Framework.

[Sch00] Carsten Schürmann. *Automating the Meta-Theory of Deductive Systems*. PhD thesis, Carnegie Mellon University, 2000.

[SN08] Anders Schack-Nielsen. Induction on concurrent terms. *Electronic Notes in Theoretical Computer Science*, 196:37–51, 2008. Proceedings of LFMTP 2007.

[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)*, pages 320–326. Springer LNCS 5195, 2008.

[SP08] Robert J. Simmons and Frank Pfenning. Linear logical algorithms. In *Automata, Languages and Programming*, pages 336–347. Springer LNCS 5126, 2008.

[SP09] Robert J. Simmons and Frank Pfenning. Linear logical approximations. In *Partial Evaluation and Program Manipulation*, pages 9–20. ACM, 2009.

[SP10] Robert J. Simmons and Frank Pfenning. Logical approximation for program analysis. Submitted for publication, available online: http://www.cs.cmu.edu/~fp/papers/lapa10.pdf, 2010.

[ŞR10] Traian Florin Şerbănuţă and Grigore Roşu. K-Maude: A rewriting based tool for semantics of programming languages. In *Rewriting Logic and its Applications*. Springer LNCS, 2010. To appear.

[Vir99] Roberto Virga. *Higher-Order Rewriting with Dependent Types*. PhD thesis, Carnegie Mellon University, 1999.

[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.

[Zei09] Noam Zeilberger. *The Logical Basis of Evaluation Order and Pattern-Matching*. PhD thesis, Carnegie Mellon University, 2009.

A Other SSOS specifications

In this section, I present sketches of several other substructural operational semantics specifications that can extend the specifications in Section 5. Each poses some challenge to the framework and to the methodology described in Section 5, and the specifications are roughly ordered from the ones presenting the least significant challenge to the ones presenting the most significant challenge. Most of these sections do not include a full discussion.

- **Section A.1: Mutable state.** Both the static and dynamic semantics can be extended in a modular way to handle state; unfortunately, the proof of safety cannot be extended straightforwardly.
- **Section A.2: Call-by-need suspensions.** Call-by-need is generally understood as an evaluation strategy for functions, but it can also be presented as a stand-alone language feature that simultaneously introduces fixed-points. We describe the dynamic semantics of these recursive suspensions; unfortunately, their static semantics can not be captured appropriately in the style of Section 5.
- **Section A.3: Partial evaluation.** This section presents an SSOS specification of the language of partial evaluation presented by Davies [Dav96]. In order to represent this system in our framework, we would need to allow canonical forms to contain sequences of transitions in the style of CLF.

- **Section A.4: Case analysis on symbols.** Harper's *Practical Foundations of Programming Languages* distinguishes strongly between variables, which are given meaning by substitution, and symbols, which are given meaning by renaming. Our framework can capture much of the character of both, with two exceptions: we cannot enforce that symbols have a dynamic scope, and we cannot perform case analysis on symbols. Different forms of nominal quantification may provide a way to address both of these issues.

A.1 Mutable state

Mutable state is another example of a modular extension to $\mathcal{L}_1$, though the safety proof we have used so far does not extend as straightforwardly to the addition of mutable state. In the dynamic semantics, a reference into the store will be represented by a variable of type loc.

loc : type.

ref : exp → exp.
deref : loc → exp.
update : loc → exp → exp.
lc : loc → exp.

tref : tp → tp.

v/loc : value (lc L).

ref_1 : frame.
deref_1 : frame.
update_1 : exp → frame.
update_2 : exp → frame.

The typing rules and frame typing rules are unsurprising;

celltp : loc → tp → type_l.
of/ref : of E T → of (ref E) (tref T).
of/deref : of E (tref T) → of (deref E) T.
of/update : of E_1 (tref T) → of E_2 T → of (update E_1 E_2) T.
of/loc : celltp L T → of (lc L) (tref T).
off/ref_1 : off ref_1 T (tref T).
off/deref_1 : off deref_1 (tref T) T.
off/update_1 : of E_2 T → off (update_1 E_2) (tref T) T.
off/update_1 : value E_1 → of E_1 (tref T) → off (update_2 E_1) T T.

A.1.1 Dynamic semantics

cell : loc → exp → type_l.
e/loc : eval(loc L) ↠ retn(loc L).
e/ref : eval(ref E) ↠ cont ref_1 · eval E.
e/ref_1 : cont ref_1 · retn E ↠ $\exists l$. retn(loc l) · ¡cell l E.
e/deref : eval(deref E) ↠ cont deref_1 · eval E.
e/deref_1 : cont deref_1 · retn(loc L) · ¡cell L E ↠ retn E · ¡cell L E.

$\mathsf{e/update}$: $\mathsf{eval(update\ E_1\ E_2)} \twoheadrightarrow \mathsf{cont(update_1\ E_2)} \cdot \mathsf{eval\ E_1}$.

$\mathsf{e/update_1}$: $\mathsf{cont(update_1\ E_2)} \cdot \mathsf{retn\ E_1} \twoheadrightarrow \mathsf{cont(update_2\ E_1)} \cdot \mathsf{eval\ E_2}$.

$\mathsf{e/update_2}$: $\mathsf{cont(update_2(loc\ L))} \cdot \mathsf{retn\ E_2} \cdot \text{¡}\mathsf{cell\ L\ E} \twoheadrightarrow \mathsf{retn\ E_2} \cdot \text{¡}\mathsf{cell\ L\ E_2}$

The dynamic semantics require that we relax the definition of contexts: there can be free parameters of type loc as long as each $(l : \mathsf{loc})$ is associated with exactly one linear resource $(x : \mathsf{cell}\, l\, \mathsf{E})$ for some E.

A.1.2 Static semantics

In order to deal with circular references in the store, the derivation for a static semantics takes a cell, *makes up some type for it* (the L in the conclusion of t/cell is universally quantified on the outside, not existentially quantified in the conclusion), and then both declares the cell to have that type by the persistent assumption (celltp L T) and creates the *obligation* that the cell be typed with that type by introducing the linear proposition checkcell L E T.

checkcell : dest → exp → type_l.

t/cell : ¡cell L E ↠ !celltp L T · ¡checkcell L E T.

t/checkcell : ¡checkcell L E T · !of E T ↠ **1**.

A.2 Call-by-need suspensions

Call-by-need suspensions are another use of destinations. When a recursive suspension is evaluated to a value. A destination D associated with a suspension is always in one of three states. The first state, *waiting*, occurs when there is a linear atomic proposition wait D E in the context. The second state, *evaluating*, where there is a linear atomic proposition blackhole D in the context; forcing the destination at this point in a sequential program would normally guarantee nontermination; in the dynamic semantics below it signals an error.[14] The third state is when the destination is permanently associated with a destination by the introduction of a persistent atomic proposition bind D V.

suspdest : type.
susp : exp → exp.
force : exp → exp.
$\mathsf{force_1}$: exp → frame.
suspv : suspdest → exp

v/suspv : value(suspv D).

wait : suspdest → exp → type_l.
blackhole : suspdest → type_l.
bind : suspdest → exp → type.

e/susp : eval(susp E) ↠ ∃d. retn(suspv d) · ¡wait d E.

e/force : eval(force E) ↠ comp $\mathsf{force_1}$ · eval E.

e/force1 : comp $\mathsf{force_1}$ · retn(suspv D) · ¡wait D E ↠ $\mathsf{comp(bind_1\ D)}$ · eval E · ¡blackhole D.

e/force2 : comp $\mathsf{force_1}$ · retn(suspv D) · ¡blackhole D E ↠ raise · ¡blackhole D E.

e/force3 : comp $\mathsf{force_1}$ · retn(suspv D) · !bind D V ↠ retn V.

e/bind : $\mathsf{comp(bind_1\ D)}$ · retn V · !blackhole D ↠ retn V · !bind D V.

[14] Some more thought would need to go into the treatment of black holes in parallel programs.

Call-by-need evaluation can also be presented as a way of evaluating function arguments (as in [PS09]), and suspensions can be made recursive, which gives a better motivation for the blackhole rules. However, this specification cannot be shown to be type safe by using the style of static semantics we have used so far. The rule t/cell works because it consumes the cell atomic proposition, thus guaranteeing that the location L will only be associated with *exactly one* type T. No such guarantee is possible when dealing with bind propositions: if an expression can be typed with two different types, it is possible to violate type safety by firing the rule twice to assign both types to the expression. The alternative construction of static semantics in Section 5.6 seems like a promising way of addressing this issue, however.

A.3 Binding-time analysis

There is an elegant SSOS specification of Davies' work "A Temporal-Logic Approach to Binding-Time Analysis" [Dav96] in a variant of the logical framework presented here. The variation is precisely that we must allow computations to take place inside canonical forms, as seen in the rule seek/prev-z.

The idea of Davies' binding-time analysis system was that, in a given stage, the syntax next E may be encountered, which is an expression that is meant to be evaluated at the next stage. However, the expression E may contain work that needs to be done at this stage, so we use the backwards-chaining seek rules to look for available work, and then perform that work using forward-chaining rules when we find it.

```
next : exp → exp.
prev : exp → exp.

seek : nat → exp → exp → type.

e/next :    eval(next E) · !seek z E E' ↠ retn(next E').

seek/next :   seek N (next E) (next E')
                ← seek (s N) E E'.

seek/prev-s : seek (s N) (prev E) (prev E')
                ← seek N E E'.

seek/prev-z : seek z (prev E) E'
                ← (eval E ↠ retn(next E')).

seek/lam :    seek N (lam λx. E x) (lam λx. E' x)
                ← (Πx. seek N x x → seek N (E x) (E' x)).

seek/app :    seek N (app E1 E2) (app E'1 E'2)
                ← seek N E1 E'1
                ← seek N E2 E'2
```

A.4 Symbols

In this section, I sketch a possible SSOS presentation of the static and dynamic semantics of symbols as presented in Harper's *Practical Foundations of Programming Languages*, Chapter 34.[15] In order to represent this variant, I require the ability to check inequality of parameters in the static semantics; the nominal existential quantifier presented by Cervesato and Scedrov in [CS09] very nearly serves our purpose, and we write $\exists^?$ instead of $\exists$ to emphasize that we are using a different quantifier. Our approach has the problem that we need to allow variables bound existentially in the premise of a rule

[15] Draft as of April 28, 2010

to appear in the conclusion of that rule; this highly nonstandard and possibly logically problematic usage is demonstrated in the rule $\mathsf{e/match}_4$.

The signature defines a type of symbols, and while free symbols are permitted in both the static and dynamic semantics, we impose a regular worlds constraint: each free symbol A is associated with a single persistent assumption (associate A T). The rule in the dynamic semantics ($\mathsf{e/new}_1$) and the rule in the static semantics (of/new) both maintain this regular worlds constraint when they extend the context.

The frames are precisely the ones that we would expect, and expressions are either containers for symbols, scoped fresh symbol generators, or and case analyses on symbols. Besides symbols, the only new syntactic object are the *rules*, which act as potential matches for a symbol.

```
%% Syntax
symbol, tp, exp, frame, rules : type.
associate : symbol → tp → type.

symtp : tp → tp.

sym : symbol → exp.
new : tp → (symbol → exp) → exp.
match : (tp → tp) → exp → (symbol → exp) → rules → exp.

localize : symbol → frame.
match1 : (tp → tp) → (symbol → exp) → rules → exp.

ϵ : rules.
sym? : symbol → exp → rules → rules.
```

The static semantics are relatively straightforward. Just as there are no non-variable canonical forms of type symbol, there are no rules defining canonical forms of the proposition associate A T. This captures the intuition that symbols and their associated types are persistent stateful information, not hypotheses defined by substitution.

```
%% Static Semantics
mobile : tp → type.
of : exp → tp → type.
off : exp → tp → tp → type.
ofrule : rule → (tp → tp) → type.

of/new : of (new TA λa. E a) T
         ← mobile T
         ← (Πa. associate a TA → of (E a) T).

of/sym : of (sym A) (symtp T)
         ← associate A T.

of/match : of (match (λt. T t) E (λa0. E0 a0) R) (T TA)
           ← of E (symtp TA)
           ← (Πa0. associate a0 TA → of (E0 a0) (T TA))
           ← ofrule R (λt. T t).

ofrule/ϵ : of ϵ (λt. T t).

ofrule/sym? of (sym? A E R) (λt. T t)
             ← associate A TA
             ← of E (T TA)
             ← ofrule R (λt. T t).
```

The dynamic semantics are a bit more nonstandard and make heavy use of the nominal existential from Cervesato and Scedrov's ω rewriting language. The use in $\mathsf{e/new}_1$ is the familiar use case where a fresh parameter is generated whenever the rule fires. The use in $\mathsf{e/new}_2$, however, is significant: because Cervesato and Scedrov's nominal quantifier must capture all instances of a variable, the rule will fail to match if the symbol A appears in the value being returned. Therefore, this dynamic semantics is the *ephemeral* or *stack-like* semantics of symbols; the persistent semantics of symbols could be achieved by either modifying $\mathsf{e/new}_1$ to not leave the stack frame $\mathsf{cont(localize}\ A\mathsf{)}$ or by modifying $\mathsf{e/new}_2$ to universally quantify over the symbol A rather than using the nominal existential in the rule's premise.

%% Dynamic Semantics
eval : exp → type$_o$.
retn : exp → type$_o$.
cont : frame → type$_o$.
e/sym : eval(sym A)
  ↠ retn(sym A).

e/new$_1$: eval(new T λa.E a)
  ↠ $\exists^? A$. !associate A T · cont(localize A) · eval(E A).

e/new$_2$: ($\exists^? A$. cont(localize A) · retn V)
  ↠ retn V.

e/match$_1$: eval(match (λt. T t) E (λa_0. E$_0$ a_0) R)
  ↠ cont(match$_1$ (λt. T t) (λa_0. E$_0$ a_0) R) · eval(E).

e/match$_2$: cont(match$_1$ (λt. T t) (λa_0. E$_0$ a_0) ϵ) · retn(sym A)
  ↠ eval(E$_0$ a)

e/match$_3$: cont(match$_1$ (λt. T t) (λa_0. E$_0$ a_0) (sym? A E R)) · retn(sym A)
  ↠ eval(E)

e/match$_4$: ($\exists^? A_1$. $\exists^? A$. cont(match$_1$ (λt. T t) (λa_0. E$_0$ a_0) (sym? A_1 E R))
  · retn(sym A))
  ↠ cont(match$_1$ (λt. T t) (λa_0. E$_0$ a_0) R) · retn(sym A)

The only other use of existential quantification is $\mathsf{e/match}_4$ where it is used to check for the *inequality* of the symbols A_1 and A; our use is nonstandard because A is also mentioned in the conclusion, but this problematic usage could probably be avoided if necessary.