Lecture Notes on
Truth is Ephemeral

15-836: Substructural Logics
Frank Pfenning

Lecture 1
August 29, 2023

1 Introduction

When studying logic in an introductory course we are used to thinking of truth as a mathematical concept, something that is objective and unalterable like the laws of physics. Truth is something we may be able to uncover and understand, but not something we can create. In this course I will try to convince you otherwise.

First, truth is ephemeral. At this point in lecture I held a piece of chalk. Direct evidence confirmed: "*Frank holds a piece of chalk.*" After I put it down on the table in front of me this proposition was no longer true. Hence, evidently, *truth is ephemeral*. Substructural logics capture and analyze this phenomenon, which is of fundamental importance in computer science. For example, while executing a program in an imperative language a variable x may hold the value 5. That's an ephemeral truth, because after assigning x the value 7 it is no longer true—instead, the value of x is then 7.

Second, certain truths are persistent. For example, given the plethora of proofs, it is difficult to deny the Pythagorean Theorem. Or, by the very nature of implication, A always implies A for any proposition A. Substructural logics account for ephemeral as well as persistent truths and therefore *generalize* rather than replace "traditional" logics that study persistent truth. In the context of this course we call such logics *structural*. The origin of the terms *structural* and *substructural* will become clear in the course of this lecture.

Logic is the study of the laws of valid inference, so we start the course in the same way. In today's lecture we avoid logical connectives entirely, just using rules of inference. It turns out that we can use inference rules to describe some algorithms that can be seen as performing logical inferences, which is one of the many connections between logic and computer science.

2 Structural Inference

Consider a relation $\mathsf{edge}(x, y)$ between vertices x and y in a directed graph. We would like to define when there is a path from x to y. Mathematically, we might say that the path relation is the transitive closure of the edge relation. We define this with two rules of inference:

$$\frac{\mathsf{edge}(x, y)}{\mathsf{path}(x, y)}\ \mathsf{Edge} \qquad \frac{\mathsf{path}(x, y) \quad \mathsf{path}(y, z)}{\mathsf{path}(x, z)}\ \mathsf{Trans}$$

Some terminology: the propositions above the line in a rule of inference are called *premises*, the propositions below *conclusions*. The variables in a rule (here x, y, and z) are called *schematic variables*.

The process of inference starts from a given state of knowledge and deduces additional propositions that must be true, according to the rules of inference. We say we *apply a rule of inference*, given a particular instantiation of the schematic variables.

Here is a small example: our initial state of knowledge is $\mathsf{edge}(a, b)$, $\mathsf{edge}(b, c)$, $\mathsf{edge}(b, d)$ for some vertices a, b, c, and d. We apply all possible inferences at each stage, but if a conclusion is already in our database of facts we don't write it down again. The justifications are just the inference rules applied to the labels of all the premises.

(1)	$\mathsf{edge}(a, b)$	given
(2)	$\mathsf{edge}(b, c)$	given
(3)	$\mathsf{edge}(b, d)$	given
(4)	$\mathsf{path}(a, b)$	$\mathsf{Edge}(1)$
(5)	$\mathsf{path}(b, c)$	$\mathsf{Edge}(2)$
(6)	$\mathsf{path}(b, d)$	$\mathsf{Edge}(3)$
(7)	$\mathsf{path}(a, c)$	$\mathsf{Trans}(4, 5)$
(8)	$\mathsf{path}(a, d)$	$\mathsf{Trans}(4, 6)$

At the last stage we have reached *saturation*: any way we can apply inference rules will result in conclusions we already know. Because we think of the rules as *defining* the propositions involved (specifically $\mathsf{path}(x, y)$ in this example) we stop inference at this point. If we want to know if there is a path we can just look it up on this final, saturated state. For example, there is a path from a to d, but there is no path from b to a.

A few remarks about the process of inference. In general, it will not be the case that given some initial facts we reach saturation. For example, rule that allows us to conclude that $\mathsf{nat}(\mathsf{s}(x))$ if $\mathsf{nat}(x)$ would lead to an infinite set of facts if 0 is also known to be a natural number. In our example, the size of the database is bounded by $2n^2$ where n is the number of vertices.

Secondly, the justification gives us a proof of each derived fact. For example, the proof of $\mathsf{path}(\mathsf{a}, \mathsf{d})$ would be

$$\dfrac{\dfrac{\mathsf{edge}(a,b)}{\mathsf{path}(a,b)}\,\mathsf{Edge} \quad \dfrac{\mathsf{edge}(b,d)}{\mathsf{path}(b,d)}\,\mathsf{Edge}}{\mathsf{path}(a,d)}\,\mathsf{Trans}$$

Such a proof can also be expressed as a term, thinking of the inference rule as a term constructor. In this example, this might be written as

$$\mathsf{Trans}(\mathsf{Edge}(1), \mathsf{Edge}(3))$$

substituting the proof terms for (4) and (6).

In general, a proposition may have multiple different proofs, including infinitely many. For example, if we add an edge from b to a, then we can cycle from a to a as many times as we wish. It is therefore important that we do not take the proofs into account when we decide saturation: we want a finite number of facts (each with at least one proof) but not necessarily a finite number of proofs.

We also observe that the order in which we perform inferences is nondeterministic but entirely irrelevant: the saturated state will always be the same. This is an example of so-called *don't-care nondeterminism*. If we tracked proofs, though, they could be different based on the order in which we performed the inferences.

Because (a) the order of the propositions in a state does not matter and (b) knowing a fact once is just as good as "knowing it twice", states form a *set*. Expressed as two laws: $P, Q = Q, P$ and $P, P = P$. Because we view this as an *equality* between states, it can be applied anywhere in a state. We call these properties *exchange* and *contraction*, respectively, although the latter may be called *idempotence* in algebra.

Datalog (see, for example, Maier et al. [2018] or Green et al. [2012]) is a programming language based on structural inference with some additional features such as stratified negation and constraints. Applications in computer science include *program analysis* [Whaley et al., 2005, Smaragdakis and Bravenboer, 2010] and algorithms such as subtyping [DeYoung et al., 2023]. An interesting sidebar is that there are some *meta-complexity theorems* that allows us to read off the complexity of algorithms from the inference rules that define the algorithms [Ganzinger and McAllester, 2001, 2002].

Because it is difficult to express many common algorithms and programming idioms just by inference, we see structural inference primarily as a way to express algorithms that can then be implemented as a library in a language with a more complete set of constructs. Our small and unrealistic implementation as part of Assignment 1 is a tiny example of such a library.

3 Linear Inference

Technically, linear inference arises from structural inference by denying *contraction* while keeping exchange. This means that the state becomes a *multiset* (or *bag*) rather than being a set. Even more importantly, when applying an inference rule we *remove* the premises from a state and then add in the conclusions.

As a first example we consider *coin exchange*: a quarter can be exchanged for two dimes and a nickel, and a dime can be exchanged for two nickels. We can also do the reverse exchange.

$$\frac{q}{d \quad d \quad n}\ \mathsf{Q} \qquad \frac{d \quad d \quad n}{q}\ \overline{\mathsf{Q}} \qquad \frac{d}{n \quad n}\ \mathsf{D} \qquad \frac{n \quad n}{d}\ \overline{\mathsf{D}}$$

The first and third rules exemplify something new, namely rules with multiple conclusions. We cannot replace them with multiple rules, each with a single conclusion since the premises will be consumed from the state during rule application.

Linear inference represents a change of state. Below we show the whole state and all results of possible inferences. As an example, in the first step we replace q by d, d, n, applying the rule Q. The additional n is just carried over.

(1)	q, n	given
(2)	d, d, n, n	$\mathsf{Q}(1)$
(3)	d, n, n, n, n	$\mathsf{D}(2)$
(4)	d, d, d	$\overline{\mathsf{D}}(2)$
(5)	n, n, n, n, n, n	$\overline{\mathsf{D}}(3)$

Note that the proof notation is somewhat approximate now: it only says which rule is applied to which line, but the premises of a rule match only a portion of the state while carrying over the remainder.

At the end of this process the application of any rule to any state will result only in a state already in our collection. We have saturated the set of possible states, rather than arrived at a single saturated state. It should be clear that termination must be redefined in this manner because inference does not monotonically increase our knowledge. Furthermore, just as for structural inference, termination is not guaranteed in general. For example, if we added a rule that allowed us to conclude d, d from d we could produce arbitrarily many dimes starting from any state with at least one.

We accomplish the change of state at the technical level by denying the structural rule of contraction, but we keep the rule of exchange. Linear inference is therefore an example of *substructural inference*.

At first one might think that the right way to handle the "holding-the-chalk" example from the introduction by using a *temporal logic*. That is, at some time t I am holding the chalk, and at some time $t + \delta$ I no longer hold the chalk. And, indeed,

temporal logic has a significant role both in philosophy and computer science. For capturing a change of state (like putting down a piece of chalk, or assigning to a variable) it has a severe drawback: it suffers from the *frame problem*. When I put down my piece of chalk in lecture, all of you remained in your seats. When we assign to a single variable, all the other variables in the program remain unchanged. So we would also have to say "*nothing else changes*", but it is difficult to circumscribe the meaning of "*nothing else*". It depends, and is therefore inherently anti-modular. If we add another variable to our program, for example, suddenly "*nothing else*" would have to include the new variable, even if in some sense it has nothing to do with our particular assignment.

Substructural inference provides an intrinsic solution to the frame problem since the premises of a rule are always matched against a portion of the state. Inherently, no matter what else we add, only the matched portion of the state changes and the remainder is carried over. Therefore, a priori, substructural logic is the better choice for a controlled change of state. Temporal logic is suitable when, intrinsically, many things happen in parallel (e.g., in a logic circuit) or when we want to reason about temporal aspects of a computation that is described by state change. Linear inference has its roots in *linear logic* [Girard, 1987], although the state-changing aspect was formulated perhaps most explicitly in *multiset rewriting* [Cervesato et al., 2000, Cervesato, 2000, Cervesato and Scedrov, 2009] and extended in the Concurrent Logical Framework (CLF) [Watkins et al., 2002, Cervesato et al., 2002, Schack-Nielsen and Schürmann, 2008, Schack-Nielsen, 2011].

4 Ordered Inference

We can go one step further in the exploration of substructural inference by also removing exchange. This means the state is now just a sequence of propositions. We write ordered states without a comma to remind ourselves of the lack of exchange. Algebraically, the state is a monoid, the unit element being the empty state.

The example we start with is recognizing a word consisting of matching parentheses. The constituents of the state are left and right parentheses. We have just one rule of inference

$$\frac{(\quad)}{\cdot}\ \textsf{Cancel}$$

This is an inference rule with zero conclusions, something that would be entirely pointless in structural inference but makes sense for substructural inference. We run through an example, but we don't bother showing the inference rule that was

applied because there is only one.

(1)	(	)	(	(	)	)	(	)	given
(2)			(	(	)	)	(	)	from (1)
(3)	(	)	(			)	(	)	from (1)
(4)	(	)	(	(	)	)			from (1)
(5)			(			)	(	)	from (2), (3) [in 3 ways], or (4)
(6)			(	(	)	)			from (2) or (4)
(7)							(	)	from (5) [in 2 ways] or (6)
(8)									from (7)

Despite the suggestive way we have written the states to make them easier to relate, note that the exact state (5) (namely, ( ) ( )) can be deduced in 5 different ways but we only list it once.

We call the final state (8) *quiescent* because no inference rule can be applied to it.

The claim is that given an ordered state consisting of left and right parentheses, we can deduce the empty state if and only if the parentheses match in the given order. In particular, if we start from a state where the parentheses don't match, we cannot reach the empty state. For example:

(1)	)	(	)	(	given
(2)	)			(	from (1)

Here, we are stuck after one inference can cannot deduce anything new. In other words, the final state (2) is quiescent but not empty.

We observe that in this example we don't actually need to explore *all* reachable states. We can arbitrarily apply cancellation (if possible) and then continue from the resulting state until we reach a quiescent state. We call this *don't-care nondeterminism*: the rules could be applied in multiple ways, but it is correct to pick an arbitrary one. This is sufficient to determine if the initial state represents matching parentheses.

The rules themselves don't contain the assumptions about the initial state, the way the rules are to be applied (in a don't-care nondeterministic manner until quiescent or generating all reachable states), or how to interpret the final state(s). So we should always give this information explicitly.

1. We are given an initial ordered state consisting of left '(' and right ')' parentheses.

2. We apply cancellation in a don't-care nondeterministic manner until we reach a quiescent state.

3. The quiescent state is empty if and only if the initial state had matching parentheses.

We say the problem representation by (structural, linear, or ordered) inference is *adequate* if we can prove, at the meta-level the third part, given the circumstances of the first and second parts.

Here is what we might say for the coin exchange:

1. We are given an initial linear state consisting of quarters ('q'), dimes ('d') and nickels ('n').
2. We apply the rules in *don't-know nondeterministic* manner until we have deduced all reachable states.
3. A state is reachable if and only if it has the same total monetary value as the initial state and consists only of quarters, dimes, and nickels.

5 Binary Increment as Ordered Inference

We explore incrementing a binary number as a second example of ordered inference. We have propositions 0 (bit 0), 1 (bit 1), and ϵ (end of number, not to be confused with the empty word) to represent a natural number in binary form. For example, the number 6 would be represented as the state ϵ 1 1 0. Furthermore, we have the proposition inc which is meant to increment the binary number to its left. We capture this meaning with the following rules:

$$\frac{0 \quad \mathsf{inc}}{1}\ \mathsf{inc}_0 \qquad \frac{1 \quad \mathsf{inc}}{\mathsf{inc} \quad 0}\ \mathsf{inc}_1 \qquad \frac{\epsilon \quad \mathsf{inc}}{\epsilon \quad 1}\ \mathsf{inc}_\epsilon$$

In the inc_1 rule, the increment in the conclusion represents the carry. We run a small example, incrementing the number 5 by 2 to obtain 7. Each line is inferred from the preceding one.

(1)	ϵ	1	0	1	inc	inc	given
(2)	ϵ	1	0	inc	0	inc	by inc_1
(3)	ϵ	1	1		0	inc	by inc_0
(4)	ϵ	1	1		1		by inc_0

In state (2) we had a choice between applying inc_0 to the first or second occurrence of inc, and we arbitrarily picked the first one. Either way would have led to the same quiescent state at the end. We express adequacy using regular expression notation to capture the permissible forms of the ordered state.

1. We are given an initial state in the form $\epsilon\ (0 \mid 1)^*\ \mathsf{inc}^*$.
2. We apply rules in a don't-care nondeterministic manner until we reach quiescence. Each intermediate state will have the form $\epsilon\ (0 \mid 1 \mid \mathsf{inc})^*$.
3. We have computed the output in the form of a final state $\epsilon\ (0 \mid 1)^*$ when we have reached quiescence.

6 Blocks World as Linear Inference

As a final example in today's lecture we present a version of the classic blocks world planning problem as linear inference. We have a robot hand that can hold a single block and a table with possible severals stacks of labeled blocks. The robot hand can pick up a block from the top of a stack and put it down on top of another stack or an empty spot on the table. We can either explore all reachable states, or create a plan to reach a particular goal state.

We start with the following propositions.

- empty (the robot hand is empty)
- $\mathsf{holds}(x)$ (the robot hand holds block x)
- $\mathsf{on}(x, y)$ (block x is on top of block y)

A plausible first attempt at a rule for picking up a block would be to state that we can pick up a block x if (a) the hand is empty, and (b) is no other block on top of it. In addition, if x is on some other block y then after we pick up x it is no longer on y so we need to remove that fact from the state.

$$\frac{\mathsf{empty} \quad \neg\exists z.\, \mathsf{on}(z, x) \quad \mathsf{on}(x, y)}{\mathsf{holds}(x)}\ \mathsf{pickup?}$$

The difficulty with this rule is that the middle premise is intended to check a condition on the state without actually altering the state. For one, we don't have the means to express this since we didn't want to use logical connectives (like negation and the existential quantifier) in this lecture. Perhaps even more troubling is that it would violate our fundamental definition of linear inference which allows us to match the premises against an *arbitrary* portion of the state and carry over the remainder unchanged. But if have a state such as $\mathsf{empty}, \mathsf{on}(b, \mathsf{table}), \mathsf{on}(a, b)$ then adding $\mathsf{on}(c, a)$ would suddenly render the rule inapplicable.

The solution is to express the condition that allows us to pick up a block in a positive way, as another proposition. We then need to maintain the new proposition as part of the inference rules. Here, we add $\mathsf{clear}(x)$ to express that block x is clear, that is, there is nothing on top of x. This should be true exactly if there does not exist a z such that z is on x. The rule then becomes

$$\frac{\mathsf{empty} \quad \mathsf{clear}(x) \quad \mathsf{on}(x, y)}{\mathsf{holds}(x) \quad \mathsf{clear}(y)}\ \mathsf{pickup}$$

Notice that application of this rule will remove $\mathsf{clear}(x)$ from the state (the hand now holds it) and adds $\mathsf{clear}(y)$ (x is no longer on top of y).

We can think of the table itself as a pseudo-block, and mark the empty slots on the table as clear. For example, if there are two spots on the table and on one we have the stack a on top of b we would represent this as the state

$$\mathsf{empty}, \mathsf{clear}(a), \mathsf{on}(a, b), \mathsf{on}(b, \mathsf{table}), \mathsf{clear}(\mathsf{table})$$

In this state we can only pick up a, which would get us to the state

$$\mathsf{holds}(a), \mathsf{clear}(b), \mathsf{on}(b, \mathsf{table}), \mathsf{clear}(\mathsf{table})$$

We can not pick up the table (the pseudo-block) because there is no proposition $\mathsf{on}(\mathsf{table}, x)$. Putting down a block is just the inverse of the rule for picking one up.

$$\frac{\mathsf{holds}(x) \quad \mathsf{clear}(y)}{\mathsf{empty} \quad \mathsf{clear}(x) \quad \mathsf{on}(x, y)}\ \mathsf{putdown}$$

At this point it should be clear that we can easily infer all the reachable state from a valid initial state. But what is a valid initial state? This is not so easy to characterize. For example, we don't want to allow "circular stacks" with $\mathsf{on}(x, y)$ and also $\mathsf{on}(y, x)$. We don't want to allow the table to be on anything. We don't want to allow two different blocks to have the same label a. We don't want the hand to be empty and hold a block at the same time, because it would mean we really have two hands. A general technique for describing valid linear states (including valid initial states, which is the same in this case) is via *generative grammars* [Simmons, 2012], which are beyond the scope of the present lecture, but which we may return to in a future lecture. Here we contend ourselves by saying that the state should consist of distinct blocks and should be "physically possible".

1. We are given a linear initial state consisting of distinct blocks and a finite number of empty spots on the table $\mathsf{clear}(\mathsf{table})$ in a physically possible configuration.
2. Inference rules are applied in a don't-know nondeterministic way.
3. A state is reachable by robot actions from the initial state if and only if we can reach it via linear inference.

Similar to the example of graph reachability, the actual plan for achieving a goal state is encoded in its proof. We will return to the notion of proof at the beginning of the next lecture.

7 Summary

We have introduced three forms of logical inference, without using any notion of logical connective.

Structural inference. States are sets of propositions representing the current state of knowledge that grows monotonically during inference. A state is *saturated* if any inference only deduces facts we already know. Some algorithms can be expressed as sets of inferences rules that must saturate, applying rules in a don't-care nondeterministic manner. It is called *structural* because states are identified up to the structural rules of *exchange* and *contraction*.

Linear inference. States are multisets of propositions. Linear inference consumes the premises of a rule and adds its conclusions, thereby describing a change of state. We can perform don't-know nondeterministic inferences until we have deduced all reachable states, or we can perform don't-care nondeterministic inference until we reach *quiescence* where no further rules can be applied. Linear inference is a form of *substructural inference* because we deny the law of contraction for states (while keeping the law of exchange).

Ordered inference. States are sequences of propositions. Inferences apply to consecutive propositions, replacing them with the (also consecutive) conclusions. Like for linear inference, we can explore all possible reachable states or proceed in a don't-care nondeterministic manner to reach a quiescent state. Ordered inference is another form of substructural inference, denying both the laws of exchange and contraction.

References

Iliano Cervesato. Typed multiset rewriting specifications of security protocols. In A. Seda, editor, *Proceedings of the First Irish Conference on the Mathematical Foundations of Computer Science and Information Technology (MFCSIT'00)*, Cork, Ireland, July 2000. Elsevier Electronic Notes in Theoretical Computer Science. To appear.

Iliano Cervesato and Andre Scedrov. Relating state-based and process-based concurrency through linear logic. *Information and Computation*, 207(10):1044–1077, October 2009.

Iliano Cervesato, Nancy A. Durgin, Max Kanovich, and Andre Scedrov. Interpreting strands in linear logic. In E. Clarke, N. Heintze, and H. Veith, editors, *Proceedings of the Workshop on Formal Methods and Computer Security (FMCS'00)*, Chicago, Illinois, July 2000.

Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

Henry DeYoung, Andreia Mordido, Frank Pfenning, and Ankush Das. Parametric subtyping for structural parametric polymorphism. *CoRR*, abs/2307.13661, July 2023. URL https://arxiv.org/abs/2307.13661. Submitted.

Harald Ganzinger and David A. McAllester. A new meta-complexity theorem for bottom-up logic programs. In T. Nipkow R. Goré, A. Leitsch, editor, *Proceedings of the First International Joint Conference on ArAutomated Reasoning (IJCAR'01)*, pages 514–528, Siena, Italy, June 2001. Springer-Verlag LNCS 2083.

Harald Ganzinger and David A. McAllester. Logical algorithms. In P. Stuckey, editor, *Proceedings of the 18th International Conference on Logic Programming*, pages 209–223, Copenhagen, Denmark, July 2002. Springer-Verlag LNCS 2401.

Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. Datalog and recursive query processing. *Foundations and Trends in Databases*, 5(2):105–195, 2012.

David Maier, K. Tuncay Tekle, Michael Kifer, and David S. Warren. Datalog: Concepts, history, and outlook. In Michael Kifer and Yanhong Annie Liu, editors, *Declarative Logic Programming: Theory, Systems, and Applications*, pages 3–100. ACM and Morgan & Claypool, 2018.

Anders Schack-Nielsen. *Implementing Substructural Logical Frameworks*. PhD thesis, IT University of Copenhagen, January 2011.

Anders Schack-Nielsen and Carsten Schürmann. Celf - a logical framework for deductive and concurrent systems. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning (IJCAR'08)*, pages 320–326, Sydney, Australia, August 2008. Springer LNCS 5195.

Robert J. Simmons. *Substructural Logical Specifications*. PhD thesis, Carnegie Mellon University, November 2012. Available as Technical Report CMU-CS-12-142.

Yannis Smaragdakis and Martin Bravenboer. Using Datalog for fast and easy program analysis. In O. de Moor, G. Gottlob, T. Furche, and A. Sellers, editors, *Datalog Reloaded*, pages 245–251, Oxford, UK, March 2010. Springer LNCS 6702. Revised selected papers.

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog and binary decision diagrams for program analysis. In K.Yi, editor, *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS'05)*, pages 97–118. Springer LNCS 3780, November 2005.

Lecture Notes on
From Inference to Logical Connectives

15-836: Substructural Logics
Frank Pfenning

Lecture 2
August 31, 2023

1 Introduction

Whenever we investigate logic we have to investigate proofs. They have many roles, but in the context of the (sub)structural inference from the first lecture they justify the truth of the propositions in a state. In different examples, this may give rise to different concrete interpretations of proofs: in graph reachability, proofs corresponded to paths, in coin exchange to a sequence of exchange actions, in parentheses matching to a kind of parse tree, in binary increment to a trace of the computation, and in blocks world to a plan to achieve a goal state. While in structural inference it seemed convenient to represent proofs as terms, this did not work so well with substructural inference, at least in part because of rules with multiple (or zero) conclusions. We start this lecture by presenting (but not pursuing in full rigorous detail) an idea proposed by C. B. Aberlé during the first lecture because it is elegant and has some useful properties we can take of advantage later in the course. Will call this proof representation CBA diagrams.

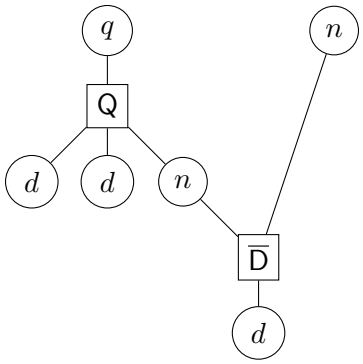
After this, we reflect back on what (substructural) inference can and cannot achieve. One limitation is that we cannot, inside the logic, ask questions such as "*If we start with edges from a to b, from b to c and from b to d, is there a path from a to d?*". Instead, we can only ask this looking at states "from the outside". Asking if-then questions, though, is central to logic so we start our path towards being able to express richer statements. For this, we need logical connectives such as conjunction, implication, disjunction, etc. There will be some surprises along the way, because connectives in substructural logics have some unusual properties.

2 CBA Diagrams for Substructural Proofs

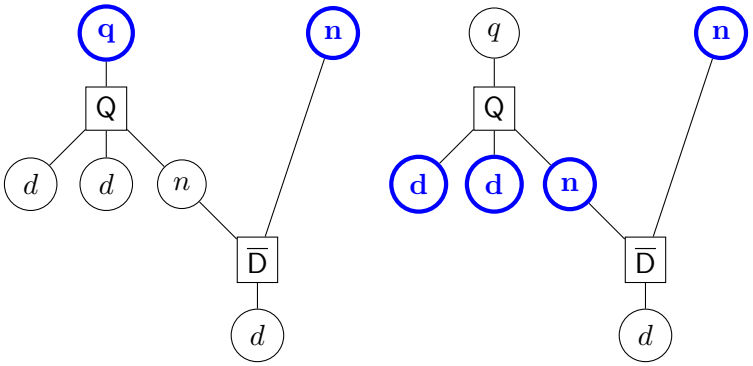
We start with linear inference and the coin exchange example

$$\frac{q}{d \quad d \quad n}\ \mathsf{Q} \qquad \frac{d \quad d \quad n}{q}\ \overline{\mathsf{Q}} \qquad \frac{d}{n \quad n}\ \mathsf{D} \qquad \frac{n \quad n}{d}\ \overline{\mathsf{D}}$$

The idea is that propositions are nodes in a diagram and inference rule applications are boxes connecting premises to conclusions. We build up the diagram here step by step, each inference adding new propositions.

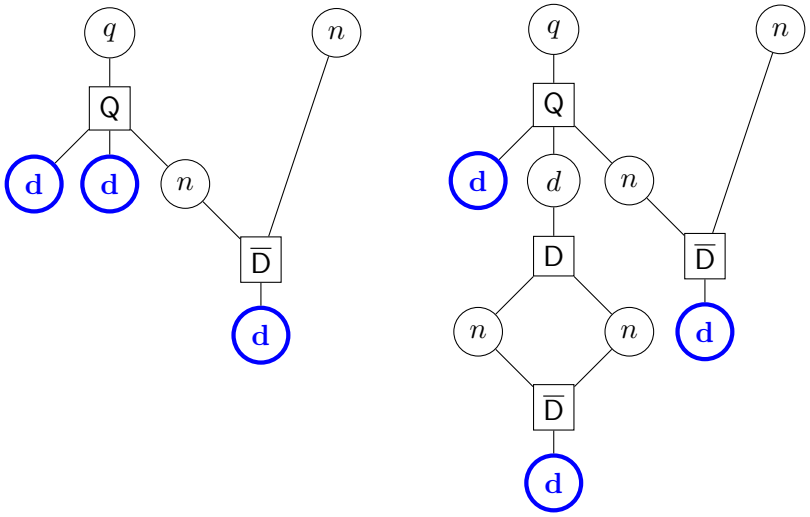


We can now visualize a reachable state as a horizontal slice through the CBA diagram. For example, the initial state would include the top two nodes. On he right, we show the slice after the Q inference. The nodes in the slides are in bold and in blue.



The final state in this example will be the slice just containing the three dimes, shown on the left below. Mathematically, it would be convenient to define the graph and possible slices simultaneously by allowing an inference rule to be applied to an existing slice and moving it by replacing the premises in it by the conclusions. We may still have to account for the fact that identical propositions are indistinguishable. For example, the version of the second diagram where the line

from the left n crosses the one from the right n should be identified. The BCA diagram here succinctly represents a number proofs in one diagram. But we can also change the diagram and it would yield a different proof. For example, we could decide to exchange one of the dimes for two nickels and then back to a dime, as shown on the right where the final slice also contains three dimes.

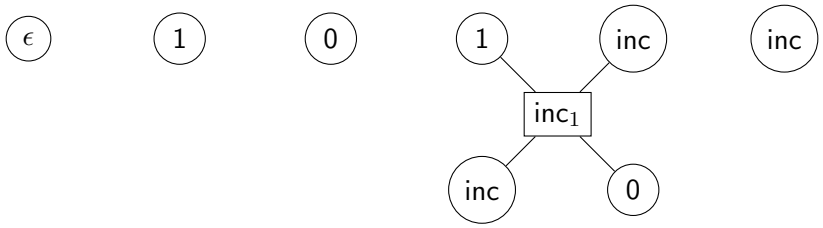


3 CBA Diagrams and True Concurrency

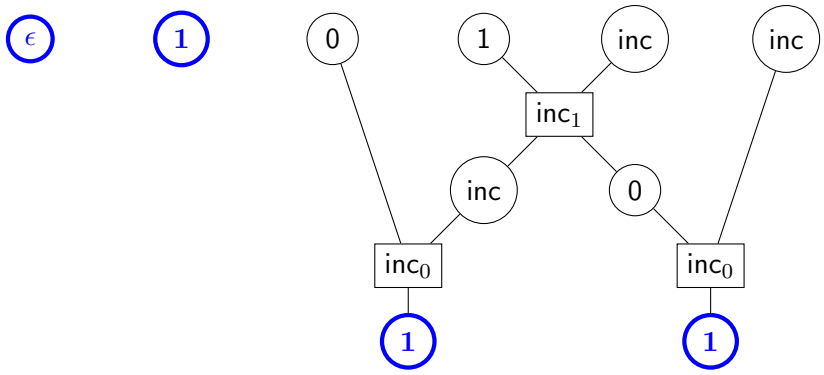
As a next example we considered CBA diagrams for ordered inference, using the example of binary increment.

$$\frac{0 \quad \text{inc}}{1}\ \text{inc}_0 \qquad \frac{1 \quad \text{inc}}{\text{inc} \quad 0}\ \text{inc}_1 \qquad \frac{\epsilon \quad \text{inc}}{\epsilon \quad 1}\ \text{inc}_\epsilon$$

We use the example from the first lecture, incrementing the number 5 twice to arrive at 7. The first step is forced.



At this point, either of the two remaining increments can interact with the 0 to its left. Since they are independent, let's do both.

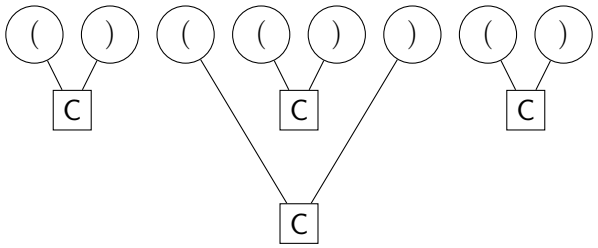


The final slice ϵ 1 1 1 is quiescent. Inference here proceeded with don't care nondeterminism and the last two inferences using inc_0 are *independent* in the sense that neither consumes or produces a proposition that the other needs. Therefore, the order between these two inferences is irrelevant. A nice property of the CBA diagrams here is that you end up with the same diagram (and therefore the same proof) no matter which of the independent actions are taken first. This phenomenon is known as *true concurrency*: we cannot observe the order of independent events. This will be useful later when we specify and reason about parallel and concurrent programming languages.

What is the difference between linear and ordered BCA diagrams? One observation made in lecture is that the lines in order diagrams cannot cross the way they can in linear diagrams. But that's not quite sufficient: the premises of a rule application need to be adjacent. That's not always obvious. As a last substructural example, let's consider matching parentheses.

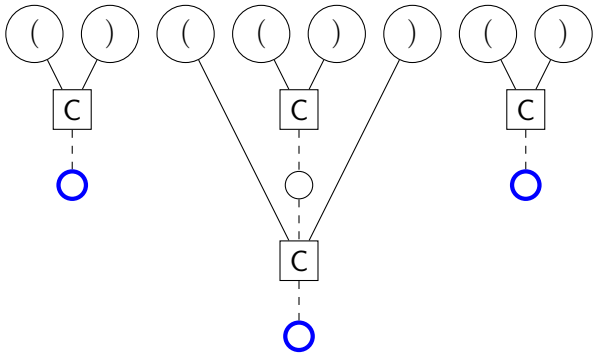
$$\frac{(\quad)}{\cdot}\ \textsf{Cancel}$$

We draw the complete BCA diagram right away for the example from lecture.



Since the rule of cancellation has no conclusions, there are no outgoing edges from the corresponding boxes. Then we see there is essentially only one proof of the

empty slice, because all the actions appear to be independent. However, that's not quite the case: between the two applications that are pictured on top of each other, the one higher up needs to be done first so that the two premises for the lower cancellation are adjacent. We can indicate that, for example, with a dashed line or empty circle indicating zero conclusion, but allowing us to express an otherwise implicit dependency.



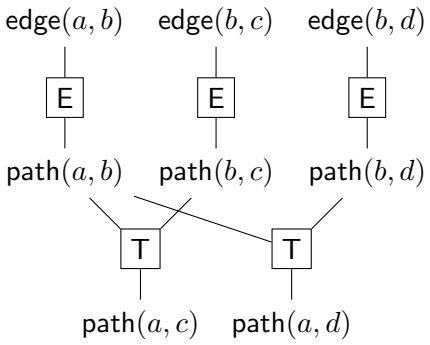
The final slice (shown above in bold blue) is empty.

4 CBA Diagrams for Structural Inference

We can apply the idea of CBA diagrams to structural inference, but slices are somewhat different because of the monotonic nature of inference. We just show the example of graph reachability from last lecture.

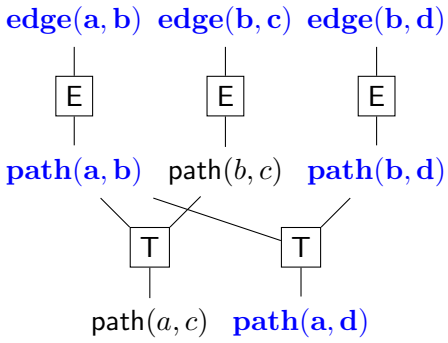
$$\frac{\mathsf{edge}(x,y)}{\mathsf{path}(x,y)}\ \mathsf{Edge} \qquad \frac{\mathsf{path}(x,y) \quad \mathsf{path}(y,z)}{\mathsf{path}(x,z)}\ \mathsf{Trans}$$

We go directly to the diagram at the point of saturation.



A slice now has to be "upwards closed" to capture the fact that inference is monotonic: we only add new facts to the slice. In the diagram below we colored a slice

containing $\mathsf{path}(a, d)$ in bold blue.



While we can intuitively construct such slices that are closed under an ancestor relation, we won't attempt to give a formal definition in this lecture. In particular the fact that there may be multiple proofs of some propositions requires some decisions regarding such a definition.

5 Hypothetical Judgments

Using inference rules we can specify the meaning of basic propositions and reason about them with (sub)structural inference. We now pose several questions in the examples we have considered. We can answer these questions via inference, but, strangely, we cannot even asked them *within* logic because we have no logical connectives!

- If we start with edges from a to b, from b to c, and from b to d, is there a path from a to d?
- Can we exchange a quarter and a nickel for three dimes?
- Is ( ) (( )) ( ) a word with matching parentheses?
- Is ϵ 1 1 1 the result of incrementing ϵ 1 0 1 twice?
- Starting from an initial state where the robot hand is empty, and we have a stack of a on b, with b on the table, and a free spot on the table, can we reach a state where b is on a?

These questions use forms of conjunction and implication, so we have to consider what the meaning of such connectives is and how we can reason with them.

Let's look at the question in the middle: *"Can we exchange a quarter and a nickel for three dimes?"* We are asking if the state with three dimes is reachable from the

state with three dimes. We visualize this question as

$$\begin{array}{c} q, n \\ \vdots \\ d, d, d \end{array}$$

So we not only have an initial state, but also a desired final state. This is a form of a *linear hypothetical judgment*: if we had a quarter and a nickel, could we (by linear inference) reach the state where we have three dimes. To express this *within* the logic, we need to figure out how to *internalize* the components of this hypothetical judgment as logical propositions. For linear logic, it will turn out that A, B (two separate propositions in a state) is expressed as $A \otimes B$. This allows us to combine the initial and final states into a single proposition. Then the vertical dots are expressed as a *linear implication*, that is,

$$\begin{array}{c} A \\ \vdots \\ B \end{array}$$

becomes the proposition $A \multimap B$. So the original situation, as a single propositions, is written as $q \otimes n \multimap d \otimes d \otimes d$.

An inference rule is also an example of a hypothetical judgment. For example,

$$\frac{q}{d\ d\ n}\ \mathsf{Q}$$

expresses that if we *had* a quarter, we *could* exchange it for two dimes and a nickel. So it would be internalized as

$$q \multimap d \otimes d \otimes n$$

There is one caveat, though: an inference rule can be used as many times as we wish, even if the process of inference itself is linear. We say the rule is *persistent* while propositions in the state like q or d are *ephemeral*. In order to express inference rules within the logic we therefore will need to model persistence. We will return to this point later in the lecture.

Focusing on the hypothetical judgment for now, we write

$$\Delta \longrightarrow \Sigma \qquad \text{for} \qquad \begin{array}{c} \Delta \\ \vdots \\ \Sigma \end{array}$$

primarily because it is easier to typeset. This form of hypothetical judgment has some nice properties. For example, it is reflexive and transitive. Furthermore, it affords is the option of reasoning "in two directions". We can either perform an inference starting with Δ, using the inference rules as we have done so far, or we can

conjecture how we might prove Σ and use an inference bottom-up. For example, we might reduce

$$\Delta \longrightarrow d, d, d$$

to proving

$$\Delta \longrightarrow d, d, n, n$$

instead.

Unfortunately this attempt at explaining hypothetical judgments as reachability between states runs into serious problem when we consider implication. How would we prove $A \multimap B$ (remembering that this means "*if we had an A we could deduce B*")? The obvious answer is that we would add A to the state and then attempt to deduce B. That is:

$$\frac{\Delta, A \vdash B, \Sigma}{\Delta \vdash A \multimap B, \Sigma}\ ??$$

Unfortunately, this brings Σ into the scope of A, which is incorrect! For example, the following purported proof is clearly wrong because the hypothesis A is supposed to be available only for the proof of B and not A.

$$\frac{\dfrac{}{A, B \vdash B, A}\ \mathsf{id}}{B \vdash A \multimap B, A}\ ??$$

In order to extract ourselves from such incorrect reasoning we limit the conclusion to be a single formula and write

$$\Delta \vdash A$$

for linear logic, with corresponding judgments for ordered ($\Omega \vdash A$) and structural ($\Gamma \vdash A$) logics. This structure is called a *sequent*, with the state to the left consisting of the *antecedents* and the proposition to the right being the *succedent*.

We can complete a hypothetical proof when a hypothesis (antecedent) matches the conclusion. In the sequent calculus, this is called the rule of *identity*. In the linear and ordered case, this must be exact; in the structural case we can silently ignore some antecedents. This is also a structural property, but it cannot be presented as an equational property. Instead, it should be thought of as a relation between states, $\Gamma \supseteq \Gamma'$. We could either have a general rule of weakening (from $\Gamma \vdash C$ infer $\Gamma, A \vdash C$ for any A) or we can build it into the initial sequents, that is, sequents without premises. We illustrate here the latter.

structural	linear	ordered
$\dfrac{}{\Gamma, A \vdash A}\ \mathsf{id}$	$\dfrac{}{A \vdash A}\ \mathsf{id}$	$\dfrac{}{A \vdash A}\ \mathsf{id}$

6 Internalizing State Formation as Conjunction

The first connective we consider is the one that expresses the state former, written as a comma in structural and linear logic and juxtaposition in ordered logic. We have the following rules, adhering to our convention that Γ is a structural state, Δ is a linear state, and Ω is an ordered state. The rules below are *left rules* because they apply to the proposition among the *antecedents*, that is, to the left of the turnstile '$\vdash$'. Because we internalize state formation, it makes sense to first consider the proposition with the connective to be among the antecedents.

structural	linear	ordered
$\dfrac{\Gamma, A, B \vdash C}{\Gamma, A \wedge B \vdash C}\ \wedge L$	$\dfrac{\Delta, A, B \vdash C}{\Delta, A \otimes B \vdash C}\ \otimes L$	$\dfrac{\Omega_L\ A\ B\ \Omega_R \vdash C}{\Omega_L\ (A \bullet B)\ \Omega_R \vdash C}\ \bullet L$

We see that the notation for the different forms is different. We also see that in the structural and linear cases we write the conjunction in the rightmost position, which is always possible due to the law of exchange. In the ordered case the conjunction can be anywhere in the state, with Ω_L to its left and Ω_R to its right.

According to the structural properties we would expect the following laws to hold or not hold in general. We write $A \dashv\vdash B$ for $A \vdash B$ and $B \vdash A$.

structural	linear	ordered
$A \wedge (B \wedge C) \dashv\vdash (A \wedge B) \wedge C$	$A \otimes (B \otimes C) \dashv\vdash (A \otimes B) \otimes C$	$A \bullet (B \bullet C) \vdash (A \bullet B) \bullet C$
$A \wedge B \dashv\vdash B \wedge A$	$A \otimes B \dashv\vdash B \otimes A$	$P \bullet Q \not\vdash Q \bullet P$
$A \wedge A \dashv\vdash A$	$P \otimes P \not\vdash P$	$P \bullet P \not\vdash P$

In the cases where the entailments do not hold (and neither direction is correct), we use atomic propositions P and Q for our counterexamples because there may be some specific propositions A and B for which such a law might hold.

In order decompose connectives when they appear as a succedent we use *right rules*.

structural	linear	ordered
$\dfrac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge R$	$\dfrac{\Delta_1 \vdash A \quad \Delta_2 \vdash B}{\Delta_1, \Delta_2 \vdash A \otimes B}\ \otimes R$	$\dfrac{\Omega_1 \vdash A \quad \Omega_2 \vdash B}{\Omega_1\ \Omega_2 \vdash A \bullet B}\ \bullet R$

In the structural rule, all antecedents are available in both premises, so there is only one way to apply this rule. This suggests we are reading the rule bottom-up, which is true for our formulations of the sequent calculus. In the linear rule, we have to find a way to split the antecedents among the two premises. Because the antecedents satisfy exchange, any submultiset Δ_1 can be used to prove A, with the remaining antecedents Δ_2 going to the proof of B. So there are 2^n possible ways to

apply this rule when there are n antecedents. In the ordered rule, we have to split the ordered context somewhere, and everything to the left of the split has to prove A, while everything to the right has to prove B. So there are $n+1$ ways to possibly apply this rule when there are n antecedents.

7 Internalizing Hypothetical Judgments as Implication

Among the statements we wanted to express as a logical proposition were if-then statements, such as "*If we had a quarter and a nickel, we could exchange them for three dimes.*" For this, we need implication, which renders the turnstile $\vdash$ as a logical connective. We'll consider this for ordered logic in the next lecture and just focus on structural and linear logic.

Intuitively, $A \supset B$ should be true if B is true under the assumption A. When our logic is structural, A can be used arbitrarily many times in the proof of B. In linear logic, A becomes part of the linear state and will be consumed when inference rules are applied, so we have $A \multimap B$ as a different notation. In this case, we write out the right rules first, because they most naturally relate the meaning of the connective to the hypothetical judgment.

structural	linear
$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B}\ \supset R$	$\dfrac{\Delta, A \vdash B}{\Delta \vdash A \multimap B}\ \multimap R$

What are the corresponding left rules? An assumption $A \supset B$ licenses us to assume B if we can prove A. That is:

structural	linear
$\dfrac{\Gamma \vdash A \quad \Gamma, B \vdash C}{\Gamma, A \supset B \vdash C}\ \supset L$	$\dfrac{\Delta_1 \vdash A \quad \Delta_2, B \vdash C}{\Delta_1, \Delta_2, A \multimap B \vdash C}\ \multimap L$

These, like all rules in our sequent calculi, should be read from the bottom upwards. The slightly subtle point in $\supset L$ is that because the antecedents form a set (and so comma is a form of union), the implication itself still remains in both premises. By contrast, in the linear case we need to split up the antecedents between the two premises and we also remove the implication itself.

As mentioned before, inference rules themselves also form a hypothetical judgment. Below are two examples from the coin exchange:

rule	proposition
$\dfrac{q}{d\ d\ n}\ Q$	$q \multimap d \otimes d \otimes n$
$\dfrac{n\ n}{d}\ \overline{D}$	$n \otimes n \multimap d$

Here is the proof that we can exchange a quarter and a nickel for three dimes. The first few steps are easy. We have offset the inference rules to make them visually easier to read.

$$\dfrac{\dfrac{\vdots}{q \multimap d \otimes d \otimes n, \quad n \otimes n \multimap d, \quad q, n \vdash d \otimes d \otimes d}\otimes L}{q \multimap d \otimes d \otimes n, \quad n \otimes n \multimap d, \quad q \otimes n \vdash d \otimes d \otimes d}\multimap R$$

$$\dfrac{\dfrac{\dfrac{}{q \vdash q}\,\mathsf{id} \quad \dfrac{\vdots}{d \otimes d \otimes n, \quad n \otimes n \multimap d, \quad n \vdash d \otimes d \otimes d}}{q \multimap d \otimes d \otimes n, \quad n \otimes n \multimap d, \quad q, n \vdash d \otimes d \otimes d}\multimap L}{q \multimap d \otimes d \otimes n, \quad n \otimes n \multimap d, \quad q \otimes n \vdash d \otimes d \otimes d}\otimes L}{q \multimap d \otimes d \otimes n, \quad n \otimes n \multimap d \quad \vdash q \otimes n \multimap d \otimes d \otimes d}\multimap R$$

Now we can break up $d \otimes d \otimes n$ and then apply implication left again. A key aspect of the implication left rule is how we split the antecedents, so the two nickels go to the first premises and the two dimes to the second (to be joined by the succedent of the implication, which is the third dime).

$$\dfrac{\dfrac{\dfrac{\dfrac{}{q \vdash q}\,\mathsf{id} \quad \dfrac{\dfrac{\vdots \qquad \vdots}{n, n \vdash n \otimes n \quad d, d, d \vdash d \otimes d \otimes d}}{d, d, n, \quad n \otimes n \multimap d, \quad n \vdash d \otimes d \otimes d}\multimap L}{d \otimes d \otimes n, \quad n \otimes n \multimap d, \quad n \vdash d \otimes d \otimes d}\otimes L^2}{q \multimap d \otimes d \otimes n, \quad n \otimes n \multimap d, \quad q, n \vdash d \otimes d \otimes d}\multimap L}{q \multimap d \otimes d \otimes n, \quad n \otimes n \multimap d, \quad q \otimes n \vdash d \otimes d \otimes d}\otimes L}{q \multimap d \otimes d \otimes n, \quad n \otimes n \multimap d \quad \vdash q \otimes n \multimap d \otimes d \otimes d}\multimap R$$

The remaining steps are straightforward applications of $\otimes R$ and identities.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{n \vdash n}\,\mathsf{id} \quad \dfrac{}{n \vdash n}\,\mathsf{id}}{n, n \vdash n \otimes n}\otimes R \quad \dfrac{\dfrac{}{d \vdash d}\,\mathsf{id} \quad \dfrac{\dfrac{}{d \vdash d}\,\mathsf{id} \quad \dfrac{}{d \vdash d}\,\mathsf{id}}{d, d \vdash d \otimes d}\otimes R}{d, d, d \vdash d \otimes d \otimes d}\otimes R}{d, d, n, \quad n \otimes n \multimap d, \quad n \vdash d \otimes d \otimes d}\multimap L}{\dfrac{}{q \vdash q}\,\mathsf{id} \quad d \otimes d \otimes n, \quad n \otimes n \multimap d, \quad n \vdash d \otimes d \otimes d}\otimes L^2}{q \multimap d \otimes d \otimes n, \quad n \otimes n \multimap d, \quad q, n \vdash d \otimes d \otimes d}\multimap L}{q \multimap d \otimes d \otimes n, \quad n \otimes n \multimap d, \quad q \otimes n \vdash d \otimes d \otimes d}\otimes L}{q \multimap d \otimes d \otimes n, \quad n \otimes n \multimap d \quad \vdash q \otimes n \multimap d \otimes d \otimes d}\multimap R$$

Even though this was not difficult, it is considerably longer and more elaborate that our earlier proof using linear inference. So we pay some price for expressing all the rules and components of the state in propositional form.

We have also cheated. We put exactly one copy of the two needed inference rules among the antecedents. But even in linear inference, the inference rules themselves can be used arbitrarily often. So, really, the propositions $q \multimap d \otimes d \otimes n$ and $n \otimes n \multimap d$ (and the ones corresponding to the other two rules we omitted) should be *persistent*! There are multiple solutions on how to achieve this, which we discuss in the next section.

8 Persistence as a Modality

We have characterized propositions in a linear state as *ephemeral* because they are consumed as part of linear inference. In contrast, propositions in a structural state are *persistent*: they are never removed, even if they may eventually be ignored. In order to internalize (persistent) rules as propositions into linear logic, the simplest way is to make them persistent. Then we have to kinds of antecedents: persistent and ephemeral ones. It is not difficult to imagine what the rules might then look like.

Another solution goes a little further: it also internalizes the very notion of persistence into linear logic as a modal operator $!A$ (pronounced "*of course* A" and sometimes "*bang* A". The key idea is that the proposition $!A$ itself is also linear (rather than persistent), but we have explicit rules to duplicate and delete such propositions. They are the following:

$$\dfrac{\Delta, !A, !A \vdash C}{\Delta, !A \vdash C}\,\mathsf{contraction} \qquad \dfrac{\Delta \vdash C}{\Delta, !A \vdash C}\,\mathsf{weakening} \qquad \dfrac{\Delta, A \vdash C}{\Delta, !A \vdash C}\,!L$$

With these rules we can obtain as many copies of A from $!A$ as we want. The $!L$ rule is also called *dereliction*. But what is the correct right rule? Because we are

supposed to be able to generate as many copies of $!A$ as we want, any proof of A can only depend on propositions that can be duplicated and erased themselves. That is:

$$\frac{!\Delta \vdash A}{!\Delta \vdash !A}\ !R$$

where $!\Delta$ means that every antecedent in Δ has the form $!B$. In the next lecture we will see some techniques to explicitly construct counterexamples to wrong rules, such as the one where we do not restrict the context.

Returning to our previous example, the rules now become

$$\Delta_0 = !(q \multimap d \otimes d \otimes n),\ !(d \otimes d \otimes n \multimap d),\ !(d \multimap n \otimes n),\ !(n \otimes n \multimap d)$$

and it should be easy to see how to construct a proof of

$$\Delta_0 \vdash q \otimes n \multimap d \otimes d \otimes d$$

using the new rules following the blueprint of our previous derivation.

Lecture Notes on
Cut and Identity Elimination

15-836: Substructural Logics
Frank Pfenning

Lecture 3
September 5, 2023

1 Introduction

As we have seen in the last lecture, in order to capture the meaning of implication we needed a hypothetical judgment with a single conclusion C

$$\begin{array}{c} A_1 \ \ldots \ A_n \\ \vdots \\ C \end{array}$$

where the hypotheses $A_1 \ \ldots \ A_n$ are interpreted according to the structural properties under consideration (a set for structural logic, a multiset for linear logic, and a sequence for ordered logic). We wrote this as $\Gamma \vdash A$ (structural), $\Delta \vdash A$ (linear) and $\Omega \vdash A$ (ordered) and wrote some inference rules for the logical connectives.

But how do we know the logical rules are correct? A standard approach due to Tarski [1931] provides mathematical models for the language of logical formulas and thereby gives external notions of soundness and completeness for a set of rules. This is in the tradition of what I called the "descriptive" approach to logic and a worthwhile enterprise. While this is fine as a way to analyze logic from a mathematical point of view, we do have to accept mathematics to start with as the basis for the external semantics, so it has its limitations from the foundational point of view.

In the current section of the course we are more interested in what I called the *creative* use of logic, where the rules are justified internally and therefore themselves create a computational universe. This is often called a *proof-theoretic semantics* because truth is justified by proofs and their structure. There is a long philosophical tradition for such an understanding of logic, perhaps starting with Gentzen [1935] and worked out further by Dummett [1991] and others. The connection between such an approach to logic and computation was noted by Curry [1934] and

Howard [1969]. A seminal paper merging these threads by Martin-Löf [1983] also highlights the difference between propositions (such as $A \supset B$) and judgments (such as "*A is true*" or "*A is false*"). This analysis comes into play here since we differentiate between "*A is a hypothesis*" and "*A is a conclusion*".

In a hypothetical judgment, we can fundamentally either work forward from the hypotheses or backwards from the conclusion we are trying to prove. Because the word "conclusion" is somewhat overloaded (also meaning the judgment "concluded" by an inference rule) we write a *sequent* as $\Omega \vdash A$ and refer to Ω as the *antecedents* and A as the *succedent*. This notion of a sequent (for structural logic) originated in Gentzen [1935]; the version of ordered logic is due to Lambek [1958]. The rules that work forward from the antecedents are called *left rules*, because they apply to propositions on the left of the turnstile '$\vdash$'. The rules that work backward from the succedent are called *right rules* because they apply to propositions on the right of the turnstile. These rules should fundamentally in balance in the following ways:

- If we have an antecedent A we should be able to conclude the succedent A. This corresponds to closing the gap betwee hypotheses and conclusion, if we think in two dimensions.
- If we have proved a succedent A we should be able to assume it as a hypothesis. This corresponds to justifying a hypothesis by a proof, so it no longer is a needed hypothesis.

When taken as rules in ordered logic, these take the following forms

$$\frac{}{A \vdash A}\ \mathsf{id}_A \qquad \frac{\Omega \vdash A \quad \Omega_L\ A\ \Omega_R \vdash C}{\Omega_L\ \Omega\ \Omega_R \vdash C}\ \mathsf{cut}_A$$

We are careful here about the order about the antecedents because, well, we are reasoning in ordered logic.

While these rules are certainly sound and useful, they should also be somehow redundant. For example, there should be sufficiently strong left rules so we can extract the component from a compound antecedent A to prove the succedent A. Conversely, when an antecedent A is used in a proof of the succedent C, we should be able to just use the proof of A wherever we use the hypothesis A. This is the essence of the properties of *identity elimination* and *cut elimination* we tackle in this lecture. They express a form of *harmony* between the left and right rules for a connectives, a property usually shown for *natural deduction* [Gentzen, 1935, Prawitz, 1965, Dummett, 1991] rather than the sequent calculus. Since natural deduction for substructural logics is somewhat delicate, we express and prove the corresponding properties on the sequent calculus—incidentally the path also taken by Gentzen.

2 Right Rules Meeting Left Rules

Rather than presenting fully formed proofs of harmony between the right and left rules, we proceed in small steps, eventually building up enough knowledge to then state the desired theorems and assemble the pieces into their proofs. We proceed connective by connective, which has the added advantage of arriving at our understanding in a modular way. When adding connectives, we (mostly) have to consider the new cases. Some of this reviews the material from the last lecture in a new light.

We foreshadow the idea that we end up with the sequent calculus *without* the rule of cut, and a form of identity that is limited to atomic propositions. In this calculus, cut and general identity should be *admissible*, that is, every instance of these rules should be valid. We used dashed lines to indicate admissible rules, so we are ultimately trying to justify

$$\frac{}{A \vdash A}\ \mathsf{id}_A \qquad \frac{\Omega \vdash A \quad \Omega_L\ A\ \Omega_R \vdash C}{\Omega_L\ \Omega\ \Omega_R \vdash C}\ \mathsf{cut}_A$$

2.1 Ordered Conjunction $A \bullet B$

Ordered conjunction $A \bullet B$ (pronounced A *fuse* B) internalizes the operation that concatenates two ordered states. Therefore we may think of the left rule as *defining* the connective.

$$\frac{\Omega_1\ A\ B\ \Omega_2 \vdash C}{\Omega_1\ (A \bullet B)\ \Omega_2 \vdash C}\ \bullet L$$

We claimed that the corresponding right rule should split the ordered antecedents somewhere devoting the first portion Ω_1 to proving A and the second portion Ω_2 to proving B.

$$\frac{\Omega_1 \vdash A \quad \Omega_2 \vdash B}{\Omega_1\ \Omega_2 \vdash A \bullet B}\ \bullet R$$

First, we want to check that the identity at $A \bullet B$ can be reduced to the identity at A and B.

$$\frac{}{A \bullet B \vdash A \bullet B}\ \mathsf{id}_{A \bullet B} \quad \longrightarrow_E \quad \frac{\dfrac{\dfrac{}{A \vdash A}\ \mathsf{id}_A \quad \dfrac{}{B \vdash B}\ \mathsf{id}_B}{A\ B \vdash A \bullet B}\ \bullet R}{A \bullet B \vdash A \bullet B}\ \bullet L$$

Notice that the first step (thinking bottom-up as we should) in the proof is forced: trying to use the $\bullet R$ rule will fail since there is only a single antecedent which we cannot split into two yet. The good news is that the left/right split worked out in this case.

Second, we should check if we can reduce a cut of proposition $A \bullet B$ into cuts at propositions A and B while preserving the conclusion. The critical we examine here is if a right rule for a connectives meets a corresponding left rule.

$$\dfrac{\dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Omega_1 \vdash A & \Omega_2 \vdash B\end{array}}{\Omega_1\ \Omega_2 \vdash A \bullet B}\bullet R \qquad \dfrac{\mathcal{E}'}{\Omega_L\ A\ B\ \Omega_R \vdash C}\bullet L}{\Omega_L\ \Omega_1\ \Omega_2\ \Omega_R \vdash C}\ \mathsf{cut}_{A \bullet B}$$

We have given names to the derivations of the premises of $\bullet R$ and $\bullet L$ so we can refer to them after the transformation. We observe we can appeal to cut on A between $\mathcal{D}_1$ and $\mathcal{E}'$, and then again on B between $\mathcal{D}_2$ and the result. This yields the following:

$$\longrightarrow_R \quad \dfrac{\begin{array}{cc}\mathcal{D}_2 & \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{E}' \\ \Omega_1 \vdash A & \Omega_L\ A\ B\ \Omega_R \vdash C\end{array}}{\Omega_L\ \Omega_1\ B\ \Omega_R \vdash C}\ \mathsf{cut}_A \\ \Omega_2 \vdash B & \end{array}}{\Omega_L\ \Omega_1\ \Omega_2\ \Omega_R \vdash C}\ \mathsf{cut}_B$$

As we can see, everything works out in both cases.

But what goes wrong if we had the incorrect right rule, swapping the antecedents?

$$\dfrac{\Omega_2 \vdash A \quad \Omega_1 \vdash B}{\Omega_1\ \Omega_2 \vdash A \bullet B}\bullet R?$$

First, we notice that the identity is no longer admissible in the calculus without cut. Here is a counterexample with atomic propositions P and Q.

$$\dfrac{\vdots}{P \bullet Q \vdash P \bullet Q}$$

Starting with $\bullet R?$ won't work, since there is only a single antecedent, which would either have to go to the first or second premise, while the other is empty. Proceeding with $\bullet L$:

$$\dfrac{\dfrac{\vdots}{P\ Q \vdash P \bullet Q}\bullet L}{P \bullet Q \vdash P \bullet Q}$$

Now there are only one rule and three possible pairs of premises with this conclusion, depending on how the antecedents are split.

$$\dfrac{\dfrac{\begin{array}{cc}\vdots & \vdots \\ P\ Q \vdash P & \cdot \vdash Q\end{array}}{P\ Q \vdash P \bullet Q}\bullet R?}{P \bullet Q \vdash P \bullet Q}\bullet L \qquad \dfrac{\dfrac{\begin{array}{cc}\vdots & \vdots \\ Q \vdash P & P \vdash Q\end{array}}{P\ Q \vdash P \bullet Q}\bullet R?}{P \bullet Q \vdash P \bullet Q}\bullet L \qquad \dfrac{\dfrac{\begin{array}{cc}\vdots & \vdots \\ \cdot \vdash P & P\ Q \vdash Q\end{array}}{P\ Q \vdash P \bullet Q}\bullet R?}{P \bullet Q \vdash P \bullet Q}\bullet L$$

No rules apply to any of these premises. Since we have explored all possibilities, the identity is not admissible for the incorrect right rules.

To make matters worse, cut would also not be admissible! You could try the reduction and see that any natural attempt would change the order of Ω_1 and Ω_2 in the conclusion.

$$\dfrac{\dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Omega_2 \vdash A & \Omega_1 \vdash B\end{array}}{\Omega_1\ \Omega_2 \vdash A \bullet B}\ \bullet R?\quad \dfrac{\mathcal{E}'}{\Omega_L\ A\ B\ \Omega_R \vdash C}\ \bullet L}{\Omega_L\ \Omega_1\ \Omega_2\ \Omega_R \vdash C}\ \mathsf{cut}_{A\bullet B}$$

$$\longrightarrow_R ?\quad \dfrac{\begin{array}{cc}\mathcal{D}_2 & \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{E}' \\ \Omega_2 \vdash A & \Omega_L\ A\ B\ \Omega_R \vdash C\end{array}}{\Omega_L\ \Omega_2\ B\ \Omega_R \vdash C}\ \mathsf{cut}_A \\ \Omega_1 \vdash B & \end{array}}{\Omega_L\ \Omega_2\ \Omega_1\ \Omega_R \vdash C}\ \mathsf{cut}_B$$

Of course, this is not a *refutation* of cut elimination. It only shows that a particular way to attempt to prove the admissibility of cut does not work. But we can fashion this failed proof attempt into a counterexample. Let's pick $\Omega_1 = B$ and $\Omega_2 = A$. Then the figure can become

$$\dfrac{\dfrac{\begin{array}{cc}\vdots & \vdots \\ A \vdash A\ \mathsf{id}_A & B \vdash B\ \mathsf{id}_B\end{array}}{B\ A \vdash A \bullet B}\ \bullet R?\quad \dfrac{\mathcal{E}'}{\Omega_L\ A\ B\ \Omega_R \vdash C}\ \bullet L}{\Omega_L\ B\ A\ \Omega_R \vdash C}\ \mathsf{cut}_{A\bullet B}$$

Now we see that if cut (and identity) were admissible, the rule of *exchange* between two ordered(!) antecedents would also be admissible. In other words, our ordered logic would become linear!

If cut and identity were primitive rules (our starting point) then we could even derive exchange, and ordered logic would collapse to linear logic—order wouldn't mean anything. Clearly, the $\bullet R?$ rule would be wrong.

2.2 Left Implication $A \setminus B$

In ordered logic, the usual implication $A \supset B$ splits into two different connectives, $A \setminus B$ (pronounced A under B) and $B \,/\, A$ (pronounced B over A), depending on whether A is added to the left end or the right and of the antecedents. In this case, we view the *right* rule as a definition of the connective.

$$\dfrac{A\ \Omega \vdash B}{\Omega \vdash A \setminus B}\ \setminus R$$

What should the matching left rule be? Starting the proof of the identity gives some intuition.

$$\dfrac{\begin{array}{c}\vdots\\ A\ (A \setminus B) \vdash B\end{array}}{A \setminus B \vdash A \setminus B}\ \setminus R$$

It looks as if the proof of A needs to come from the left of the implication. So let's hypothesize:

$$\dfrac{\Omega_A \vdash A \quad \Omega_L\ B\ \Omega_R \vdash C}{\Omega_L\ \Omega_A\ (A \setminus B)\ \Omega_R \vdash C}\ \setminus L$$

We first check identity expansion.

$$\dfrac{}{A \setminus B \vdash A \setminus B}\ \mathsf{id}_{A\setminus B} \qquad \longrightarrow_E \qquad \dfrac{\dfrac{\dfrac{}{A \vdash A}\ \mathsf{id}_A \quad \dfrac{}{B \vdash B}\ \mathsf{id}_B}{A\ (A \setminus B) \vdash B}\ \setminus L}{A \setminus B \vdash A \setminus B}\ \setminus R$$

Next, the cut reduction when the right rule meets the left rule.

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}'\\ A\ \Omega \vdash B\end{array}}{\Omega \vdash A \setminus B}\ \setminus R \quad \dfrac{\begin{array}{c}\mathcal{E}_1\\ \Omega_A \vdash A\end{array} \quad \dfrac{\begin{array}{c}\mathcal{E}_2\\ \Omega_L\ B\ \Omega_R \vdash C\end{array}}{\Omega_L\ \Omega_A\ (A \setminus B)\ \Omega_R \vdash C}\ \setminus L}{\Omega_L\ \Omega_A\ \Omega\ \Omega_R \vdash C}\ \mathsf{cut}_{A\setminus B}$$

$$\longrightarrow_R \qquad \dfrac{\begin{array}{c}\mathcal{E}_1\\ \Omega_A \vdash A\end{array} \quad \dfrac{\begin{array}{c}\mathcal{D}'\\ A\ \Omega \vdash B\end{array}}{\Omega_A\ \Omega \vdash B}\ \mathsf{cut}_A \quad \dfrac{\begin{array}{c}\mathcal{E}_2\\ \Omega_L\ B\ \Omega_R \vdash C\end{array}}{\Omega_L\ \Omega_A\ \Omega\ \Omega_R \vdash C}\ \mathsf{cut}_B$$

It works out! A significant observation here is that the result of the reduction is not unique. For example, we could have cut $\mathcal{D}'$ with $\mathcal{E}_2$ first, and then $\mathcal{E}_1$ with the end sequent.

3 Right Implication $B \,/\, A$

This is symmetric to the left implication, so we just show the rules. It is a recommended exercise to go through the cases of identity expansion and cut reduction.

$$\dfrac{\Omega\ A \vdash B}{\Omega \vdash B \,/\, A}\ /R \qquad \dfrac{\Omega_A \vdash A \quad \Omega_L\ B\ \Omega_R \vdash C}{\Omega_L\ (B \,/\, A)\ \Omega_A\ \Omega_R \vdash C}\ /L$$

4 Excursion: Parsing with the Lambek Calculus

A point in Lambek's original calculus [Lambek, 1958] is to model natural language parsing by (ordered) logical inference. For this purpose, we use the following specialized versions of the $\backslash L$ and $\backslash R$ rules:

$$\frac{\Omega_L\ B\ \Omega_R \vdash C}{\Omega_L\ A\ (A \setminus B)\ \Omega_R \vdash C}\ \backslash L^* \qquad \frac{\Omega_L\ B\ \Omega_R \vdash C}{\Omega_L\ (B\ /\ A)\ A\ \Omega_R \vdash C}\ /L^*$$

Their soundness is easy to see, but it turns out we can't quite replace the general rules with these specialized ones (see Tasks 8 and 9 of Assignment 2).

Then we assign a syntactic category to every word appearing in a sentence. We start with n for names such as *Alice* or *Bob*. We also have s for complete sentences. An intransitive verb such as *works* has category $n \setminus s$ which means that if we find a name to its left then their combination forms a sentence.

To start the inference process we annotate each word with its syntactic category, writing $(w : A)$. We would like to parse a sequence of words as a sentence, so the succedent of our sequent is s.

$$(\mathit{Alice} : n)\ (\mathit{works} : n \setminus s) \vdash s$$

In this case, we can complete it in just two steps. We just concatenate the two proof terms for the result of $\backslash L^*$.

$$\frac{\dfrac{}{(\mathit{Alice} \cdot \mathit{works}) : s \vdash s}\ \mathsf{id}}{(\mathit{Alice} : n)\ (\mathit{works} : n \setminus s) \vdash s}\ \backslash L^*$$

Note that the proof of s here represents a parse tree, in this case trivial. What about adjectives such as *poor*? If *poor* precedes a name, the phrase again functions as a name. So $(\mathit{poor} : n\ /\ n)$. We would then parse "*poor Alice works*" as

$$\frac{\dfrac{\dfrac{}{((\mathit{poor} \cdot \mathit{Alice}) \cdot \mathit{works} : s) \vdash s}\ \mathsf{id}}{(\mathit{poor} \cdot \mathit{Alice} : n)\ (\mathit{works} : n \setminus s) \vdash s}\ \backslash L^*}{(\mathit{poor} : n\ /\ n)\ (\mathit{Alice} : n)\ (\mathit{works} : n \setminus s) \vdash s}\ /L^*$$

A transitive verb such as *likes* has category $n \setminus (s\ /\ n)$: if there is a name to its left and to its right, then the result is a sentence. For example:

$$(\mathit{poor} : n\ /\ n)\ (\mathit{Alice} : n)\ (\mathit{likes} : n \setminus (s\ /\ n))\ (\mathit{Bob} : n) \vdash s$$

This example illustrates some nondeterminism in the parsing process. For example, $(\mathit{Alice} : n)$ is to the left of $(\mathit{likes} : n \setminus (s/n)$ so we could apply $\backslash L^*$, but eventually we would get stuck at

$$(\mathit{poor} : n\ /\ n)\ ((\mathit{Alice} \cdot \mathit{likes}) \cdot \mathit{Bob} : s) \vdash s$$

which we cannot complete. Instead, we need to group $poor$ with $Alice$ first so that $(poor \cdot Alice : n)$. Then we can arrive at

$$(((poor \cdot Alice) \cdot likes) \cdot Bob : s) \vdash s$$

Below is a small table with syntactic categories of other words.

Word	Type	Part of Speech
works	$n \setminus s$	intransitive verb
poor	$n \,/\, n$	adjective
here	$s \setminus s$	adverb
never	$(n \setminus s) \,/\, (n \setminus s)$	adverb
likes	$n \setminus (s \,/\, n)$	transitive verb
and	$s \setminus (s \,/\, s)$	conjunction
and	$s \setminus (n^* \,/\, s)$	conjunction

For the last example we use n^* for a plural name so that phrases like *Alice and Bob work* can be parsed correctly with $(work : n^* \setminus s)$. But "*and*" is more complicated because it is overloaded. For example, it would also have syntactic type $s \setminus (s / s)$ because it can conjoin two sentences into a longer sentence. This cannot be expressed in Lambek's original calculus, but it fits into the so-called *full Lambek calculus* which we have called *ordered logic*. We can write

$$and : (s \setminus (s \,/\, s)) \,\&\, (s \setminus (n^* \,/\, s))$$

where $A \,\&\, B$ is a new connective. If we have the antecedent $A \,\&\, B$ we can choose between A and B, while $A \bullet B$ necessarily gives us both, next to each other. We give the rules below, in Section 6.

5 A Small Example

From the parsing intuition, we would expect $A \setminus (C \,/\, B)$ to be equivalent to $(A \setminus C) \,/\, B$—a form of associativity. Whether we prove A to the left or B to the right first should be irrelevant, since we need both before we obtain C.

Let's prove one direction.

$$\vdots$$
$$A \setminus (C \,/\, B) \vdash (A \setminus C) \,/\, B$$

It turns out that the right rules for the two forms of ordered implication are *invertible* in the sense that we can always apply them during bottom-up proof construction without ever considering alternatives.

It turns out the every connective is either invertible on the right or on the left. A quick test (although not a proof) to see which one, see which side the proof of

the identity starts on. For $B \,/\, A$ and $A \setminus B$ it starts with a right rule. So we can start as follows without having to think:

$$\dfrac{\dfrac{\dfrac{\vdots}{A\ (A \setminus (C \,/\, B))\ B \vdash C}}{(A \setminus (C \,/\, B))\ B \vdash A \setminus C}\setminus R}{A \setminus (C \,/\, B) \vdash (A \setminus C) \,/\, B}/R$$

At this point we can only apply $\setminus L$ because it is the only connective at the top level among all antecedents and the succedent.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dots\dots}{A \vdash A}\ \mathsf{id} \qquad \dfrac{\vdots}{(C \,/\, B)\ B \vdash C}}{A\ (A \setminus (C \,/\, B))\ B \vdash C}\setminus L}{(A \setminus (C \,/\, B))\ B \vdash A \setminus C}\setminus R}{A \setminus (C \,/\, B) \vdash (A \setminus C) \,/\, B}/R$$

The open subgoal now follows by $/L$ and two identities.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dots\dots}{A \vdash A}\ \mathsf{id} \qquad \dfrac{\dfrac{\dots\dots}{B \vdash B}\ \mathsf{id} \quad \dfrac{\dots\dots}{C \vdash C}\ \mathsf{id}}{(C \,/\, B)\ B \vdash C}/L}{A\ (A \setminus (C \,/\, B))\ B \vdash C}\setminus L}{(A \setminus (C \,/\, B))\ B \vdash A \setminus C}\setminus R}{A \setminus (C \,/\, B) \vdash (A \setminus C) \,/\, B}/R$$

The entailment in the other direction proceeds in a similar vein.

6 External Choice ($A \,\&\, B$)

The parsing example suggests the following left rules:

$$\dfrac{\Omega_L\ A\ \Omega_R \vdash C}{\Omega_L\ (A \,\&\, B)\ \Omega_R \vdash C}\,\&L_1 \qquad \dfrac{\Omega_L\ B\ \Omega_R \vdash C}{\Omega_L\ (A \,\&\, B)\ \Omega_R \vdash C}\,\&L_1$$

What is the corresponding right rule? Rather then splitting the antecedents as for $A \bullet B$, we propagate all of them to both premises.

$$\dfrac{\Omega \vdash A \quad \Omega \vdash B}{\Omega \vdash A \,\&\, B}\,\&R$$

At first glance it may seem that this violates the principle that every hypothesis is used exactly once. But any *use* of an antecedent $A \,\&\, B$ will pick either just A or just B. This is reflected in the two local reductions.

$$\dfrac{\dfrac{\mathcal{D}_1 \quad \mathcal{D}_2}{\Omega \vdash A \quad \Omega \vdash B}\,\&R \qquad \dfrac{\mathcal{E}'}{\Omega_L\ A\ \Omega_R \vdash C}\,\&L_1}{\Omega_L\ \Omega\ \Omega_R \vdash C}\,\mathsf{cut}_{A\&B}$$

$$\longrightarrow_R \qquad \dfrac{\dfrac{\mathcal{D}_1}{\Omega \vdash A} \quad \dfrac{\mathcal{E}'}{\Omega_L\ A\ \Omega_R \vdash C}}{\Omega_L\ \Omega\ \Omega_R \vdash C}\,\mathsf{cut}_A$$

$$\dfrac{\dfrac{\mathcal{D}_1 \quad \mathcal{D}_2}{\Omega \vdash A \quad \Omega \vdash B}\,\&R \qquad \dfrac{\mathcal{E}'}{\Omega_L\ B\ \Omega_R \vdash C}\,\&L_2}{\Omega_L\ \Omega\ \Omega_R \vdash C}\,\mathsf{cut}_{A\&B}$$

$$\longrightarrow_R \qquad \dfrac{\dfrac{\mathcal{D}_2}{\Omega \vdash B} \quad \dfrac{\mathcal{E}'}{\Omega_L\ B\ \Omega_R \vdash C}}{\Omega_L\ \Omega\ \Omega_R \vdash C}\,\mathsf{cut}_B$$

We see both premises of $\&R$ are necessary, because we don't know which left rule ($\&L_1$ or $\&L_2$) it might meet. Also, if we had split the antecedents, then we wouldn't have enough of them in one branch or the other, or both. This would lend itself to a counterexample.

The identity is straightforward, and we must start with the right rule. This means the right rule for $A \,\&\, B$ is invertible.

$$\dfrac{}{A \,\&\, B \vdash A \,\&\, B}\,\mathsf{id}_{A\&B} \qquad \longrightarrow_E \qquad \dfrac{\dfrac{\dfrac{}{A \vdash A}\,\mathsf{id}_A}{A \,\&\, B \vdash A}\,\&L_1 \qquad \dfrac{\dfrac{}{B \vdash B}\,\mathsf{id}_B}{A \,\&\, B \vdash B}\,\&L_2}{A \,\&\, B \vdash A \,\&\, B}\,\&R$$

Here we notice that both left rules are necessary. If we just had one (say, $\wedge L_1$) we wouldn't be able to complete the identity expansion because $A \,\&\, B \vdash B$ would not have a proof.

7 The Empty State (1)[1]

[1] covered in Lecture 4, but for continuity included here

We can also internalize the empty state as the proposition $\mathbf{1}$. As a left rule, the proposition simply disappears. As a right rule, the state must be empty. We can think of this as the unit for *fuse* in the sense that $A \bullet \mathbf{1} \dashv\vdash A \dashv\vdash \mathbf{1} \bullet A$.

$$\frac{}{\cdot \vdash \mathbf{1}}\ \mathbf{1}R \qquad \frac{\Omega_L\ \Omega_R \vdash C}{\Omega_L\ (\mathbf{1})\ \Omega_R \vdash C}\ \mathbf{1}L$$

Identity expansion and cut reduction are immediate, since there are no subformulas of $\mathbf{1}$.

$$\dfrac{\dfrac{}{\cdot \vdash \mathbf{1}}\ \mathbf{1}R \quad \dfrac{\overset{\mathcal{E}'}{\Omega_L\ \Omega_R \vdash C}}{\Omega_L\ (\mathbf{1})\ \Omega_R \vdash C}\ \mathbf{1}L}{\Omega_L\ \Omega_R \vdash C}\ \mathsf{cut}_{\mathbf{1}} \quad \longrightarrow_R \quad \overset{\mathcal{E}'}{\Omega_L\ \Omega_R \vdash C}$$

$$\dfrac{}{\mathbf{1} \vdash \mathbf{1}}\ \mathsf{id}_{\mathbf{1}} \quad \longrightarrow_E \quad \dfrac{\dfrac{}{\cdot \vdash \mathbf{1}}\ \mathbf{1}R}{\mathbf{1} \vdash \mathbf{1}}\ \mathbf{1}L$$

8 Disjunction ($A \oplus B$)[2]

Disjunction (also called *internal choice*) changes remarkably little from structural to linear to ordered logic. We may think of it as being defined by the following two right rules:

$$\frac{\Omega \vdash A}{\Omega \vdash A \oplus B}\ \oplus R_1 \qquad \frac{\Omega \vdash B}{\Omega \vdash A \oplus B}\ \oplus R_2$$

The situation is almost entirely symmetric to the one for *external choice* ($A \,\&\, B$). If $A \oplus B$ is among our antecedents, we do not now whether A or B will be true, but we know only one of the two rules will be applied.

$$\frac{\Omega_L\ A\ \Omega_R \vdash C \quad \Omega_L\ B\ \Omega_R \vdash C}{\Omega_L\ (A \oplus B)\ \Omega_R \vdash C}\ \oplus L$$

You can now easily convince yourself that identity expansion and cut reduction are possible.

9 Truth ($\top$)

Truth is the unit of external choice in the sense that $A \,\&\, \top \dashv\vdash A \dashv\vdash \top \,\&\, A$. Because external choice has two left rules, its nullary version will have none. Conversely,

[2] covered in Lecture 4, but for continuity included here

the right rule for external choice has two premises, so the right rule for its nullary version has none.

$$\frac{}{\Omega \vdash \top}\ \top R \qquad \text{no } \top L \text{ rules}$$

Because there is no left rule, the right rule cannot meet any left rule and there is no cut reduction. But there is a simple identity expansion:

$$\frac{\vdots}{\top \vdash \top}\ \mathsf{id}_\top \quad \longrightarrow_E \quad \frac{}{\top \vdash \top}\ \top R$$

10 Falsehood (0)

Falsehood is the unit of internal choice in the sense that $\mathbf{0} \oplus A \dashv\vdash A \dashv\vdash A \oplus \mathbf{0}$. Its properties are symmetric to that of truth ($\top$).

$$\text{no } \mathbf{0}R \text{ rules} \qquad \frac{}{\Omega_L\ (\mathbf{0})\ \Omega_R \vdash C}\ \mathbf{0}L$$

Because there are no right rules, there cannot be a case where a right rule meets a left rule. But an identity expansion is possible.

$$\frac{\vdots}{\mathbf{0} \vdash \mathbf{0}}\ \mathsf{id}_\mathbf{0} \quad \longrightarrow_E \quad \frac{}{\mathbf{0} \vdash \mathbf{0}}\ \mathbf{0}L$$

This concludes the introduction of the connectives; see the summary in Figure 1.

11 Admissibility of Identity, as a Theorem

We can put together all the cases for identity expansions in the following theorem.

Theorem 1 (Admissibility of Identity) *In the system with identity restricted to atomic propositions, the rule*

$$\frac{\vdots}{A \vdash A}\ \mathsf{id}_A$$

is admissible for every A.

Proof: By induction on the structure of A. Many of the individual cases were presented in lecture; the others are analogous. □

12 Admissibility of Cut, as a Theorem

The cut reductions we have presented so far only cover the case where the cut combines a right rule for a connective with its left rule. There are two other classes of cases: if one of the premises is the identity (whether atomic or not), and when the last inference on one or both sides is *not* on the cut formula.

Cuts and identity cancel each other. There are only two cases; we show one.

$$\dfrac{\dfrac{}{A \vdash A}\,\mathsf{id}_A \qquad \begin{array}{c}\mathcal{E}'\\ \Omega_L\ A\ \Omega_R \vdash C\end{array}}{\Omega_L\ A\ \Omega_R \vdash C}\,\mathsf{cut}_A \qquad \longrightarrow_R \qquad \begin{array}{c}\mathcal{E}'\\ \Omega_L\ A\ \Omega_R \vdash C\end{array}$$

Now we consider a case of $\bullet L$ in $\mathcal{D}$, where $\mathcal{E}$ remains completely arbirary.

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}'\\ \Omega_1\ B_1\ B_2\ \Omega_2 \vdash A\end{array}}{\Omega_1\ (B_1 \bullet B_2)\ \Omega_2 \vdash A}\,\bullet L \qquad \begin{array}{c}\mathcal{E}\\ \Omega_L\ A\ \Omega_R \vdash C\end{array}}{\Omega_L\ \Omega_1\ (B_1 \bullet B_2)\ \Omega_2\ \Omega_R \vdash C}\,\mathsf{cut}_A$$

Since $\mathcal{D}'$ still has succedent A, we can now cut it with $\mathcal{E}$ and then apply $\bullet L$ *afterwards*.

$$\longrightarrow_R \qquad \dfrac{\dfrac{\begin{array}{c}\mathcal{D}'\\ \Omega_1\ B_1\ B_2\ \Omega_2 \vdash A\end{array} \qquad \begin{array}{c}\mathcal{E}\\ \Omega_L\ A\ \Omega_R \vdash C\end{array}}{\Omega_L\ \Omega_1\ B_1\ B_2\ \Omega_2\ \Omega_R \vdash C}\,\mathsf{cut}_A}{\Omega_L\ \Omega_1\ (B_1 \bullet B_2)\ \Omega_2\ \Omega_R \vdash C}\,\bullet L$$

In essence, we are "pushing the cut up", past the preceding inference. There is some nondeterminism here. For example, if $\mathcal{E}$ ends in a right rule, or a left rule on a proposition that is not A, then we cut push it up into the second premise as well.

The immediate concern should be that we have *not reduced the structure of the cut formula*: it is still A! However, we reduced the cut from one on $\mathcal{D}$ (the first premise of the cut) to $\mathcal{D}'$, a subproof.

All other cases of this kind often called *commutative cases* proceed in an analogous manner. So we summarize: we reduce a cut on

$$\dfrac{\begin{array}{c}\mathcal{D}\\ \Omega \vdash A\end{array} \qquad \begin{array}{c}\mathcal{E}\\ \Omega_L\ A\ \Omega_R\end{array}}{\Omega_L\ \Omega\ \Omega_R \vdash C}\,\mathsf{cut}_A$$

either

1. to cuts on a subformulas of A, or
2. to cuts on the same formula A but subderivations of $\mathcal{D}$ or $\mathcal{E}$.

Such reductions must always terminate, either because we reach derivations without subderivations, or because we reach formulas without subformula. Formally, it is a well-founded induction on a *lexicographic ordering*, first on A and then on $\mathcal{D}$ and $\mathcal{E}$. This is also called a *nested induction*. We summarize in the following theorem.

Theorem 2 (Admissibility of Cut) *In the inference system without cut (where the identity may be general or restricted to atoms), the rule*

$$\frac{\begin{array}{cc}\mathcal{D} & \mathcal{E} \\ \Omega \vdash A & \Omega_L\ A\ \Omega_R\end{array}}{\Omega_L\ \Omega\ \Omega_R \vdash C}\ \mathsf{cut}_A$$

is admissible.

Proof: By nested induction, first on the structure of A, and second on the structures of $\mathcal{D}$ and $\mathcal{E}$. We have the following classes of cases:

Principal Cases: $\mathcal{D}$ ends in a right rule inferring A and $\mathcal{E}$ ends in a left rule inferring A. We have shown several of these cases and we appeal to the induction hypothesis on subformulas on A. If A has no subformulas we have a base case (as for $\mathbf{1}$).

Identity Cases: Either $\mathcal{D}$ or $\mathcal{E}$ is an identity. Then we directly reduce to the other derivation.

Commuting Cases: Either $\mathcal{D}$ or $\mathcal{E}$ ends in an inference on a formula other than A. In this case we appeal to the induction hypothesis, possible in more than one way, on the same A and a subderivation on $\mathcal{D}$ or $\mathcal{E}$, and the reapply the inference.

□

From the admissibility of identity and cut we obtain the following straightforward corollaries that follow by straightforward induction over the structure of the given derivation.

Corollary 3 (Identity Elimination) *Given an arbitrary sequent derivation with uses of the identity (with or without cut). Then we can eliminate all uses of the identity except on atomic propositions P, obtaining a derivation with or without cut, respectively.*

Proof: By induction on the structure of the given derivation. We appeal to the induction hypothesis and reapply the rule, except for identity when we appeal to the admissibility of identity. □

Corollary 4 (Cut Elimination) *Given an arbitary sequent derivation with uses of cut (with or without a general identity). Then we can eliminate all uses of cut, obtaining a derivation with or without general identity, respectively.*

Proof: By induction on the structure of the given derivation. We appeal to the induction hypothesis and reapply the rule, except for cut when we appeal to the admissibility of cut. □

13 Summary

The rules for the sequent calculus, marking cut and identity as admissible, are summarized in Figure 1.

The admissibility of cut and identity in the sequent calculus without these rules (except for identity on atomic formulas) is the key property to ensure we have a proof-theoretic semantics for a logic. Cut-free proofs in particular always only refer to subformulas of the original goal sequents, so any semantic content is internal to what we are trying prove.

While we have shown some details for ordered logic, similar (and slightly more complicated) arguments apply to linear and structural logics and, eventually, to logics mixing these. This is the blueprint, and future proofs of identity and cut elimination will often be discussed via the differences from the present approach.

In a few lectures from now we will see that sequent proofs correspond to programs, and principal cut reductions play a fundamental role in interpreting the dynamics of programs. Identity expansions and commuting reductions correspond to equality between programs and are therefore slightly less significant.

$$\frac{}{A \vdash A}\ \mathsf{id}_A \qquad \frac{}{P \vdash P}\ \mathsf{id}^*_P \qquad \frac{\Omega \vdash A \quad \Omega_L\ A\ \Omega_R \vdash C}{\Omega_L\ \Omega\ \Omega_R \vdash C}\ \mathsf{cut}_A$$

$$\frac{\Omega_1 \vdash A \quad \Omega_2 \vdash B}{\Omega_1\ \Omega_2 \vdash A \bullet B}\ \bullet R \qquad \frac{\Omega_1\ A\ B\ \Omega_2 \vdash C}{\Omega_1\ (A \bullet B)\ \Omega_2 \vdash C}\ \bullet L$$

$$\frac{A\ \Omega \vdash B}{\Omega \vdash A \setminus B}\ \setminus R \qquad \frac{\Omega_A \vdash A \quad \Omega_L\ B\ \Omega_R \vdash C}{\Omega_L\ \Omega_A\ (A \setminus B)\ \Omega_R \vdash C}\ \setminus L$$

$$\frac{\Omega\ A \vdash B}{\Omega \vdash B\ /\ A}\ / R \qquad \frac{\Omega_A \vdash A \quad \Omega_L\ B\ \Omega_R \vdash C}{\Omega_L\ (B\ /\ A)\ \Omega_A\ \Omega_R \vdash C}\ / L$$

$$\frac{\Omega \vdash A \quad \Omega \vdash B}{\Omega \vdash A \,\&\, B}\ \& R \qquad \frac{\Omega_L\ A\ \Omega_R \vdash C}{\Omega_L\ (A \,\&\, B)\ \Omega_R \vdash C}\ \& L_1 \qquad \frac{\Omega_L\ B\ \Omega_R \vdash C}{\Omega_L\ (A \,\&\, B)\ \Omega_R \vdash C}\ \& L_1$$

$$\frac{\Omega \vdash A}{\Omega \vdash A \oplus B}\ \oplus R_1 \qquad \frac{\Omega \vdash B}{\Omega \vdash A \oplus B}\ \oplus R_2 \qquad \frac{\Omega_L\ A\ \Omega_R \vdash C \quad \Omega_L\ B\ \Omega_R \vdash C}{\Omega_L\ (A \oplus B)\ \Omega_R \vdash C}\ \oplus L$$

$$\frac{}{\cdot \vdash \mathbf{1}}\ \mathbf{1}R \qquad \frac{\Omega_L\ \Omega_R \vdash C}{\Omega_L\ (\mathbf{1})\ \Omega_R \vdash C}\ \mathbf{1}L$$

$$\frac{}{\Omega \vdash \top}\ \top R \qquad \text{no } \top L \text{ rules}$$

$$\text{no } \mathbf{0}R \text{ rules} \qquad \frac{}{\Omega_L\ (\mathbf{0})\ \Omega_R \vdash C}\ \mathbf{0}L$$

Figure 1: Ordered Sequent Calculus

References

H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences, U.S.A.*, 20:584–590, 1934.

Michael Dummett. *The Logical Basis of Metaphysics*. Harvard University Press, Cambridge, Massachusetts, 1991. The William James Lectures, 1976.

Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

W. A. Howard. The formulae-as-types notion of construction. Unpublished note. An annotated version appeared in: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, 479–490, Academic Press (1980), 1969.

Joachim Lambek. The mathematics of sentence structure. *The American Mathematical Monthly*, 65(3):154–170, 1958.

Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. Notes for three lectures given in Siena, Italy. Published in *Nordic Journal of Philosophical Logic*, 1(1):11-60, 1996, April 1983. URL http://www.hf.uio.no/ifikk/forskning/publikasjoner/tidsskrifter/njpl/vol1no1/meaning.pdf.

Dag Prawitz. *Natural Deduction*. Almquist & Wiksell, Stockholm, 1965.

Alfred Tarski. The concept of truth in formalized languages. In John Corcoran and J. H. Woodger, editors, *Logic, Semantics, Metamathematics*, pages 152–278. Clarendon Press, Oxford, 1931. Translation of a paper from 1931.

Lecture Notes on Proof Terms

15-836: Substructural Logics
Frank Pfenning

Lecture 4
September 7, 2023

1 Introduction

In the last lecture we have seen the fundamental properties of cut and identity elimination. They guarantee the *harmony* of the right and left rules for the connectives provides us with a *proof-theoretic semantics*: the meaning of a proposition is given by its cut-free proofs. This is a valid semantic point of view since the left and right rules only decompose propositions into their constituents so we don't have to look "outside" for their meaning. To put it another way: the proof-theoretic semantics is compositional.

In intuitionistic logics, therefore, proofs are the primary carriers of meaning. We therefore should think of them as being "first-class", which is not usually the case in classical mathematics: proofs are carried out, of course, but the study of their *formal structure* is not so important. For example, you are unlikely to see a notation for mathematical proofs as objects.

Besides the fact that proofs fundamentally provide meaning to the propositions, they also have a central *computational role*. We will explore this in the next lecture. So we need notations so we can write out proofs, reason about them, execute them, etc. In this lecture we take a neutral point of view: all we want to do is to compactly record the structure of proofs in the form of terms. These terms should have enough information to unwind them into the two-dimensional proofs we are used to, and vice versa.

These desiderata don't change from structural to substructural logics, although the process of checking them may change substantially.

2 Annotating the Sequent

As in the last lecture, we will focus here on ordered logic but the approach itself is quite general. Our goal is to endow the inference rules with additional information

so they operate on sequents of the form

$$\Omega \vdash M : A$$

where M is a proof (term) of A. In order to write out proof terms we should be able to refer to particular antecedents in Ω. For example, if we have a rule $\backslash L$ and we have multiple antecedents for the form $A \setminus B$, which antecedent is the rule applied to? One solution is to *count*. For example, the rule might be applied to the fifth antecedent. This works to an extent in ordered logic because the order of antecedents never changes. However, it is complicated by the fact that antecedents are split in some rules. For example, in a concrete rule application

$$\frac{A_0\ A_1 \vdash B \quad A_2\ A_3 \vdash C}{A_0\ A_1\ A_2\ A_3 \vdash B \bullet C}\ \bullet R$$

the numbering of antecedents is simple in the conclusion, but in the second premise we suddenly start counting at 2 instead of 0. It is possible to account for that, but proofs are very difficult to read. Also, since antecedents in *linear* logic are subject to exchange, the numbering might change in complicated ways, as in

$$\frac{A_0\ A_2 \vdash B \quad A_1\ A_3 \vdash C}{A_0\ A_1\ A_2\ A_3 \vdash B \otimes C}\ \otimes R$$

Again, this can be dealt with, but there is a more abstract alternative. We label all antecedents with distinct variables that we can refer to in proof terms. A sequent then has the form

$$(x_1 : A_1)\ \ldots\ (x_n : A_n) \vdash M : A$$

where all the x_i are distinct and may be mentioned in M. We show the proof terms (but not the variables in the antecedents) in blue.

We start with $A \setminus B$. Starting out, here is rule we want to annotate on the left, and a partial annotation on the right. The Ω's now stand for antecedents annotated with variables.

$$\frac{A\ \Omega \vdash B}{\Omega \vdash A \setminus B}\ \backslash R \qquad \frac{(x : A)\ \Omega \vdash ? : B}{\Omega \vdash ? : A \setminus B}\ \backslash R$$

A first thing we can say is that x must be chosen so it is fresh and doesn't occur already in Ω. This is so pervasive that it may often not be explicitly stated, relying on the presupposition that all variables declared in the antecedent are distinct.

Continuing, we see that somehow there will be proof $M : B$ once we annotate the premise. We then just need to fill in the slot in the conclusion with a term using it. We uniformly use the name of the rule as a proof constructor.

$$\frac{(x : A)\ \Omega \vdash M : B}{\Omega \vdash ? : A \setminus B}\ \backslash R \qquad \frac{(x : A)\ \Omega \vdash M : B}{\Omega \vdash (\backslash R\ (x.\ M)) : A \setminus B}\ \backslash R$$

We also need to indicate the variable x and track, somehow, that it will be fresh in the premise. We use the notation $(x.\, M)$ for a *bound* occurrence of x with scope M. Because the concrete names of bound variables do not matter, we can always silently rename it in case the particular name x is already among the antecedents. Many rules will take advantage of this notation and convention.

What about the left rule? It is applied to a particular antecedent, so this needs to be explicit.

$$\frac{\Omega_A \vdash A \quad \Omega_L\ B\ \Omega_R \vdash C}{\Omega_L\ \Omega_A\ (A \setminus B)\ \Omega_R \vdash C}\ \setminus L \qquad \frac{\Omega_A \vdash M : A \quad \Omega_L\ (y : B)\ \Omega_R \vdash P : C}{\Omega_L\ \Omega_A\ (x : A \setminus B)\ \Omega_R \vdash (\setminus L\ x\ ?\ ?) : C}\ \setminus L$$

We have already filled in the proof terms for the two premises, and also a name (y) for the antecedent B in the second premise. Now M is carried down without change, because all variables in Ω_A already exists in the conclusion, but we need to abstract P over y because it must be fresh.

$$\frac{\Omega_A \vdash M : A \quad \Omega_L\ (y : B)\ \Omega_R \vdash P : C}{\Omega_L\ \Omega_A\ (x : A \setminus B)\ \Omega_R \vdash (\setminus L\ x\ M\ (y.\, P)) : C}\ \setminus L$$

The rules for right implication can be developed entirely analogously.

$$\frac{\Omega\ A \vdash M : B}{\Omega \vdash (/R\ (x.\, M)) : B\ /\ A}\ /R \qquad \frac{\Omega_A \vdash M : A \quad \Omega_L\ (y : B)\ \Omega_R \vdash P : C}{\Omega_L\ (x : B\ /\ A)\ \Omega_A\ \Omega_R \vdash (/L\ x\ M\ (y.\, P)) : C}\ /L$$

At this point we are almost ready for an example, except for the identity. In certain places, like the first argument to left rules, only variables x are allowed. For the succedent we have an arbitrary term M. This means we could either include variables as a special case of a term, or we could use an explicit term construction like Id. We use the latter approach, so that every inference rules is turned into a corresponding constructor without exception.

$$\frac{}{(x : A) \vdash (\mathsf{Id}\ x) : A}\ \mathsf{id}$$

As an example, let's look at Lambek's associativity law from last lecture, using identity as a full rule.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\ \mathsf{id} \quad \dfrac{\dfrac{}{B \vdash B}\ \mathsf{id} \quad \dfrac{}{C \vdash C}\ \mathsf{id}}{(C\ /\ B)\ B \vdash C}\ /L}{A\ (A \setminus (C\ /\ B))\ B \vdash C}\ \setminus L}{(A \setminus (C\ /\ B))\ B \vdash A \setminus C}\ \setminus R}{A \setminus (C\ /\ B) \vdash (A \setminus C)\ /\ B}\ /R$$

To annotate this proof with a proof term, we start bottom up, labeling the antecedent with a variable and writing question marks where we have not yet filled in the information.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{? : A \vdash ? : A}\,\mathsf{id} \quad \dfrac{\dfrac{}{(? : B) \vdash ? : B}\,\mathsf{id} \quad \dfrac{}{(? : C) \vdash ? : C}\,\mathsf{id}}{(? : C \,/\, B)\ (? : B) \vdash ? : C}\,/L}{(? : A)\ (x : A \setminus (C \,/\, B))\ (? : B) \vdash ? : C}\,\setminus L}{(x : A \setminus (C \,/\, B))\ (? : B) \vdash ? : A \setminus C}\,\setminus R}{x : A \setminus (C \,/\, B) \vdash ? : (A \setminus C) \,/\, B}\,/R$$

The $/R$ rule introduces a new variable. In order to keep things straight, let's give it the name b.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{? : A \vdash ? : A}\,\mathsf{id} \quad \dfrac{\dfrac{}{(b : B) \vdash ? : B}\,\mathsf{id} \quad \dfrac{}{(? : C) \vdash ? : C}\,\mathsf{id}}{(? : C \,/\, B)\ (b : B) \vdash ? : C}\,/L}{(? : A)\ (x : A \setminus (C \,/\, B))\ (b : B) \vdash ? : C}\,\setminus L}{(x : A \setminus (C \,/\, B))\ (b : B) \vdash ? : A \setminus C}\,\setminus R}{x : A \setminus (C \,/\, B) \vdash ? : (A \setminus C) \,/\, B}\,/R$$

Actually, we now have some information about the proof term in the conclusion, but let's hold off filling that in until we have propagated more information upward. The second inference for $\setminus R$ works in a symmetric way. We call the new variable a.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{a : A \vdash ? : A}\,\mathsf{id} \quad \dfrac{\dfrac{}{(b : B) \vdash ? : B}\,\mathsf{id} \quad \dfrac{}{(? : C) \vdash ? : C}\,\mathsf{id}}{(? : C \,/\, B)\ (b : B) \vdash ? : C}\,/L}{(a : A)\ (x : A \setminus (C \,/\, B))\ (b : B) \vdash ? : C}\,\setminus L}{(x : A \setminus (C \,/\, B))\ (b : B) \vdash ? : A \setminus C}\,\setminus R}{x : A \setminus (C \,/\, B) \vdash ? : (A \setminus C) \,/\, B}\,/R$$

Now the $\setminus L$ rule introduces a new variable, as does the following $/L$. We write those in, naming sure to choose fresh names.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{a : A \vdash ? : A}\,\mathsf{id} \quad \dfrac{\dfrac{}{(b : B) \vdash ? : B}\,\mathsf{id} \quad \dfrac{}{(c : C) \vdash ? : C}\,\mathsf{id}}{(z : C \,/\, B)\ (b : B) \vdash ? : C}\,/L}{(a : A)\ (x : A \setminus (C \,/\, B))\ (b : B) \vdash ? : C}\,\setminus L}{(x : A \setminus (C \,/\, B))\ (b : B) \vdash ? : A \setminus C}\,\setminus R}{x : A \setminus (C \,/\, B) \vdash ? : (A \setminus C) \,/\, B}\,/R$$

Now that we have named all antecedents in all sequents, we can fill in the proof terms according to our proof term language, starting at the top and moving towards the bottom. We combine these into two steps, starting with the identities.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{a : A \vdash (\mathsf{Id}\ a) : A}\ \mathsf{id} \quad \dfrac{\dfrac{}{(b : B) \vdash (\mathsf{Id}\ b) : B}\ \mathsf{id} \quad \dfrac{}{(c : C) \vdash (\mathsf{Id}\ c) : C}\ \mathsf{id}}{(z : C \,/\, B)\ (b : B) \vdash ? : C}\ /L}{(a : A)\ (x : A \setminus (C \,/\, B))\ (b : B) \vdash ? : C}\ \setminus L}{(x : A \setminus (C \,/\, B))\ (b : B) \vdash ? : A \setminus C}\ \setminus R}{x : A \setminus (C \,/\, B) \vdash ? : (A \setminus C) \,/\, B}\ /R$$

And then we complete the terms for the remaining rule applications, working downwards.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{a : A \vdash (\mathsf{Id}\ a) : A}\ \mathsf{id} \quad \dfrac{\dfrac{}{(b : B) \vdash (\mathsf{Id}\ b) : B}\ \mathsf{id} \quad \dfrac{}{(c : C) \vdash (\mathsf{Id}\ c) : C}\ \mathsf{id}}{(z : C \,/\, B)\ (b : B) \vdash (/L\ z\ (\mathsf{Id}\ b)\ (c.\,\mathsf{Id}\ c)) : C}\ /L}{(a : A)\ (x : A \setminus (C \,/\, B))\ (b : B) \vdash (\setminus L\ x\ (\mathsf{Id}\ a)\ (/L\ z\ (\mathsf{Id}\ b)\ (c.\,\mathsf{Id}\ c))) : C}\ \setminus L}{(x : A \setminus (C \,/\, B))\ (b : B) \vdash (\setminus R\ (a.\setminus L\ x\ (\mathsf{Id}\ a)\ (/L\ z\ (\mathsf{Id}\ b)\ (c.\,\mathsf{Id}\ c)))) : A \setminus C}\ \setminus R}{x : A \setminus (C \,/\, B) \vdash (/R\ (b.\setminus R\ (a.\setminus L\ x\ (\mathsf{Id}\ a)\ (/L\ z\ (\mathsf{Id}\ b)\ (c.\,\mathsf{Id}\ c))))) : (A \setminus C) \,/\, B}\ /R$$

There is a lot of redundant information in this derivation. In fact, starting with

$$x : A \setminus (C \,/\, B) \vdash (/R\ (b.\setminus R\ (a.\setminus L\ x\ (\mathsf{Id}\ a)\ (/L\ z\ (\mathsf{Id}\ b)\ (c.\,\mathsf{Id}\ c))))) : (A \setminus C) \,/\, B$$

we can reconstruct the whole derivation in a unique way.

This correspondence can be stated formally as two theorems (for antecedents Ω that are labelled with unique variables).

(i) Given Ω, M, and A, either there is a unique derivation

$$\begin{array}{c}\mathcal{D}\\ \Omega \vdash M : A\end{array}$$

or there is no such derivation.

(ii) Given a derivation $\Omega \vdash A$ where the applications of left rules are marked with their corresponding variable, then there is a unique term M such that $\Omega \vdash M : A$.

The process s of constructing a derivation from a term is not entirely straightforward because of the necessary splits of the hypotheses in rules with multiple premises. We could either look ahead to see which variables occur, or we can propagate all antecedents to one of the premises and then pass on the ones that were

not used to the other premises. In this algorithm, we need to make sure that uses of variables in a derivation are consecutive so that order is suitably respected.

A general algorithm for the input/output interpretation of antecedents that works for structural, linear, and ordered antecedents and even a mix is described by Polakow [2000], with more proof details in his Ph.D. thesis [Polakow, 2001]. This system work *during proof search* when a full proof term isn't even available for checking, so it solves a somewhat more difficult problem than is needed here. Still, it provides an elegant algorithmic solution.

We show one more example, for $A \bullet B$. There is no variable binding in the right rule.

$$\frac{\Omega_1 \vdash A \quad \Omega_2 \vdash B}{\Omega_1\ \Omega_2 \vdash A \bullet B}\ \bullet R \qquad \frac{\Omega_1 \vdash M : A \quad \Omega_2 \vdash N : B}{\Omega_1\ \Omega_2 \vdash (\bullet R\ M\ N) : A \bullet B}\ \bullet R$$

The left rule is a kind of pattern matching and therefore has to bind *two* fresh variables.

$$\frac{\Omega_L\ A\ B\ \Omega_R \vdash C}{\Omega_L\ (A \bullet B)\ \Omega_R \vdash C}\ \bullet L \qquad \frac{\Omega_L\ (y : A)\ (z : B)\ \Omega_R \vdash P : C}{\Omega_L\ (x : A \bullet B)\ \Omega_R \vdash (\bullet L\ x\ (y.\, z.\, P)) : C}\ \bullet L$$

Since the proof terms are constructed quite systematically, we don't show the remaining rules. The language of proof terms is summarized in Figure 1.

3 Cut Reductions on Proof Terms

We an express the cut reductions between two proofs on the proof terms themselves. We show only one example (the principal reduction for fuse), but others are similar. We first introduce a proof term for cut, taking it here as a given rules of inference rather than just admissible.

$$\frac{\Omega \vdash M : A \quad \Omega_L\ (x : A)\ \Omega_R \vdash P : C}{\Omega_L\ \Omega\ \Omega_R \vdash (\mathsf{Cut}_A\ M\ (x.\, P)) : C}\ \mathsf{cut}_A$$

Now to the particular case. Before the reduction, we have

$$\frac{\dfrac{\Omega_1 \vdash M : A \quad \Omega_2 \vdash N : B}{\Omega_1\ \Omega_2 \vdash (\bullet R\ M\ N) : A \bullet B}\ \bullet R \qquad \dfrac{\Omega_L\ (y : A)\ (z : B)\ \Omega_R \vdash P : C}{\Omega_L\ (x : A \bullet B)\ \Omega_R \vdash (\bullet L\ x\ (y.\, z.\, P)) : C}\ \bullet L}{\Omega_L\ \Omega_1\ \Omega_2\ \Omega_R \vdash (\mathsf{Cut}_{A \bullet B}\ (\bullet R\ M\ N)\ (x.\, \bullet L\ x\ (y.\, z.\, P))) : C}\ \mathsf{cut}_{A \bullet B}$$

and after the reduction (writing in proof terms afresh):

$$\longrightarrow_R \quad \frac{\Omega_2 \vdash N : B \qquad \dfrac{\Omega_1 \vdash M : A \quad \Omega_L\ (y : A)\ (z : B)\ \Omega_R \vdash P : C}{\Omega_L\ \Omega_1\ (z : B)\ \Omega_R \vdash (\mathsf{Cut}_A\ M\ (y.\, P)) : C}\ \mathsf{cut}_A}{\Omega_L\ \Omega_1\ \Omega_2\ \Omega_R \vdash (\mathsf{Cut}_B\ N\ (z.\, \mathsf{Cut}_A\ M\ (y.\, P))) : C}\ \mathsf{cut}_B$$

Expressing this purely on the proof term, we can recognize it as a kind of pattern matching reduction, except that we don't substitute the way we would usually think of it in the definition of functional languages.

$$\mathsf{Cut}_{A \bullet B}\ (\bullet R\ M\ N)\ (\bullet L\ x\ (y.\, z.\, P)) \longrightarrow_R \mathsf{Cut}_B\ N\ (z.\, \mathsf{Cut}_A\ M\ (y.\, P))$$

In the next lecture similar cut reductions play a much more significant role because we will make the computational intuition more precise.

4 Invertibility and Polarity

When constructing proofs, bottom-up, a priori we have many possible choices. Any left rule might apply to any matching antecedent, or a right rule to a matching succedent. Applying a rule is a small step, breaking down just one connective. Then we are again faced with a similar choice. Reducing this nondeterminism is critical in proof search procedures, although it may not mean much regarding the question of decidability.

For example, it is easy to see that the pure ordered logic we have seen is decidable once we know cut elimination, because the premises of all the rules are smaller than the conclusion in the sense of having fewer connectives in them. Therefore, any way we can try to construct a proof, bottom-up, will have to terminate, either in success or in failure. If we try all of them, we will either find a proof or there cannot be any.

Fortunately, we don't need to search that blindly while remaining complete. For each connective, either the left rule or the right rule in the sequent calculus is *invertible* in the sense that the premises are provable if and only if the conclusion is. So we can use such a rule, bottom-up, without having to consider any other choices because we have preserved provability exactly.

The question is which rules are invertible. There is an easy test: whichever rule is applied first (again, reading bottom-up) in the identity expansion is the invertible rule while the counterpart on the other side is not. Here is a tiny example:

$$\dfrac{\dfrac{\dfrac{}{A \vdash A}\ \mathsf{id}_A \quad \dfrac{}{B \vdash B}\ \mathsf{id}_B}{A\ B \vdash A \bullet B}\ \bullet R}{A \bullet B \vdash A \bullet B}\ \bullet L$$

While it is now plausible that $\bullet L$ rule is invertible, we can see that $\bullet R$ is not because we cannot (yet) break up the antecedents appropriately.

We can *prove* invertibility of $\bullet L$ in pure ordered logic using the admissibility of cut and identity. You may want to try this yourself before peeking at the solution on the next page.

$$\dfrac{\dfrac{\cdots\cdots\ \mathsf{id}_A \quad \cdots\cdots\ \mathsf{id}_B}{A \vdash A \qquad B \vdash B}\ \bullet R \qquad \Omega_L\ (A \bullet B)\ \Omega_R \vdash C}{\Omega_L\ A\ B\ \Omega_R \vdash C}\ \mathsf{cut}_{A \bullet B}$$

You can see if you read this from the unproved premise to the conclusion, it is just the inverted $\bullet L$. Given any concrete proof of the second premise we can apply cut elimination to obtain a direct proof of the conclusion.

But we have to be a bit careful that this notion of *invertibility* may not completely coincide with the inference rule being reversible. For example, the rule

$$\frac{}{\cdot \vdash \mathbf{1}}\ \mathbf{1}R$$

is technically invertible in the sense that whenever the conclusion is, so are all the premises (namely none). However, we cannot always apply this rule when we see $\mathbf{1}$ in a succedent because the antecedents may not be empty. If we had formulated the rule slightly differently:

$$\frac{\Delta = (\cdot)}{\Delta \vdash \mathbf{1}}\ \mathbf{1}R$$

then it would not no longer be invertible.

Therefore, instead of talking about the right or left invertibility of a *rule*, we talk about the *right or left invertibility of a connective*. If we can *always* apply its right or left rule without losing provability when a connective appears at the top level of a proposition, we call the connective invertible on the right or on the left, respectively.

The distinction of whether left or right rules are invertible is of fundamental importance in studying proof theory, and its connection to computation. We call the right invertible connectives *negative*, while left invertible connectives are *positive*. For ordered logic, we get the following classification:

Negative (right invertible): $A \setminus B$, $A \,/\, B$, $A \mathbin{\&} B$, $\top$

Positive (left invertible): $A \bullet B$, $A \circ B$, $\mathbf{1}$, $A \oplus B$, $\mathbf{0}$

5 A Zoo of Connectives

In linear and ordered logic, the polarity of each connective is uniquely determined. Somewhat surprisingly, though, in structural logic conjunction has both invertible left and right rules. This is because it actually unifies two different connectives we know from linear logic: truth in the same state $A \otimes B$ (positive) and external choice $A \& B$ (negative). It turns out that it is highly beneficial to make this distinction even

for intuitionistic logic, but this is rarely done—its significance was not recognized until the discovery of call-by-push-value (really: a polarized type system) [Levy, 2006]. We will see that positive types (including positive pairs) are *eager* while negative types (including externa choice) are *lazy*.

Below is the table of connective in the various logics, where we see that certain connectives further on the right becomes indistinguishable when we move to the left. For example, if the order of antecedents is irrelevant (e.g., in linear logic) then left and right implication ($A \backslash B$ and B / A) become indistinguishable and are written as $A \multimap B$.

structural	linear	ordered	polarity	pronunciation
$A \supset B$	$A \multimap B$	$A \setminus B$	negative	A under B
		$B \,/\, A$	negative	B over A
$A \wedge B$	$A \otimes B$	$A \bullet B$	positive	A fuse B
		$A \circ B$	positive	A twist B
	$A \mathbin{\&} B$	$A \mathbin{\&} B$	negative	A with B
$A \vee B$	$A \oplus B$	$A \oplus B$	positive	A plus B
$\top$	1	1	positive	one
	$\top$	$\top$	negative	top
$\bot$	0	0	positive	zero

The ambiguous nature of general structural conjunction $A \wedge B$ and $\top$ is resolved at the linear level because these connectives split into two each: one positive and one negative.

6 Summary

The language of proof terms is in Figure 1. Since the constructors are named after the inference rules we don't bother showing the inference rules. You should be able to easily write them out.

Valid proof terms are in one-to-one correspondence with proofs, so they merely serve as a compact notation here. We can then express operations such as cut reduction on these terms, rather than showing complex derivations.

If we think of cut and identity as being admissible, then Id_A (at types other than atoms P) and Cut_A would be meta-level operations to compute a cut-free proof from the arguments. But we need to keep in mind that cut reduction is highly nondeterministic, so perhaps $\mathsf{Cut}_A\ M\ (x.\, P) \sim N$ it is best thought of a 4-place relation between A, M, $x.\, P$, and N (all of the proofs being cut-free).

$$
\begin{array}{rcll}
M, N, P & ::= & \mathsf{Cut}_A\ M\ (x.\, P) & \mid\ \mathsf{Id}\ x \\
& \mid & \backslash R\ (x.\, M) & \mid\ \backslash L\ x\ M\ (y.\, P) \\
& \mid & /R\ (x.\, M) & \mid\ /L\ x\ M\ (y.\, P) \\
& \mid & \bullet R\ M\ N & \mid\ \bullet L\ x\ (y.\, z.\, P) \\
& \mid & \circ R\ M\ N & \mid\ \circ L\ x\ (z.\, y.\, P) \\
& \mid & \mathbin{\&} R\ M\ N & \mid\ \mathbin{\&} L_1\ x\ (y.\, P) \mid \mathbin{\&} L_2\ x\ (z.\, P) \\
& \mid & \mathbf{1} R & \mid\ \mathbf{1} L\ x\ M \\
& \mid & \oplus R_1\ M \mid \oplus R_2\ N & \mid\ \oplus L\ x\ (y.\, N)\ (z.\, P) \\
& \mid & \top R & \mid \\
& \mid & & \mid\ \mathbf{0} L\ x
\end{array}
$$

Figure 1: Proof terms for ordered logic

References

Paul Blain Levy. Call-by-push-value: Decomposing call-by-value and call-by-name. *Higher-Order and Symbolic Computation*, 19(4):377–414, 2006.

Jeff Polakow. Linear logic programming with an ordered context. In M. Gabbrielli and F. Pfenning, editors, *Conference on Principles and Practice of Declarative Programming (PPDP 2000)*, pages 68–79, Montreal, Canada, September 2000. ACM.

Jeff Polakow. *Ordered Linear Logic and Applications*. PhD thesis, Department of Computer Science, Carnegie Mellon University, August 2001.

Lecture Notes on
Linear Message Passing I

15-836: Substructural Logics
Frank Pfenning

Lecture 5
September 12, 2023

1 Introduction

In this lecture we start a new section of the course. We have studied proof systems for substructural logics and their properties, such as cut and identity elimination. We have also seen that substructural inference itself can express certain algorithms (e.g., for parsing) at a high level of abstraction. We can summarize this with the slogan "*computation is proof construction*". The final answer is a proof, or sometimes just the information of whether a proof exists or not.

Now we look at a connection where proofs themselves are programs, and computation proceeds by *proof reduction* rather than *proof construction*. The new slogan is "*computation is proof reduction*". This notion of computation inherits many desirable properties from logic and proof theory, but it is certainly not without its own set of challenges and difficulties. We will come back to these challenges in Lecture 7, once we have developed an intuitive understanding of the relationship.

Here is the basic table of correspondences:

Logic	**Programming**
Proposition	Type
Proof	Program
Reduction	Computation

The specifics of the correspondence are dramatically dependent on the following variables (and maybe more):

- **The logic.** Ordered logic is different from linear logic, which is be different from structural logic, and several of these come in intuitionistic as well as classical versions. Other examples are temporal logics, modal logics, epistemic logics, and so on, each with their opportunity for computational meaning.

- **The proof system.** Since proofs are programs, the specifics of each proof system determine the structure of programs. And different proof systems have different notion of reduction, which induce different forms of computation.

Such variations are not trivial, but fundamentally change the way we think about programs and their computation. For example, early work by Curry [1934] essentially assigned computational meaning to axiomatic proofs in Hilbert-style systems. Such computation is in the form of combinatory reduction which can be seen at the root of the APL programming language. Later work by Howard [1969] established a relationship between Gentzen's system of natural deduction [Gentzen, 1935, Prawitz, 1965] and Church's typed λ-calculus [Church, 1940]. Here, computation proceeds by substitution which is at the root of modern functional programming languages.

In today's lecture, we begin to establish a connection between linear logic [Girard, 1987, Girard and Lafont, 1987] presented as a sequent calculus, and message-passing processes. The propositions of linear logic express communication protocols, giving a post-hoc logical justification for *session types* [Honda and Tokoro, 1991, Honda, 1993, Honda et al., 1998]. The connection in this form was first established by Caires and Pfenning [2010] and followed up in various ways (e.g., [Wadler, 2012, Caires et al., 2016]).

But enough of the generalities. Let's get started! Instead of ordered logic which has been mostly our focus so far, we now move to linear logic because of the wider variety of programs it supports.

2 Cut as Process Composition

The first two fundamental ideas are the following:

- Proofs represent processes.
- Cut corresponds to the parallel composition of two processes with a private communication channel connecting them.

In order to see how processes are connected, exactly, we label antecedents as we did in Lecture 4. This removes ambiguity, for example, when we have more than one antecedent with a particular proposition A. In addition, we also label the *succedent* with a channel in order to clarify which of the antecedents in the other premise of a cut it is connected to. Without an explicit proof term, the sequent then would have the form

$$\underbrace{x_1 : A_1, \ldots, x_n : A_n}_{\text{channels used}} \quad \vdash \quad \underbrace{x : A}_{\text{channel provided}}$$

A process provides exactly one channel and may use multiple channels. All the variables x_i and x must be distinct.

We need cut as a primitive rule now because cut reduction induces computation. Without cut, there will be no computation! We call the process P (= the proof of the first premise) the *provider* (or server) and the the process Q (= the proof of the second premise) the *client*.

$$\frac{\begin{array}{cc} P & Q \\ \Delta \vdash (x : A) & \Delta', x : A \vdash (z : C) \end{array}}{\Delta, \Delta' \vdash (z : C)}\ \mathsf{cut}$$

The variable x represents the channel of communication between provider and client. This channel is *private* in the sense that P and Q are the only two endpoints, which is guaranteed by our convention about the uniqueness of variable names.

Because the variables in a sequent represent channels, we often just refer to them as channels, just like we might say "the integer x" when x is a variable standing for an integer.

3 Cut Reduction as Communication

The next question is how cut reduction corresponds to communication. We have seen that there are three different kinds of cut reduction:

1. *principal reductions*, where a right rule for a connective meets a corresponding left rule;
2. *identity reductions*, where one of the premises is an identity rule; and
3. *permuting reductions*, where an inference is applied to an antecedent or succedent not involved in the cut.

It turns out that only the first two are of interest *computationally* while the third represents a form of equality reasoning between processes.

We start with principal reductions, using *internal choice* $A \oplus B$ as a guiding example. There are two—we show the first one, since the second one is entirely symmetric.

$$\frac{\left.\begin{array}{cc} \dfrac{\begin{array}{c} P_1 \\ \Delta \vdash x : A \end{array}}{\Delta \vdash x : A \oplus B}\ \oplus R_1 & \dfrac{\begin{array}{cc} Q_1 & Q_2 \\ \Delta', x : A \vdash z : C & \Delta', x : B \vdash z : C \end{array}}{\Delta', x : A \oplus B \vdash z : C}\ \oplus L \end{array}\right\} = Q}{\Delta, \Delta' \vdash z : C}\ \mathsf{cut}_{A \oplus B}$$

$$\longrightarrow_R \quad \frac{\begin{array}{cc} P_1 & Q_1 \\ \Delta \vdash x : A & \Delta', x : A \vdash z : C \end{array}}{\Delta, \Delta' \vdash z : C}\ \mathsf{cut}_A$$

There are some syntactic details to consider, but the first and most important question is "*What is the flow of information here between the first premise (process P) and the second premise (process Q)?*" We see that Q, with the $\oplus L$ rule is prepared for both eventualities: either A might be true or B might be true. This choice is made by P which ends in either $\oplus R_1$ (A is true) or $\oplus R_2$ (B is true). Therefore, P has to communicate this information to Q.

We say that P either sends π_1 or π_2, and Q is set to receive and branch on either of those two tokens. So we write

$$\begin{array}{lcl} P & = & \mathbf{send}\ x\ \pi_1\ ;\ P_1 \\ Q & = & \mathbf{recv}\ x\ (\pi_1 \Rightarrow Q_1 \mid \pi_2 \Rightarrow Q_2) \end{array}$$

where P could also send π_2. If we write the process P into the judgment in the form

$$\Delta \vdash P :: (x : A)$$

then we get the following three rules

$$\frac{\Delta \vdash P_1 :: (x : A)}{\Delta \vdash \mathbf{send}\ x\ \pi_1\ ;\ P_1 :: (x : A \oplus B)}\ \oplus R_1 \qquad \frac{\Delta \vdash P_2 :: (x : B)}{\Delta \vdash \mathbf{send}\ x\ \pi_2\ ;\ P_2 :: (x : A \oplus B)}\ \oplus R_2$$

$$\frac{\Delta, x : A \vdash Q_1 :: (z : C) \quad \Delta, x : B \vdash Q_2 :: (z : C)}{\Delta, x : A \oplus B \vdash \mathbf{recv}\ x\ (\pi_1 \Rightarrow Q_1 \mid \pi_2 \Rightarrow Q_2) :: (z : C)}\ \oplus L$$

These rules are actually closely related to the rules for proof terms from the last lecture, except that our purpose and therefore notation are quite different. For one, we have labeled the succedent with a variable that represent a communication channel. For another, we have used the terms *send* and *receive* to capture the communication action.

We now prefer to read these rules as *typing rules* for the processes P and Q, but we should keep in mind that erasing all the process information turns them back into the familiar logical rules.

We can now go back to the cut reduction and annotate each sequent with its process term. Writing the cut with x as a private channel as $P \parallel_x Q$, we can read off the reduction on processes from the reduction on proofs.

$$\begin{array}{lcl} (\mathbf{send}\ x\ \pi_1\ ;\ P_1) \parallel_x (\mathbf{recv}\ x\ (\pi_1 \Rightarrow Q_1 \mid \pi_2 \Rightarrow Q_2)) & \longrightarrow_R & P_1 \parallel_x Q_1 \\ (\mathbf{send}\ x\ \pi_2\ ;\ P_2) \parallel_x (\mathbf{recv}\ x\ (\pi_1 \Rightarrow Q_1 \mid \pi_2 \Rightarrow Q_2)) & \longrightarrow_R & P_2 \parallel_x Q_2 \end{array}$$

An interesting observation here is that the type of the channel x *evolves* from $A \oplus B$ to either A or B, depending on whether the message was π_1 or π_2. In most programming languages the type of a variable never changes, but here this seems essential. We also see that the "outside" channels (the ones in the conclusion of the cut) which we wrote as Δ, Δ' and $z : C$ do not change their type. This will be important in Lecture 7 when we investigate the properties of the programming language as distinct from the properties of the proof system.

4 Communication and Polarity

With the example of internal choice $A \oplus B$, we have seen that the type prescribes a *communication protocol*: the provider sends either π_1 or π_2 and the client must be prepared to receive either one. Before we look at other connectives, can we predict whether the provider or the client will have information to send? A key idea is that the rule for an invertible connective does not have any information. After all, the premises can be derived if and only if the conclusion can. On the other hand, noninvertible connectives are noninvertible precisely because applying them requires a choice. The information contained in this choice (like π_1 and π_2) is then communicated to the connected process.

Recall that all *positive* connectives are noninvertible on the right. Therefore, taking the provider's perspective, the right rules for positive connectives will *send* a message, while their left rules will *receive* a message. In linear logic, these are $A \oplus B$, $\mathbf{1}$, and $A \otimes B$. Looking at the unit, we have

$$\frac{}{\cdot \vdash \mathbf{1}}\ \mathbf{1}R \qquad \frac{\Delta \vdash C}{\Delta, \mathbf{1} \vdash C}\ \mathbf{1}L$$

$\mathbf{1}$ is not a right invertible connective, because the rule cannot be applied to a section $\Delta \vdash 1$ unless Δ is empty.

$$\frac{\dfrac{}{\cdot \vdash (x : \mathbf{1})}\ \mathbf{1}R \quad \dfrac{\overset{Q}{\Delta' \vdash C}}{\Delta', x : \mathbf{1} \vdash C}\ \mathbf{1}L}{\Delta' \vdash (z : C)}\ \mathsf{cut} \qquad \longrightarrow_R \qquad \overset{Q}{\Delta' \vdash C}$$

The information conveyed, therefore, is only that the associated process terminates, which is done by sending the unit message $()$. First, the typing rules and then the reduction rule.

$$\frac{}{\cdot \vdash \mathbf{send}\ x\ ()\ ::\ (x : \mathbf{1})}\ \mathbf{1}R \qquad \frac{\Delta \vdash Q :: (z : C)}{\Delta, x : \mathbf{1} \vdash \mathbf{recv}\ x\ (() \Rightarrow Q)\ ::\ (z : C)}\ \mathbf{1}L$$

$$\mathbf{send}\ x\ ()\ \|_x\ \mathbf{recv}\ x\ (() \Rightarrow Q) \quad \longrightarrow_R \quad Q$$

Before we move on to the meaning of $A \otimes B$ and the other linear connectives, we program some small examples that are already expressible with just $A \oplus B$ and 1.

5 An Example: Booleans

A very simple type is that of the Booleans.

$$\mathsf{bool} \quad = \quad \mathbf{1} \oplus \mathbf{1}$$

Perhaps not coincidentally, $1 + 1 = 2$ is also the number of different message sequences that can be communicated on a channel $x : \mathsf{bool}$. Namely:

$$\cdot \vdash \mathit{false} :: (c : \mathsf{bool})$$
$$\cdot \vdash \mathit{true} :: (c : \mathsf{bool})$$

$$\mathit{false} = \mathbf{send}\ c\ \pi_1\ ;\ \mathbf{send}\ c\ (\,)$$
$$\mathit{true} = \mathbf{send}\ c\ \pi_2\ ;\ \mathbf{send}\ c\ (\,)$$

It is possible for a process P with the typing $\cdot \vdash P :: (c : \mathsf{bool})$ to spawn other processes and perform a lot of computation, but ultimately it can only send $\pi_1, (\,)$ or $\pi_2, (\,)$ along c, because that is what the type of the channel enforces. It might also fail to terminate if the language permits recursion, which we come to in the next section.

Before that, let's consider a process that receives a Boolean and passes on the negation.

$$a : \mathsf{bool} \vdash \mathit{neg} :: (c : \mathsf{bool})$$

$$\begin{aligned}\mathit{neg} = \mathbf{recv}\ a\ (\,&\pi_1 \Rightarrow \mathbf{recv}\ a\ (() \Rightarrow \mathbf{send}\ c\ \pi_2\ ;\ \mathbf{send}\ c\ (\,))\\ &\mid \pi_2 \Rightarrow \mathbf{recv}\ a\ (() \Rightarrow \mathbf{send}\ c\ p_1\ ;\ \mathbf{send}\ c\ (\,))\,)\end{aligned}$$

From the purely logical perspective, this is uninteresting because this program represents a proof of

$$\mathbf{1} \oplus \mathbf{1} \vdash \mathbf{1} \oplus \mathbf{1}$$

There should be four cut-free and identity-free proofs of this proposition that represents the four unary Boolean functions.

Even though for the moment we have a perfect correspondence between proofs and programs, there is a shift in perspective. Proofs are primarily thought of as evidence for truth, while programs are primarily thought of as objects that compute. Through the correspondence each view influences the others, and we can see relationships and interpretations that may otherwise be missed or found insignificant.

6 Another Example: Natural Numbers

Natural numbers are a mainstay both in logic and programming languages. In logic, they are studied in Peano Arithmetic, in programming languages they are either primitive (perhaps with a bounded range) or thought of as an inductive type.

We will put off any investigation of induction and inductive types and instead go directly to *recursion*, both at the level of types and at the level of programs. Consider, for example, the type $\mathbf{1} \oplus \mathbf{1} \oplus \mathbf{1}$. This has three possible message sequences, $\mathbf{1} \oplus \mathbf{1} \oplus \mathbf{1} \oplus \mathbf{1}$ has four, and so on. There are infinitely many natural numbers, so the definition would be infinite:

$$\mathsf{nat} = \mathbf{1} \oplus (\mathbf{1} \oplus \ldots)$$

Using recursion, we can express this directly as

$$\mathsf{nat} = \mathbf{1} \oplus \mathsf{nat}$$

The corresponding message sequences can also be recursively defined:

$$\overline{n} = \pi_1\,(\,) \mid \pi_2\,\overline{n}$$

We see that the use of π_1 and π_2 is a bit awkward from the programming perspective, so we generalize the binary sum $A \oplus B$ to $\oplus\{\ell : A_\ell\}_{\ell \in L}$ where ℓ are *labels* (also called *tags*) and L is a *finite* index set. Then the binary sum can be defined with the index set $\{\pi_1, \pi_2\}$, maybe written as $A \oplus B \triangleq \oplus\{\pi_1 : A, \pi_2 : B\}$.

Then we define:

$$\mathsf{nat} = \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\}$$

$$\cdot \vdash \mathit{zero} :: (n : \mathsf{nat})$$
$$\mathit{zero} = \mathbf{send}\ n\ \mathsf{zero}\ ;\ \mathbf{send}\ n\ (\,)$$

In order to define the successor process, it is convenient to consider the computational interpretation of the identity. First, the cut reductions, which show that cut and identity "cancel" each other. The structure of A is irrelevant, because the cut is directly eliminated.

$$\dfrac{\dfrac{\overset{P(x)}{\Delta \vdash x : A} \quad \dfrac{}{x : A \vdash y : A}\,\mathsf{id}}{\Delta \vdash y : A}\,\mathsf{cut} \qquad \longrightarrow_R \qquad \overset{P(y)}{\Delta \vdash y : A}$$

$$\dfrac{\dfrac{}{y : A \vdash x : A}\,\mathsf{id} \quad \overset{Q(x)}{\Delta', x : A \vdash z : C}}{\Delta', y : A \vdash z : C}\,\mathsf{cut} \qquad \longrightarrow_R \qquad \overset{Q(y)}{\Delta', y : A \vdash z : C}$$

We see that cut reduction in these two cases performs a variable substitution or renaming ($P(x)$ becomes $P(y)$ and $Q(x)$ becomes $Q(y)$). This renaming is necessary so that the conclusion before and after the reduction remains the same. Computationally, this is necessary because the channels in the conclusion of the cut are connected to other processes (either clients or providers), and these other processes should be able continue to communicate along the same channels as before.

We refer to this operation as *forwarding* because the identity intuitively forwards any messages on the x and y channels to the other.

$$\dfrac{}{x : A \vdash \mathbf{fwd}\ y\ x :: (y : A)}\,\mathsf{id}$$

The provided channel y here comes first, which may seem unintuitive but is part of a number of coordinated decisions is the design of the MPASS programming

language. The reductions:

$$\begin{array}{lcl} P(x) \parallel_x \mathbf{fwd}\ y\ x & \longrightarrow_R & P(y) \\ \mathbf{fwd}\ x\ y \parallel_x Q(x) & \longrightarrow_R & Q(y) \end{array}$$

To make sure the forwarder is connected to the correct provider $P(x)$ or client $Q(x)$, the channel x must actually occur in these processes. Since communication channels are private and linear, this condition is sufficient to guarantee a correct reduction.

We can now complete the brief example of natural numbers by writing the successor process.

$$\mathsf{nat} = \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\}$$

$$\begin{array}{l} \cdot \vdash \mathit{zero} :: (n : \mathsf{nat}) \\ \mathit{zero} = \mathbf{send}\ n\ \mathsf{zero}\ ;\ \mathbf{send}\ n\ (\,) \end{array}$$

$$\begin{array}{l} m : \mathsf{nat} \vdash \mathit{succ} :: (n : \mathsf{nat}) \\ \mathit{succ} = \mathsf{send}\ n\ \mathsf{succ}\ ;\ \mathbf{fwd}\ n\ m \end{array}$$

In programming language parlance, types like nat are *equirecursive*, which means here that there is no message associated with the unfolding of the recursion. In the context of a language such as ML we would think of the type nat as *inductive* because we would like values of this type to be isomorphic to the usual natural numbers. In a non-strict language such as Haskell the type would instead be interpreted coinductively because we can write a simple program that produces an infinite stream of succ constructors. Similarly, in the context of our message-passing programming language, a recursive program could easily send an infinite stream of succ labels. So types in MPASS are interpreted coinductively. This means that we *disallow* type definition such as $\omega = \omega$: the right-hand side of a type definition must always start with a constructor so it uniquely represents a potentially infinite type (that is, a potentially infinite communication protocol) in a finitary way.

For example, the type

$$\mathsf{bits} = \oplus\{\mathsf{b0} : \mathsf{bits}, \mathsf{b1} : \mathsf{bits}\}$$

represents an infinite stream of bits 0 and 1. It is easy to write a transducer process that negates each bit as it comes in.

7 MPASS Syntax

We introduce the syntax of the MPASS language for the remaining examples of this lecture and the following two lectures. You can download a version of MPASS from the course resources page. This contains a `readme.txt` file with a full grammar

and other useful information. The examples from this lecture can be found in the file lecture5.mps.

We can define types recursively at the top level using the **type** keyword, including mutual recursion. Labels must be preceded by a single quote so they are syntactically distinguished from the names of types, channels, and processes.

```
% unary natural numbers
type nat = +{'zero : 1, 'succ : nat}
```

Processes are defined with the **proc** keyword, followed by the name of the process, then followed by the channel provided by the process and its type. Continuing the example:

```
proc zero (n : nat) = send n 'zero ; send n ()
```

This avoids the need for a separate type declaration. If the process additionally *uses* channels, they follow after the provided one.

```
proc succ (n : nat) (m : nat) = send n 'succ ; fwd n m
```

8 Example: Natural Numbers in Binary Form

We have already seen natural numbers in binary form as an example for ordered inference. Now we think of them in terms of message-passing like the natural numbers, where the least significant bit is sent first.

```
type bin = +{'b0 : bin, 'b1 : bin, 'e : 1}

proc zero (x : bin) = send x 'e ; send x ()
proc succ (x : bin) (y : bin) =
  recv y ( 'b0 => send x 'b1 ; fwd y x
         | 'b1 => send x 'b0 ; call succ x y   % carry
         | 'e => recv y (() => send x 'b1 ; send x 'e ; send x () ) )
```

We also see an example for the **call** keyword. Its first argument is a process (here a recursive call), the second is the provided channel, and the remaining ones are the used channels passed to the process. These arguments must match the type declarations in the process header.

To write and understand such a program it is often extremely valuable to calculate the type of every variable at various program points. That's because they change, and yet determine what may be possible as a next interaction.

The syntax for cut is, abstractly $x_A \leftarrow P(x)\ ;\ Q(x)$, where $P(x)$ *provides* x and $Q(x)$ *uses* x. The typing rule:

$$\frac{\Delta \vdash P(x) :: (x : A) \quad \Delta', x : A \vdash Q(x) :: (z : C)}{\Delta, \Delta' \vdash x_A \leftarrow P(x)\ ;\ Q(x) :: (z : C)}\ \mathsf{cut}$$

The type A indicates the type of the new private channel, which may not be readily inferable. We do not indicate how the antecedents of the conclusion are to be split between Δ and Δ'; instead this is determined by a type-checking algorithm.

Dynamically, cut creates a (globally fresh) channel a, spawns the process $P(a)$ and continues as $Q(a)$. So there is a small asymmetry here inherent in the nature of the (intuitionstic) sequent with a single conclusion.

$$x_A \leftarrow P(x)\ ;\ Q(x) \quad \longrightarrow \quad P(a) \parallel_a Q(a) \quad (a \text{ fresh})$$

Since cut just allocates a fresh channel and spawns a new process, there is no interprocess communication involved in its operational interpretation.

We can use this to test our small successor programs: we create a channel initialized to zero and then increment it several times. We show the current state of the typing judgment after a line of code. For example, the first call to `succ` will pass x_0 to it, so it is no longer in the current context.

```
proc test (x : bin) =
  x0 <- call zero x0 ;      % x0 : bin |- x : bin
  x1 <- call succ x1 x0 ;   % x1 : bin |- x : bin
  x2 <- call succ x2 x1 ;   % x2 : bin |- x : bin
  x3 <- call succ x3 x2 ;   % x3 : bin |- x : bin
  fwd x x3
```

We recognize an idiom here, where we allocate a fresh channel like x_1 and spawn a new named process providing it at the same time. When we use cut this way we can omit the type annotation for the new channel.

Because our language is concurrent, all these successor processes may be lined up in a pipeline, passing bits through. Because the process `test` provides a channel x but does not use any channels, we can execute this program with the **`exec`** keyword.

```
exec test
```

This will print back the channels that have been created and are externally observable, starting with the initial channel (0), and the message observed on each of the channels. Here, there is only one because the other ones are closed when the successor processes terminate.

```
% executing test
(0) -> b1.b1.e.()
```

So the sequence of messages on channel (0) is `b1`, followed by `b1`, followed by `e` and $(\,)$ which closes this channel.

9 Summary

We have introduced a message-passing interpretation of sequent calculus proofs in linear logic and given it a syntax. We will summarize the statics (typing rules) and

dynamics (computation rules) after the next lecture. So far, we have only consider internal choice and unit, but due to the presence of recursion we were already able to write some small but nontrivial programs.

References

Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*, pages 222–236, Paris, France, August 2010. Springer LNCS 6269.

Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 26(3):367–423, 2016. Special Issue on Behavioural Types.

Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences, U.S.A.*, 20:584–590, 1934.

Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

Jean-Yves Girard and Yves Lafont. Linear logic and lazy computation. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, volume 2, pages 52–66, Pisa, Italy, March 1987. Springer-Verlag LNCS 250.

Kohei Honda. Types for dyadic interaction. In E. Best, editor, *4th International Conference on Concurrency Theory (CONCUR 1993)*, pages 509–523. Springer LNCS 715, 1993.

Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In P. America, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'91)*, pages 133–147, Geneva, Switzerland, July 1991. Springer-Verlag LNCS 512.

Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In C. Hankin, editor, *7th European Symposium on Programming Languages and Systems (ESOP 1998)*, pages 122–138. Springer LNCS 1381, 1998.

W. A. Howard. The formulae-as-types notion of construction. Unpublished note. An annotated version appeared in: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, 479–490, Academic Press (1980), 1969.

Dag Prawitz. *Natural Deduction*. Almquist & Wiksell, Stockholm, 1965.

Philip Wadler. Propositions as sessions. In *Proceedings of the 17th International Conference on Functional Programming (ICFP 2012)*, pages 273–286, Copenhagen, Denmark, September 2012. ACM Press.

Lecture Notes on
Linear Message Passing II

15-836: Substructural Logics
Frank Pfenning

Lecture 6
September 14, 2023

1 Introduction

We continue the development of a concurrent linear message-passing language. In the last lecture we introduced only the constructs below. Spawn and forward work for arbitrary types, other others apply to a specific type. The actions are viewed from the provider's point of view.

Process	**Action**	**Rule**	**Type**
$x_A \leftarrow P(x)\ ;\ Q(x)$	spawn	cut	A
fwd $x\ y$	forward	id	A
send $x\ k\ ;\ P$	send label k	$\oplus R$	$\oplus\{\ell : A_\ell\}_{\ell \in L}$
recv $x\ (\ell \Rightarrow Q_\ell)_{\ell \in L}$	receive label k	$\oplus L$	$\oplus\{\ell : A_\ell\}_{\ell \in L}$
send $x\ (\,)$	send unit	$\mathbf{1}R$	$\mathbf{1}$
recv $x\ (() \Rightarrow Q)$	receive unit	$\mathbf{1}L$	$\mathbf{1}$
call $p\ x\ y_1 \ldots y_n$	call process p		

The last one is not associated with any particular type but invokes a defined process with name p, passing it x and $y_1, \ldots, y_n$. This is how recursion enters (and exceeds the sequent calculus of linear logic) because these processes may be recursively defined.

We begin by reformulating the dynamics slightly from the previous lecture, recognizing it as a form of *linear inference*!

2 Dynamics as Linear Inference

We mentioned in an earlier lecture that linear inference provides a form of *true concurrency* because with the CBA-graphs we cannot actually tell the order of independent inferences. The rules from the last lecture can in fact be written in the form

of linear inference and take advantage of the earlier observation to get a concurrent semantics "for free".

We need a predicate, say proc, so that $\mathsf{proc}(P)$ is a semantic object representing a running process in state P. Today, we refer to these as *objects* rather than propositions, because we have already coopted those to represent types! We write channels now as a, b and c, because such channels replace variables in the program at runtime. They are typed just as variables are. The whole collection of semantic objects form a *configuration* $\mathcal{C}$, where the order is irrelevant.

$$\frac{\mathsf{proc}(\mathbf{send}\ a\ (\,))\quad \mathsf{proc}(\mathbf{recv}\ a\ ((\,)\Rightarrow Q))}{\mathsf{proc}(Q)}$$

Recall that such a rule picks out two matching objects from the linear state and replaces them by the conclusion. Similarly:

$$\frac{\mathsf{proc}(\mathbf{send}\ a\ k\ ;\ P)\quad \mathsf{proc}(\mathbf{recv}\ a\ (\ell\Rightarrow Q_\ell)_{\ell\in L})\quad (k\in L)}{\mathsf{proc}(P)\qquad \mathsf{proc}(Q_k)}$$

$$\frac{\mathsf{proc}(x_A\leftarrow P(x)\ ;\ Q(x))\quad (a\ \text{fresh})}{\mathsf{proc}(P(a))\qquad \mathsf{proc}(Q(a))}$$

The condition on the freshness of a is somewhat problematic since the premise matches only part of the linear state, but a must be *globally fresh* in the whole state. At the more technical level, this corresponds to *existential quantification*, but since we haven't discussed quantifiers just rely on this global freshness condition.

Now that we know that the dynamics is "just" linear inference, we'll revert to the left-to-right arrow notation for the rules on semantic objects. We actually introduced this $\Delta \longrightarrow \Delta'$ for linear inference before as a more convenient notation (not to be confused with the Δ that types channels in the antecedent of a sequent). Here, it will be $\mathcal{C} \longrightarrow \mathcal{C}'$.

3 Typing Finite Internal Choice

We actually defined only the typing rules for the binary internal choice. Here are the ones for finite sums (or: internal choice between a finite number of alternatives).

$$\frac{\Delta \vdash A_k\quad (k\in L)}{\Delta \vdash \oplus\{\ell : A_\ell\}_{\ell\in L}}\ \oplus R \qquad \frac{\Delta, A_\ell \vdash C\quad (\forall \ell\in L)}{\Delta, \oplus\{\ell : A_\ell\}_{\ell\in L} \vdash C}\ \oplus L$$

The $\oplus L$ rule has one premise for each $\ell \in L$ and is therefore finitary. Adding process terms yields typing rules:

$$\frac{\Delta \vdash P :: (x : A_k) \quad (k \in L)}{\Delta \vdash \mathbf{send}\ x\ k\ ;\ P :: (x : \oplus\{\ell : A_\ell\}_{\ell \in L})}\ \oplus R$$

$$\frac{\Delta, x : A_\ell \vdash Q_\ell :: (z : C) \quad (\forall \ell \in L)}{\Delta, x : \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \mathbf{recv}\ x\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z : C)}\ \oplus L$$

It is convenient to assume that the set L is nonempty, but it certainly wouldn't be problematic logically since $\mathbf{0} = \oplus\{\,\}$.

4 Sending Channels Along Channels

A characteristic of Milner's π-calculus [Milner, 1999] is that one can send channels along channels, changing how the processes are interconnected. In the linear setting, such a protocol is the computational interpretation of $A \otimes B$ and $A \multimap B$. Since we have worked with positive types so far, we start with $A \otimes B$. First, let's review and analyze the logic rules.

$$\frac{\Delta_1 \vdash A \quad \Delta_2 \vdash B}{\Delta_1, \Delta_2 \vdash A \otimes B}\ \otimes R \qquad \frac{\Delta', A, B \vdash C}{\Delta', A \otimes B \vdash C}\ \otimes L$$

The cut reduction between these two rules is straightforward

$$\frac{\dfrac{\Delta_1 \vdash A \quad \Delta_2 \vdash B}{\Delta_1, \Delta_2 \vdash A \otimes B}\ \otimes R \quad \dfrac{\Delta', A, B \vdash C}{\Delta', A \otimes B \vdash C}\ \otimes L}{\Delta', \Delta_1, \Delta_2 \vdash C}\ \mathsf{cut}_{A \otimes B}$$

$$\longrightarrow_R \quad \frac{\Delta_2 \vdash B \qquad \dfrac{\Delta_1 \vdash A \quad \Delta', A, B \vdash C}{\Delta', \Delta_1, B \vdash C}\ \mathsf{cut}_A}{\Delta', \Delta_1, \Delta_2 \vdash C}\ \mathsf{cut}_B$$

From the point of view of processes, the provider of the channel $x : A \otimes B$ spawns two processes, one providing A and one providing B. Both will be connected to the original client of x. There are several pragmatic reasons to chose the following alternative right rule, shown also with the channel-annotated versions.

$$\frac{\Delta \vdash B}{\Delta, A \vdash A \otimes B}\ \otimes R^* \qquad \frac{\Delta \vdash x : B}{\Delta, w : A \vdash x : A \otimes B}\ \otimes R^*$$

$$\frac{\Delta', A, B \vdash C}{\Delta', A \otimes B \vdash C}\ \otimes L \qquad \frac{\Delta', y : A, x : B \vdash C}{\Delta', x : A \otimes B \vdash C}\ \otimes L$$

A process ending with the $\otimes R^*$ rule will send the channel $w : A$ along $x : A \otimes B$ and continue to provide $x : B$. Reusing the send/receive style syntax for these processes, we get:

$$\frac{\Delta \vdash P :: (x : B)}{\Delta, w : A \vdash \mathbf{send}\ x\ w\ ;\ P :: (x : A \otimes B)}\ \otimes R^*$$

$$\frac{\Delta', y : A, x : B \vdash Q(y) :: (z : C)}{\Delta', x : A \otimes B \vdash \mathbf{recv}\ x\ (y \Rightarrow Q(y)) :: (z : C)}\ \otimes L$$

The computation rule just passes the given channel.

$$\mathsf{proc}(\mathbf{send}\ a\ b\ ;\ P), \mathsf{proc}(\mathbf{recv}\ a\ (y \Rightarrow Q(y))) \quad \longrightarrow \quad \mathsf{proc}(P), \mathsf{proc}(Q(b))$$

A nice property of this formulation with $\otimes R^*$ as compared to the standard two-premise version $\otimes R$ is that the only rule that spawns a new process is now the cut rule. Also, every construct (send or receive) has a single continuation, except for sending the unit which terminates a process.

But is this rule substitution actually okay? We should check that (a) $\otimes R^*$ is sound in the sense that we can derive it using $\otimes R$, and (b) that it is complete in the sense that we can use it to derive $\otimes R$. This is an instructive exercise, so you might want to give it a try before reading on.

In one direction we require an identity, in the other direction a cut. If we make up a syntax for the standard two-premise rule, we can translate the derivations below into small programs.

$$\frac{\dfrac{}{A \vdash A}\,\mathsf{id} \qquad \Delta \vdash B}{\Delta, A \vdash A \otimes B}\,\otimes R \qquad\qquad \frac{\Delta_1 \vdash A \qquad \dfrac{\Delta_2 \vdash B}{A, \Delta_2 \vdash A \otimes B}\,\otimes R^*}{\Delta_1, \Delta_2 \vdash A \otimes B}\,\mathsf{cut}$$

The downside of a wholesale *replacement* of $\otimes R$ with $\otimes R^*$ is that cut elimination no longer holds in its usual formulation. So the new calculus is most easily justified by translation as we did here. Alternatively, one can formulate an alternative "cut elimination" theorem that allows some cuts that satisfy the subformula property (so-called *analytic cuts*) to remain in proofs. We will not pursue this further here since it does not impact the computational interpretation of the MPASS language.

5 Example: Sequences

In a functional language it is often convenient to work with lists. Here, they are sequences of channels of some arbitrary type A, but we still call them lists.

$$\mathsf{list}_A = \oplus\{\mathsf{nil} : \mathbf{1}, \mathsf{cons} : A \otimes \mathsf{list}_A\}$$

This type is entirely positive, so messages still only flow from provider to client. We can define the standard constructors corresponding to the alternatives in the sum. Since MPASS at present does not support polymorphism, we restrict ourselves to lists of binary numbers. It should be clear that nothing depends on this choice. We can define the boilerplate constructors corresponding to nil and cons.

```
type list = +{'nil : 1, 'cons : bin * list}

proc nil (l : list) = send l 'nil ; send l ()
proc cons (l : list) (x : bin) (k : list) =
  send l 'cons ; send l x ; fwd l k
```

A standard *functional* program appends two lists. We can write the same program in different ways. In Listing 1 we show a version where the recursive call to append is a *tail call* (which usually isn't possible in a functional language). We annotate each line with the typing of the continuation of the process following this line. In the file lecture6.mps there is also a test case for the append process.

6 External Choice

We now come to negative connectives, starting with external choice. Analogously to internal choice, we also generalize external choice to have a finite number of

```
1 proc append (l : list) (k1 : list) (k2 : list) =
2   recv k1 ( 'nil =>           % k1 : 1, k2 : list |- l : list
3             recv k1 (() =>    % k2 : list |- l : list
4             fwd l k2)
5           | 'cons =>          % k1 : bin * list, k2 : list |- l : list
6             send l 'cons ;    % k1 : bin * list, k2 : list |- l : bin * list
7             recv k1 (x =>     % x : bin, k1 : list, k2 : list |- l : bin * list
8             send l x ;        % k1 : list, k2 : list |- l : list
9             call append l k1 k2) )
```

Listing 1: Append process in MPASS

alternatives. First, the purely logical rules.

$$\frac{\Delta \vdash A_\ell \quad (\forall \ell \in L)}{\Delta \vdash \&\{\ell : A_\ell\}_{\ell \in L}}\ \&R \qquad \frac{\Delta, A_k \vdash C \quad (k \in L)}{\Delta, \&\{\ell : A_\ell\}_{\ell \in L} \vdash C}\ \&L$$

Conversely to the rules for internal choice, the provider has to be prepared for the client to choose a suitable $k \in L$. We also see that if these two proofs are combined with a cut, then in this case the *clients sends a label to the provider*. This is a characteristic of negative types, where the right rules are invertible and therefore contain no information.

Because of the symmetry between external and internal choice, we can reuse the process notation, just swapping the roles of provider and client.

$$\frac{\Delta \vdash P_\ell :: (x : A_\ell) \quad (\forall \ell \in L)}{\Delta \vdash \mathbf{recv}\ x\ (\ell \Rightarrow P_\ell)_{\ell \in L} :: (x : \&\{\ell : A_\ell\}_{\ell \in L})}\ \&R$$

$$\frac{\Delta, x : A_k \vdash Q :: (z : C) \quad (k \in L)}{\Delta, x : \&\{\ell : A_\ell\}_{\ell \in L} \vdash \mathbf{send}\ x\ k\ ;\ Q :: (z : C)}\ \&L$$

Furthermore, the same rule in the dynamics still applies and doesn't need to be changed! Writing it again in our linear notation, just for reference:

$$\mathsf{proc}(\mathbf{recv}\ a\ (\ell \Rightarrow P_\ell)_{\ell \in L}), \mathsf{proc}(\mathbf{send}\ a\ k\ ;\ Q) \quad \longrightarrow \quad \mathsf{proc}(P_k), \mathsf{proc}(Q) \quad (k \in L)$$

7 Linear Implication

Linear implication $A \multimap B$ is symmetric to $A \otimes B$: the provider *receives* a channel of type A instead of sending one. We also change the sequent calculus rules in an analogous way to avoid multiple premises for the $\multimap L$ rule.

$$\frac{\Delta, A \vdash B}{\Delta \vdash A \multimap B}\ \multimap R \qquad \frac{\Delta' , B \vdash C}{\Delta', A, A \multimap B \vdash C}\ \multimap L^*$$

We move directly to the process assignment, leaving the logical investigation of these rules to Assignment 2.

$$\frac{\Delta, y : A \vdash P :: (x : B)}{\Delta \vdash \mathbf{recv}\ x\ (y \Rightarrow P(y)) :: (x : A \multimap B)}\ \multimap R$$

$$\frac{\Delta', x : B \vdash Q :: (z : C)}{\Delta', w : A, x : A \multimap B \vdash \mathbf{send}\ x\ w\ ;\ Q :: (z : C)}\ \multimap L^*$$

Once again, the computational rule for this construct already exists: the rule for tensor applies. We restate it for completeness.

$$\mathsf{proc}(\mathbf{recv}\ a\ (y \Rightarrow P(y))), \mathsf{proc}(\mathbf{send}\ a\ b\ ;\ Q) \quad \longrightarrow \mathsf{proc}(P(b)), \mathsf{proc}(Q)$$

8 Example: A Storage Server

We now use an interface to a storage server as an example incorporating negative types (external choice and linear implication). The client has the option between inserting or deleting an element from the store. When deleting, the provider replies with none if there is no element in the store, or some followed by the element if there is one.

$$\begin{aligned}\mathsf{store}_A = \&\{\ &\mathsf{ins} : A \multimap \mathsf{store}_A,\\ &\mathsf{del} : \oplus\{\ \mathsf{none} : \mathbf{1}, \mathsf{some} : A \otimes \mathsf{store}_A\ \}\ \}\end{aligned}$$

As for sequences, in the implementation we commit to the type A to be the binary numbers since MPASS doesn't currently support polymorphism.

Our implementation behaves like a stack; for a queue, see Assignment 3. The state of the store is represented by a sequence of processes, each holding one element, with the last one being empty. We define this empty process first. In the case of insert we have to start a `node` process with the element we received. The tail of the this new store must again be empty, so we have to spawn a new empty process.

```
proc empty (s : store) =
     recv s ( 'ins => recv s (x => e <- call empty e ;
                                   call node s x e )
            | 'del => send s 'none ; send s () )
```

When receiving a `'del` message we respond with `'none` and terminate. In order to keep the store service running, we could also change the type and recurse. With that change, though, a process using a store could never terminate, because the store it uses can never terminate. So we would have to add another option to the interface to deallocate the store. However, due to linearity, this could only succeed if the store is in fact empty, so it seems better to integrate this into the interaction after a deletion.

For a node process, we create a new node s' with the prior value x and then become (through a tail call) a node holding the new value y. The external interface remains the channel s. We'll walk through this code practicing *type-directed programming*: letting the types inform us about our choices. Linearity is an even bigger help here than types are in functional programming already because they further constrain our choices.

We begin with just the header, showing the typing judgment in effect for the body of the node.

```
1  proc node (s : store) (x : bin) (t : store) =
2       % (x : bin) (t : store) |- (s : store)
```

At this point we can't do anything useful with x or t. On the other hand, the type of s is an external choice between labels ins and del, so the client will send one of these two messages. We write down the two branches and note the typing in the first branch. Note that the type of s has advanced from store to bin $\multimap$ store due to the receipt of the ins label.

```
1  proc node (s : store) (x : bin) (t : store) =
2     recv s ( 'ins => ... % (x : bin) (t : store) |- (s : bin -o store)
3            | 'del => ... )
```

From the type of s we can see it is still the client's turn, this time to send us a channel (say y) of type bin. After receiving y we own channel y (in addition to x and t), and the type of s has cycled all the way around back to store.

```
1  proc node (s : store) (x : bin) (t : store) =
2     recv s ( 'ins => recv s (y =>  ...
3                      % (y : bin) (x : bin) (t : store) |- (s : store)
4            | 'del => ... )
```

At this point it seems we have more than one option. We could insert x or y into t. Or we could create a new node holding x with tail t and then recur with y. Let's do the latter, because it looks like it might avoid a cascading sequent of inserts rippling to the end of the chain of nodes. The idiom allocates a new channel s' and spawns a new process mode providing this channel. Since we pass it x and t they disappear from the antecedents and are supplanted by s'.

```
1  proc node (s : store) (x : bin) (t : store) =
2     recv s ( 'ins => recv s (y => s' <- call node s' x t ;
3                      % (y : bin) (s' : store) |- (s : store)
4            | 'del => ... )
```

At this point we can make a recursive call to *node*, providing s and holding the value y with tail s'.

```
1  proc node (s : store) (x : bin) (t : store) =
2     recv s ( 'ins => recv s (y => s' <- call node s' x t ;
3                                    call node s y s')
4            | 'del => ... )
```

This takes care of the insert branch. In the delete branch we have the following type:

```
1  proc node (s : store) (x : bin) (t : store) =
2     recv s ( 'ins => recv s (y => s' <- call node s' x t ;
3                                 call node s y s')
4            | 'del =>
5     % (x : bin) (t : store) |- (s : +{'none : 1, 'some : bin * store})
6               ... )
```

Since we hold an element x we have to respond with the some label.

```
1  proc node (s : store) (x : bin) (t : store) =
2       recv s ( 'ins => recv s (y => s' : store <- call node s' x t ;
3                                 call node s y s')
4              | 'del => send s 'some ;
5                 % (x : bin) (t : store) |- (s : bin * store)
6              )
```

Again, the positive type bin $\otimes$ store means we should send a channel of type bin along s—in this case the element we hold (x).

```
1  proc node (s : store) (x : bin) (t : store) =
2       recv s ( 'ins => recv s (y => s' : store <- call node s' x t ;
3                                 call node s y s')
4              | 'del => send s 'some ;
5                        send s x ;
6                        % (t : store) |- (s : store)
7              )
```

At this point we forward from t to s, since t represents the remainder of the stack.

```
1  proc node (s : store) (x : bin) (t : store) =
2       recv s ( 'ins => recv s (y => s' : store <- call node s' x t ;
3                                 call node s y s')
4              | 'del => send s 'some ;
5                        send s x ;
6                        fwd s t )
```

We can convert a store back to a list simply by recursively deleting the elements until the store is empty. This time we just show the code at the end, but we recommend you walk through it similar to the way we did for the *node* process.

```
proc store2list (l : list) (s : store) =
   send s 'del ;
   recv s ( 'none => recv s (() => send l 'nil ; send l ())
          | 'some => recv s (x => send l 'cons ; send l x ;
                     call store2list l s) )
```

You can find a few more examples of store/list programs in the file lecture6.mps.

9 A Brief Note on Parametricity

As a side remark, the *structural* version of the polymorphic type

$$\begin{aligned}\mathsf{store}_A = \&\{\ &\mathsf{ins} : A \to \mathsf{store}_A,\\ &\mathsf{del} : +\{\ \mathsf{none} : \mathbf{1}, \mathsf{some} : A \times \mathsf{store}_A\ \}\ \}\end{aligned}$$

guarantees that the elements returned by the store are among the ones inserted into an empty store. This is an application of *parametricity* [Reynolds, 1983]. However, the store may drop or duplicate elements. The linear type

$$\begin{aligned}\mathsf{store}_A = \&\{\ &\mathsf{ins} : A \multimap \mathsf{store}_A,\\ &\mathsf{del} : \oplus\{\ \mathsf{none} : \mathbf{1}, \mathsf{some} : A \otimes \mathsf{store}_A\ \}\ \}\end{aligned}$$

sharpens this: the elements returned must be a *permutation* of the elements inserted.

We also conjecture (but haven't proved) that in the case of ordered connectives we can guarantee the behavior of a queue or a stack.

10 Summary

We summarize the statics (typing rules) and dynamics (computation rules) of the MPASS language in Figure 1 and Figure 2 respectively.

References

Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

John C. Reynolds. Types, abstraction, and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing 83*, pages 513–523. Elsevier, September 1983.

$$\frac{\Delta \vdash P(x) :: (x : A) \quad \Delta', x : A \vdash Q(x) :: (z : C)}{\Delta, \Delta' \vdash x_A \leftarrow P(x) \; ; \; Q(x) :: (z : C)} \; \mathsf{cut} \qquad \frac{}{x : A \vdash \mathbf{fwd}\ y\ x :: (y : A)} \; \mathsf{id}$$

$$\frac{}{\cdot \vdash \mathbf{send}\ x\ (\,) :: (x : \mathbf{1})}\ \mathbf{1}R \qquad \frac{\Delta \vdash Q :: (z : C)}{\Delta, x : \mathbf{1} \vdash \mathbf{recv}\ x\ (() \Rightarrow Q) :: (z : C)}\ \mathbf{1}L$$

$$\frac{\Delta \vdash P :: (x : A_k) \quad (k \in L)}{\Delta \vdash \mathbf{send}\ x\ k\ ;\ P :: (x : \oplus\{\ell : A_\ell\}_{\ell \in L})}\ \oplus R$$

$$\frac{\Delta, x : A_\ell \vdash Q_\ell :: (z : C) \quad (\forall \ell \in L)}{\Delta, x : \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \mathbf{recv}\ x\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (z : C)}\ \oplus L$$

$$\frac{\Delta \vdash P :: (x : B)}{\Delta, w : A \vdash \mathbf{send}\ x\ w\ ;\ P :: (x : A \otimes B)}\ \otimes R^*$$

$$\frac{\Delta', y : A, x : B \vdash Q(y) :: (z : C)}{\Delta', x : A \otimes B \vdash \mathbf{recv}\ x\ (y \Rightarrow Q(y)) :: (z : C)}\ \otimes L$$

$$\frac{\Delta \vdash P_\ell :: (x : A_\ell) \quad (\forall \ell \in L)}{\Delta \vdash \mathbf{recv}\ x\ (\ell \Rightarrow P_\ell)_{\ell \in L} :: (x : \&\{\ell : A_\ell\}_{\ell \in L})}\ \& R$$

$$\frac{\Delta, x : A_k \vdash Q :: (z : C) \quad (k \in L)}{\Delta, x : \&\{\ell : A_\ell\}_{\ell \in L} \vdash \mathbf{send}\ x\ k\ ;\ Q :: (z : C)}\ \& L$$

$$\frac{\Delta, y : A \vdash P :: (x : B)}{\Delta \vdash \mathbf{recv}\ x\ (y \Rightarrow P(y)) :: (x : A \multimap B)}\ \multimap R$$

$$\frac{\Delta', x : B \vdash Q :: (z : C)}{\Delta', w : A, x : A \multimap B \vdash \mathbf{send}\ x\ w\ ;\ Q :: (z : C)}\ \multimap L^*$$

Figure 1: Statics for MPass

$$
\begin{array}{lcll}
\mathsf{proc}(x_A \leftarrow P(x)\ ;\ Q(x)) & \longrightarrow & \mathsf{proc}(P(a)), \mathsf{proc}(Q(a)) & (a \text{ fresh}) \\
\mathsf{proc}(P(b)), \mathsf{proc}(\mathbf{fwd}\ a\ b) & \longrightarrow & \mathsf{proc}(P(a)) & \\
\mathsf{proc}(\mathbf{fwd}\ a\ b), \mathsf{proc}(Q(a)) & \longrightarrow & \mathsf{proc}(Q(b)) & \\
\mathsf{proc}(\mathbf{send}\ x\ k\ ;\ P), \mathsf{proc}(\mathbf{recv}\ x\ (\ell \Rightarrow Q_\ell)_{\ell \in L}) & \longrightarrow & \mathsf{proc}(P), \mathsf{proc}(Q_k) & (k \in L) \\
\mathsf{proc}(\mathbf{send}\ x\ (\,)), \mathsf{proc}(\mathbf{recv}\ x\ ((\,) \Rightarrow Q)) & \longrightarrow & Q & \\
\mathsf{proc}(\mathbf{send}\ a\ b\ ;\ P), \mathsf{proc}(\mathbf{recv}\ a\ (y \Rightarrow Q(y))) & \longrightarrow & \mathsf{proc}(P), \mathsf{proc}(Q(b)) &
\end{array}
$$

Figure 2: Dynamics for MPASS

Lecture Notes on Preservation and Progress

15-836: Substructural Logics
Frank Pfenning

Lecture 7
September 19, 2023

1 Introduction

Our investigation has shown the close correspondences between linear propositions and session types, between sequent proofs and synchronous message-passing programs, and between cut reduction and communication. Despite these close connections, there are also differences. For example, permuting cut reductions and identity expansion are related to process equalities, but not directly to computation. Here are some other key differences:

Recursion. Recursion is a central concept in programming languages but much less prevalent in the study of logic. Nevertheless, there is a whole branch of logic dedicated to arithmetic, including induction and primitive recursion. See, for example, once again Gentzen's pioneering work [Gentzen, 1936]. Also, *infinitary proofs* have been studied—we'll see an example in Lecture 8 on *Subtyping*.

Observability. The primary purpose of proofs is to convince you that a proposition is true and explain why. As such, the whole proof must be subject to inspection so we can check it and also understand it. The primary purpose of programs is computation. As such, we are mostly interested in observing its outcome (assuming that maybe we have separately verified or trust in its correctness). But we do not observe functions directly, only their results on particular inputs. This gives the provider of a library the freedom to change function definitions (e.g., improve their efficiency) without changing the observable input/output behavior.

After motivating our MPASS language through logic and proof theory, we will see the differences that arise when we put our programmers' hats on. In particular, the theorems we prove about the logic and the theorems we prove about our linear

message-passing programming language will by necessity be different. Nevertheless, we can see how the effort we invested in proof of cut elimination does not go to waste; it is just that the key insights appear in different contexts.

From the foregoing discussion it might appear that logic and programming and in fact two different subjects, albeit with close connections. In my view they are in fact synthesized and generalized in *constructive type theory*. On one side, type theory generalizes logic by providing the intrinsic ability to talk about its own proofs. On the other side, type theory generalizes programming languages by providing the intrinsic ability to reason about their correctness. I don't know how much opportunity we will have to explore *substructural type theory* (in this sense) in this course. Many questions here are not yet well understood.

2 Integrating Recursion

In the last two lectures and the MPASS examples we have seen that recursion is introduced in two ways, both coming down to *definitions*:

(1) *Types* may be defined recursively. For example,

$$\mathsf{nat} = \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\}$$

Such types are *equirecursive* in the sense that recursion at the type level is not associated with any messages. During type-checking we are permitted to silently replace a type name such as nat with its definition. In Lecture 8 we explore the algorithmic consequences of this decision.

(2) *Processes* may be defined recursively. For example, the successor process on binary numbers required recursion in order to represent the carry bit.

These two go hand-in-hand: often the recursive structure of processes is dictated by the recursive structure of the types they operate on.

Based on these observation we integrate recursion into our programming language via a *signature* Σ containing definitions at the type and process level. We write t and s for type names, p for process names, and $\overline{(y : B)}$ for a sequence of parameters $y_i : B_i$.

$$\text{Signature} \quad \Sigma \quad ::= \quad \cdot \mid \Sigma, t = A \mid \Sigma, p\ (x : A)\ \overline{(y : B)} = P$$

Because we want to make mutual recursion as natural as possible, the individual declarations in a signature Σ are checked for correctness against the whole signature rather than the usual left-to-right manner. We write $\vdash \Sigma\ \mathit{sig}$ to mean that Σ is a valid signature, and $\vdash_\Sigma \Sigma'\ \mathit{sig}$ to mean that all declarations in Σ' are valid in the signature Σ. We have the following rules:

$$\frac{}{\vdash_\Sigma (\cdot)\ \mathit{sig}} \qquad \frac{\vdash_\Sigma \Sigma'\ \mathit{sig} \quad \vdash_\Sigma A\ \mathit{type}}{\vdash_\Sigma (\Sigma', t = A)\ \mathit{sig}} \qquad \frac{\vdash_\Sigma \Sigma'\ \mathit{sig} \quad \overline{(y : B)} \vdash_\Sigma P(x, \overline{y}) :: (x : A)}{\vdash_\Sigma (\Sigma', p\ (x : A)\ \overline{(y : B)} = P(x, \overline{y})\ \mathit{sig}}$$

In these rules the index Σ in $\vdash_\Sigma$ never changes: the signature is in a sense global. Therefore we omit it from all the judgments and imagine that in any given situation it will be fixed and valid. Furthermore, all type names and process names in a signature must be distinct.

The judgment $\vdash_\Sigma A\ \mathit{type}$ means that A is (or can be implicitly expanded to) one of the types in our language, and that all the type names in A are defined in Σ.

Computationally, a **call** simply expands to its definition.

$$\mathsf{proc}(\mathbf{call}\ p\ a\ \overline{b}) \quad \longrightarrow \quad \mathsf{proc}(P(a, \overline{b})) \quad \text{where } p\ (x : A)\ \overline{(y : B)} = P(x, \overline{y}) \in \Sigma$$

3 Typing Configurations of Processes

When running a program, we imagine starting with a single process P. During the computation, many new processes may be spawned and interact with each other. We think of these processes of defining a *multiset* in the sense of linear inference. We can define it more syntactically with the following.

$$\text{Configuration} \quad \mathcal{C} \quad ::= \quad \mathsf{proc}(P) \mid \mathcal{C}_1, \mathcal{C}_2 \mid \cdot$$

Here, the comma operator is associative and commutative with the empty configuration $(\cdot)$ as its unit.

There are many meaningless configurations, such as one where one process sends a label that the recipient does not expect, or one where a process send unit while the recipient expects a channel. Undoubtedly, while programming in MPASS you have encountered such incorrect processes, which would have become incorrect configurations when running.

How do we ensure configurations are meaningful, which is to say, they are well-typed? For a single process, our judgment is $\Delta \vdash P :: (x : A)$ which means that P provides x at type A and uses the channels in Δ at their given types. A configuration may consist of multiple processes, so this generalizes to

$$\Delta \vdash \mathcal{C} :: \Delta'$$

which means the configuration $\mathcal{C}$ *uses* (is a client to) all the channel in Δ and provides all the channels in Δ'. We might at first hypothesize the following rule:

$$\frac{\Delta \vdash P :: (a : A)}{\Delta \vdash \mathsf{proc}(P) :: (a : A)}\ \mathsf{proc}?$$

Here, we take advantage of the fact that channels a, b, c behave exactly the same under typing as variables x, y, z so that the typing judgment in the premise is well-defined. But this is not quite sufficient: There may be *other* channels among the

antecedents to $\mathsf{proc}(P)$ that are not used by P. These will still be available to clients of this configuration. So we get

$$\frac{\Delta \vdash P :: (a : A)}{\Delta', \Delta \vdash \mathsf{proc}(P) :: (\Delta', a : A)}\ \mathsf{proc}$$

For process terms, cut requires that the provider and the client agree on the type of the private channel that enables communication between them. For configurations, this has to hold for *all* channels since they are all derived from cut. The empty context just passes through all channels provided to it, since it neither uses nor provides any channels of its own.

$$\frac{\Delta_0 \vdash \mathcal{C}_1 :: \Delta_1 \quad \Delta_1 \vdash \mathcal{C}_2 :: \Delta_2}{\Delta_0 \vdash \mathcal{C}_1, \mathcal{C}_2 :: \Delta_2}\ \mathsf{join} \qquad \frac{}{\Delta \vdash (\cdot) :: \Delta}\ \mathsf{empty}$$

At this point we notice an emerging conflict. On one hand we think of configurations as multisets in the sense that the order of the individual processes is relevant. On the other, for a typing derivation we require some ordering. As we can see from the join rule, the typing derivation requires that each provider precedes its client. Furthermore, we need to make sure that this relation is uniquely determined so we stipulate that each channel in a configuration with

$$\Delta \vdash \mathcal{C} :: \Delta'$$

occurs either in Δ, or in Δ' (or both), or has exactly one provider and exactly one client in $\mathcal{C}$. When we start from an initial configuration with a single main process, this will be true, and all the rules can be seen to preserve this property. The only doubt one might have is about cut, but the new channel is chosen such that it does not already occur in the whole configuration. Moreover, this new channel has exactly one provider and exactly one client.

Coming back to the ordering, the join operator as combining two *derivations* is associative with unit empty, but it is not commutative. For a configuration to be well-typed we require that there is an ordering of the processes that can be typed with the given rules. This ordering is not unique: satisfying the provider-before-client requirement still may leave many options. Which of these possibilities we pick is irrelevant. A key property only briefly mentioned in lecture is the following exchange lemma.

Lemma 1 (Exchange) *If process P provides a channel a which is not used by the following process Q in the configuration typing, then the two processes can be exchanged.*

Proof: By inspection of the two typing derivations, since the first represents the most general case of two consecutive processes satisfying the given condition. Given

$$\frac{\dfrac{\Delta_P \vdash P :: (a : A)}{\Delta', \Delta_P \vdash \mathsf{proc}(P) :: (\Delta', a : A)}\ \mathsf{proc} \qquad \dfrac{\Delta_Q \vdash Q :: (c : C)}{\Delta', a : A, \Delta_Q \vdash \mathsf{proc}(Q) :: (\Delta', a : A, c : C)}\ \mathsf{proc}}{\Delta', \Delta_P, \Delta_Q \vdash \mathsf{proc}(P), \mathsf{proc}(Q) :: (\Delta', a : A, c : C)}\ \mathsf{join}$$

we construct

$$\dfrac{\dfrac{\Delta_Q \vdash Q :: (c : C)}{\Delta', \Delta_P, \Delta_Q \vdash \mathsf{proc}(Q) :: (\Delta', \Delta_P, c : C)}\ \mathsf{proc} \qquad \dfrac{\Delta_P \vdash P :: (a : A)}{\Delta', \Delta_P, c : A \vdash \mathsf{proc}(P) :: (\Delta', a : A, c : C)}\ \mathsf{proc}}{\Delta', \Delta_P, \Delta_Q \vdash \mathsf{proc}(Q), \mathsf{proc}(P) :: (\Delta', a : A, c : C)}\ \mathsf{join}$$

□

We can iterate the exchange so that a process P that provides a channel a can always be moved to the right until it is next to its client. If it does not have a client, then it can be moved to be the rightmost process in a configuration.

4 Preservation

Unlike pure logic where cut elimination always terminates in a cut-free proof, computation may run forever due to the presence of recursion. So rather than cut elimination (or admissibility of cut as the key lemma) we prove that as computation proceeds the configuration remains well-typed. Since the computation rules are (mostly) derived from principal cut reductions, patterns from the proof of the admissibility of cut recur.

We have already remarked on a fundamental property, namely that the type of channels evolves as communication takes place. So it what sense are types actually preserved? What happens is that the types of *internal channels* in a configuration changes consistently between client and provider, but the types of *externally visible channels* remain invariant.

Theorem 2 (Preservation) *If $\Delta \vdash \mathcal{C} :: \Delta'$ and $\mathcal{C} \longrightarrow \mathcal{D}$ then $\Delta \vdash \mathcal{D} :: \Delta'$.*

Theorem 3 *The proof proceeds by cases over the forms of the reduction. There are four kinds of cases: spawn (= cut), fwd (= identity), call, and interactions. We show two representative cases. We analyze the configuration in the order of its typing derivation.*

Cut: $\mathcal{C} = (\mathcal{C}_L, \mathsf{proc}(x_A \leftarrow P(x)\ ;\ Q(x)), \mathcal{C}_R)$ *where*

$$\begin{array}{l} \Delta \vdash \mathcal{C}_L :: \Delta_L \\ \Delta_L \vdash \mathsf{proc}(x_A \leftarrow P(x)\ ;\ Q(x)) :: \Delta_R \\ \Delta_R \vdash \mathcal{C}_R :: \Delta' \end{array}$$

and

$$\mathcal{D} = (\mathcal{C}_L, \mathsf{proc}(P(a)), \mathsf{proc}(Q(a)), \mathcal{C}_R)$$

for a globally fresh channel a.

In order to construct a typing derivation for $\mathcal{D}$ as required, we apply inversion *to the typing*

$$\Delta_L \vdash \mathsf{proc}(x_A \leftarrow P(x)\ ;\ Q(x)) :: \Delta_R$$

This means we analyze the typing rules for processes and consider what we might say about this typing derivation. We find that for some Δ'_L, Δ_P, Δ_Q, and $c : C$, we must have

$$\begin{array}{l}\Delta_L = (\Delta'_L, \Delta_P, \Delta_Q) \\ \Delta_P \vdash P(x) :: (x : A) \\ \Delta_Q, x : A \vdash Q(x) :: (c : C) \\ \Delta_R = (\Delta'_L, c : C)\end{array}$$

Because a is globally fresh, we can substitute it for x in the two typing derivation in the middle, and conclude that

$$\begin{array}{l}\Delta_L = (\Delta'_L, \Delta_P, \Delta_Q) \\ \Delta_P \vdash P(a) :: (a : A) \\ \Delta_Q, a : A \vdash Q(a) :: (c : C) \\ \Delta_R = (\Delta'_L, c : C)\end{array}$$

Now we can construct a typing derivation for $\mathcal{D} = (\mathcal{C}_L, \mathsf{proc}(P(a)), \mathsf{proc}(Q(a)), \mathcal{C}_R)$ from

$$\begin{array}{l}\Delta \vdash \mathcal{C}_L :: \Delta_L \\ \Delta_L = (\Delta'_L, \Delta_P, \Delta_Q) \\ \Delta'_L, \Delta_P, \Delta_Q \vdash \mathsf{proc}(P(a)) :: (\Delta'_L, \Delta_Q, a : A) \\ \Delta'_L, \Delta_Q, a : A \vdash \mathsf{proc}(Q(a)) :: (\Delta'_L, c : C) \\ (\Delta'_L, c : C) = \Delta_R \\ \Delta_R \vdash \mathcal{C}_R :: \Delta'\end{array}$$

This concludes this case of type preservation.

$\otimes R/\otimes L$: *$\mathcal{C} = (\mathcal{C}_L, \mathsf{proc}(\mathbf{send}\ a\ b\ ;\ P), \mathsf{proc}(\mathbf{recv}\ a\ (y \Rightarrow Q(y))), \mathcal{C}_R)$ where*

$$\begin{array}{l}\Delta \vdash \mathcal{C}_L :: \Delta_L \\ \Delta_L \vdash \mathsf{proc}(\mathbf{send}\ a\ b\ ;\ P), \mathsf{proc}(\mathbf{recv}\ a\ (y \Rightarrow Q(y))) :: \Delta_R \\ \Delta_R \vdash \mathcal{C}_R :: \Delta\end{array}$$

and

$$\mathcal{D} = (\mathcal{C}_L, \mathsf{proc}(P), \mathsf{proc}(Q(b)), \mathcal{C}_R)$$

Here we have taken advantage the exchange lemma to restrict ourselves to the case where the provider and client are immediately adjacent in the typing derivation. Again we apply inversion *to analyze the possible typing derivations for the middle line and find that for some Δ'_L, Δ_P, $a : A$, $b : B$, and $c : C$ we must have*

$$\begin{array}{l}\Delta_L = (\Delta'_L, \Delta_P, b : B, \Delta_Q) \\ \Delta_P \vdash P :: (a : A) \\ \Delta_Q, a : A, b : B \vdash Q(b) :: (c : C) \\ \Delta_R = (\Delta'_L, c : C)\end{array}$$

A critical step here is examine the typing rules for $\mathbf{send}\ a\ b\ ;\ P$ *and* $\mathbf{recv}\ a\ (y \Rightarrow Q(y))$ *for which there is only one each once we know the first process is the provider (which comes from their relative position in the typing derivation for* $\mathcal{C}$*)*

From these pieces we can assemble a typing derivation for

$$\mathcal{D} = (\mathcal{C}_L, \mathsf{proc}(P), \mathsf{proc}(Q(b)), \mathcal{C}_R)$$

as follows:

$$\begin{array}{l}
\Delta \vdash \mathcal{C}_L :: \Delta_L \\
\Delta_L = (\Delta'_L, \Delta_P, b : B, \Delta_Q) \\
\Delta'_L, \Delta_P, b : B, \Delta_Q \vdash \mathsf{proc}(P) :: (\Delta'_L, a : A, b : B, \Delta_Q) \\
\Delta'_L, a : A, b : B, \Delta_Q \vdash Q(b) :: (\Delta'_L, c : C) \\
(\Delta'_L, c : C) = \Delta_R \\
\Delta_R \vdash \mathcal{C}_R :: \Delta'
\end{array}$$

There is a lot of bureaucracy in the proof of preservation, but ultimately the core reasoning step in the communication steps is that cut reduction preserves the conclusion of the cut (in particular, the antecedents and the succedent).

5 Progress

Preservation means that in any computation $\mathcal{C}_1 \longrightarrow \mathcal{C}_2 \longrightarrow \cdots$ the interface to the configuration never changes. Cut elimination would also predict that reduction always terminates, but that's not true in the presence of recursion unless we make some restrictions. Instead, we would like to prove that "we never get stuck": either we can take a step, or the configuration is *final* in a well-defined way. We are looking for analogue to the statement that in functional languages every expression e either can take a step or it is a value already. But what's the analogue of value? In our language of *synchronous communication* (that is, both sender and receiver proceed in lock-step when a message is exchanged) a configuration is *final* if all processes attempt to communicate along an external channel. Such a channel does not have a second endpoint, so such a process can legitimately not make further progress.

To keep the argument simple we assume that the configuration is closed on the left, that is,

$$\cdot \vdash \mathcal{C} :: \Delta$$

In other words, $\mathcal{C}$ *provides* some external channels but does not use any. That's analogous to the usual assumption that in a functional language we only evaluate *closed* expressions, that is, expressions without free variables.

Theorem 4 (Progress) *If* $\cdot \vdash \mathcal{C} :: \Delta$ *then either* $\mathcal{C}$ *is final or* $\mathcal{C} \longrightarrow \mathcal{D}$ *for some* $\mathcal{D}$*.*

Proof: This time we do a right-to-left induction over the structure the given typing derivation (which we associate to the left). So $\mathcal{C} = (\mathcal{C}_L, \mathsf{proc}(P))$ for some process P with $\cdot \vdash \mathcal{C}_L :: \Delta'$ and $\Delta' \vdash \mathsf{proc}(P) :: \Delta$.

By induction hypothesis, either $\mathcal{C}_L \longrightarrow \mathcal{D}_L$ for some $\mathcal{D}_L$ or $\mathcal{C}_L$ is final.

In the first case $\mathcal{C} \longrightarrow (\mathcal{D}_L, \mathsf{proc}(P))$ by definition of reduction.

In the second case, all processes in $\mathcal{C}_L$ will try to communicate along the channel that they provide. Now we distinguish cases based on the process P.

Cut/Spawn: $P = (x_A \leftarrow P_1(x) \ ; \ P_2(x))$ for some P_1 and P_2. Then $\mathsf{proc}(P) \longrightarrow \mathsf{proc}(P_1(a)), \mathsf{proc}(P_2(a))$ for a fresh a, and therefore also $\mathcal{C} \longrightarrow (\mathcal{C}_L, \mathsf{proc}(P_1), \mathsf{proc}(P_2))$.

Receive Channel: $P = (\mathbf{recv}\ a\ (y \Rightarrow P'(y)))$ for some P'. If P provides a (that is, $a : A \in \Delta$ for some A) then all of $\mathcal{C}$ is final.

Otherwise P uses a and must (by inversion) end in the $\otimes L$ rule. That is the typing derivation of $\Delta' \vdash \mathsf{proc}(P) :: \Delta$ looks like

$$\dfrac{\dfrac{\Delta_P, a : A, y : B \vdash P'(y) :: (c : C)}{\Delta_P, a : B \otimes A \vdash (\mathbf{recv}\ a\ (y \Rightarrow P'(y))) :: (c : C)}\ \otimes L}{\Delta' \vdash \mathsf{proc}(P) :: \Delta}\ \mathsf{proc}$$

where $\Delta' = (\Delta_L, \Delta_P, a : B \otimes A)$ and $\Delta = (\Delta_L, c : C)$

Because $\mathcal{C}_L :: (\Delta_L, \Delta_P, a : B \otimes A)$ there must be a process in $\mathcal{C}_L$ providing $a : B \otimes A$. In particular, it cannot be part of the antecedents of $\mathcal{C}_L$ because these must be empty.

By inversion on the typing of a we find that there must be an object $\mathsf{proc}(Q)$ in $\mathcal{C}_L$ that provides $a : B \otimes A$. Moreover, since $\mathcal{C}_L$ is final, this process must be trying communicate along a, so it must have form $Q = (\mathbf{send}\ a\ b\ ;\ Q')$. By the rule for sending and receiving a channel Q and P can interact, and therefore $\mathcal{C} \longrightarrow \mathcal{D}$ for some $\mathcal{D}$.

□

Again, there is a lot of bureaucracy, but in the end the progress theorem comes down to the fact that during the proof of admissibility of cut all the principal cases could be reduced (= make progress).

6 Observation

We think of a closed configuration $\cdot \vdash \mathcal{C} :: \Delta$ as a collection of processes that provide all the channels in Δ. As we have seen, the external interface Δ will never change during the computation. Moreover, when $\mathcal{C}$ is final, every process in $\mathcal{C}$ is trying to

communicate along a channel in Δ. Because our language is synchronous (both send and receive block), this means none of the processes in $\mathcal{C}$ can take a step.

The question is how do we observe the outcome of the computation? Unlike functional languages, the value is not presented to us a whole. For example, if we have a final configuration

$$\cdot \vdash \mathsf{proc}(P) :: (a : \mathsf{nat})$$

where $\mathsf{nat} = \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\}$, then we do not yet have any information except that the process P terminated.

So we need to *receive* a label from the channel a in order to observe the outcome. As soon as we interact with P, it will resume computation with its continuation until is once again blocks with a send.

An observation may actually change the type of the channel a at the interface. For example, if we received zero along a then afterwards we have $P' :: (a : \mathbf{1})$. If instead we received succ, then afterwards the continuation process P' will again provide $a : \mathsf{nat}$.

A similar interaction protocol holds for all positive types ($A \otimes B$, $\mathbf{1}$, $\oplus\{\ell : A_\ell\}_{\ell \in L}$). For a negative type like $\&\{\ell : A_\ell\}_{\ell \in L}$ the situation is different. As pointed out in the introduction, we cannot actually observe the process that is trying to receive along the channel it provides. The best thing we could do at this point is send it (separately) each of the labels ℓ in the set L and observe the continuation A_ℓ in each case. This strategy breaks down when we encounter $a : B \multimap A$ because we cannot possibly send it a channel B that explores all possible behaviors along a. For example, if $B = \mathsf{nat}$, there would be infinitely many.

This means when we encounter a channel of negative type we stop our observation process. An analogous decision is made in functional languages such as ML or Haskell: values of function types are simply not directly observable, although we can probe their behavior by applying them to different arguments.

The implementation of the **`exec`** `P` in MPASS observes the outcome of a computation just as described above and prints the observed messages. When a negative type is encountered it prints just a dash.

7 Refactoring the Dynamics

It is often convenient to treat all the send and receive actions in a uniform way. In order to support this, we can refactor the syntax and also the dynamics with the

following definitions.

Processes	P, Q	$::=$	$x_A \leftarrow P(x)\ ;\ Q(x)$	(cut)
		$\mid$	$\mathbf{fwd}\ x\ y$	(id)
		$\mid$	$\mathbf{send}\ x\ m\ ;\ P$	(positive right or negative left rules)
		$\mid$	$\mathbf{recv}\ x\ K$	(positive left or negative right rules)
		$\mid$	$\mathbf{call}\ p\ x\ \overline{y}$	(possibly recursive process p)
Messages	m	$::=$	$(\,)$	$(\mathbf{1})$
		$\mid$	k	$(\oplus, \&)$
		$\mid$	y	$(\otimes, \multimap)$
Continuations	K	$::=$	$(\,) \Rightarrow P$	$(\mathbf{1})$
		$\mid$	$(\ell \Rightarrow P_\ell)_{\ell \in L}$	$(\oplus, \&)$
		$\mid$	$(y \Rightarrow P(y))$	$(\otimes, \multimap)$

In the dynamics, we pass a message to a continuation $m \triangleright K$ to obtain a process.

$$\begin{array}{lcll} (\,) & \triangleright & ((\,) \Rightarrow P) & = & P \\ k & \triangleright & (\ell \Rightarrow P_\ell)_{\ell \in L} & = & P_k \qquad (k \in L) \\ b & \triangleright & (y \Rightarrow P(y)) & = & P(b) \end{array}$$

The computation rules then simplify.

$$\begin{array}{lcl} \mathsf{proc}(x_A \leftarrow P(x)\ ;\ Q(x)) & \longrightarrow & \mathsf{proc}(P(a)), \mathsf{proc}(Q(a)) \quad (a \text{ fresh}) \\ \mathsf{proc}(P(b)), \mathsf{proc}(\mathbf{fwd}\ a\ b) & \longrightarrow & \mathsf{proc}(P(a)) \\ \mathsf{proc}(\mathbf{send}\ a\ m\ ;\ P), \mathsf{proc}(\mathbf{recv}\ a\ K) & \longrightarrow & \mathsf{proc}(P), \mathsf{proc}(m \triangleright K) \\ \mathsf{proc}(\mathbf{call}\ p\ a\ \overline{b}) & \longrightarrow & \mathsf{proc}(P(a, \overline{b})) \\ & & \text{where } p\ (x : A)\ \overline{(y : B)} = P(x, \overline{y}) \in \Sigma \end{array}$$

There are some possible variations on the identity rules that are sometimes useful. For an implementation, for example, we might enforce that $P(b)$ actually tries to communicate along b so it is expecting to interact. There is also a symmetric rule to the given one where $\mathsf{proc}(\mathbf{fwd}\ a\ b)$ interacts with its client $\mathsf{proc}(P(a))$ to yield $\mathsf{proc}(P(b))$. Such variations are consistent with logical cut reduction but we are not forced to specialize or generalize the rule above.

References

Gerhard Gentzen. Die Widerspuchsfreiheit der reinen Zahlentheorie. *Mathematische Annalen*, 112:493–565, 1936. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 132–213, North-Holland, 1969.

Lecture Notes on Subtyping

15-836: Substructural Logics
Frank Pfenning

Lecture 8
September 21, 2023

1 Introduction

So far, we have always worked under the presupposition that the provider and client of a channel agree on its type. This is fundamentally inspired by the cut rule in logic, and it also seems necessary to ensure that all messages are properly understood. For example, the progress property would fail spectacularly if one process sends a label while the recipient expects a channel.

In this lecture we consider if we can loosen this restriction without violating progress and preservation. If we can, it might allow us to simplify some programs, or to capture more properties of the programs we write in their type.

As an introductory example, consider the following two types.

$$\begin{array}{l} \mathsf{nat} = \{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\} \\ \mathsf{pos} = \{\phantom{\mathsf{zero} : \mathbf{1},}\ \mathsf{succ} : \mathsf{nat}\} \end{array}$$

If we have a provider $- \vdash P :: (n : \mathsf{pos})$ and a client $(n : \mathsf{nat}) \vdash Q :: -$ then nothing can go wrong. P restricts itself to start with the message succ but the client does not have to be aware of this—it will simply not receive a first message zero. On the other hand, if we have $- \vdash P :: (n : \mathsf{nat})$ and $(n : \mathsf{pos}) \vdash Q :: -$ then things can go wrong immediately because P could send the label zero that Q is not expecting.

The notion of subtyping we consider here has been developed by Gay and Hole [2005]. Although for a different underlying programming language, the result is essentially the same, except that in their setting the roles of sender and receiver are reversed from ours.

2 Message Understood

The key to subtyping in the message-passing setting is to make sure that the recipient of a message is ready for every possible message it could receive. Semantically,

we define subtyping $A \leq B$ this way:

> *If* $\Delta \vdash P :: (a : A)$ *and* $A \leq B$ *and* $\Delta', a : B \vdash Q :: (c : C)$ *then every message along channel* a *is understood by the receiver.*

For our purposes of study here, we'd like the relation $A \leq B$ to be as large as possible. Or, to put it another way, if $A \not\leq B$ then there should be counterexample, that is, a message along the channel a that the recipient does not understand. By "does not understand" we mean that in the refactored rule from the last lecture

$$\mathsf{proc}(\mathbf{send}\ a\ m\ ;\ P), \mathsf{proc}(\mathbf{recv}\ a\ K) \longrightarrow \mathsf{proc}(P), \mathsf{proc}(m \triangleright K)$$

the operation $m \triangleright K$ is undefined.

In order to see what kind of subtyping might hold we walk through the critical steps in type preservation and progress in a hand-wavy fashion. These form the core of the proof of progress and preservation in the presence of subtyping. An important point here is to think connective by connective, so that it is open-ended and adaptable to other languages.

Before we get to the specifics, there are a few general properties we expect. We expect these to be *admissible* and rather than primitive.

- Subtyping should be reflexive: $A \leq A$ for all types A. This vaguely corresponds to identity.
- Subtyping should be transitive: if $A \leq B$ and $B \leq C$, then $A \leq C$. This vaguely corresponds to cut.
- Right subsumption should be admissible: If $\Delta \vdash P :: (a : A)$ and $A \leq B$ then also $\Delta \vdash P :: (a : B)$.
- Left subsumption should be admissible: If $A \leq B$ and $\Delta, b : B \vdash P :: (c : C)$ then $\Delta, b : A \vdash P :: (c : C)$

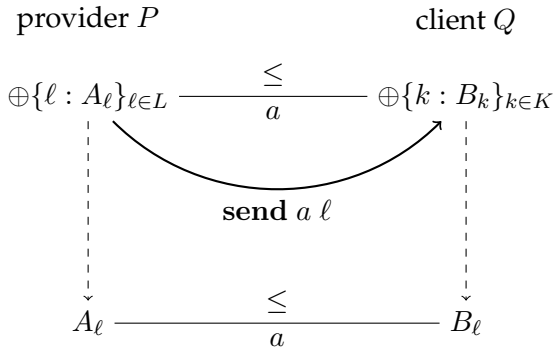
We now build up a set of rules that allow us to conclude $A \leq B$ as a judgment.

One thing we can say immediately based on the semantic definition: there should be no rules for $A \leq B$ if the top level type constructor of A and B is different. For example, $1 \not\leq A \otimes B$ and $A \multimap B \not\leq \oplus\{\ell : A_\ell\}_{\ell \in L}$. In all these cases, a message sent along the channel will not be understood by the recipient.

3 Internal Choice and Unit

As the example in the introduction suggests, $\oplus\{\ell : A_\ell\}_{\ell \in L} \leq \oplus\{k : B_k\}_{k \in K}$ requires that every label in L must also be in K. That's because the provider will send some $\ell \in L$ along channel a so the recipient must be ready for it. But that's not quite sufficient: after a label $\ell \in L$ is sent, the two processes will still be connected along

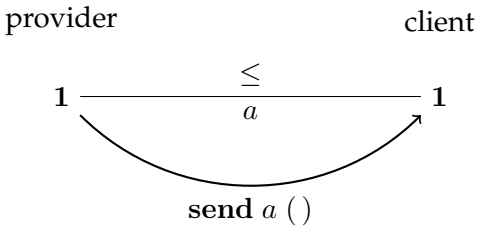
a, but now the provider will have type A_ℓ and the client B_ℓ, so one must be a subtype of the other. Pictorially:



From this we extract the rule

$$\frac{L \subseteq K \quad A_\ell \leq B_\ell \ (\forall \ell \in L)}{\oplus\{\ell : A_\ell\}_{\ell\in L} \leq \oplus\{k : B_k\}_{k\in K}}$$

We also have $\mathbf{1} \leq \mathbf{1}$ without any condition since the channel a is closed.



The corresponding rule

$$\frac{}{\mathbf{1} \leq \mathbf{1}}$$

has no premise because the unit message closes the channel.

At this point we can already show some examples. Since type definitions are equirecursive we just unfold them.

$$\dfrac{\dfrac{\mathsf{nat} \leq \mathsf{nat}}{\oplus\{\mathsf{succ} : \mathsf{nat}\} \leq \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\}}}{\mathsf{pos} \leq \mathsf{nat}}$$

Here we stopped at reflexivity. But if we think of reflexivity as just admissible, we

would continue:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{1 \leq 1} \quad \mathsf{nat} \leq \mathsf{nat}}{\oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\} \leq \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\}}}{\mathsf{nat} \leq \mathsf{nat}}}{\oplus\{\mathsf{succ} : \mathsf{nat}\} \leq \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\}}}{\mathsf{pos} \leq \mathsf{nat}}$$

At this point we realize we can continue indefinitely building a deeper and deeper derivation by expanding the recursive definition. But somehow that should be okay: if there is no subproof where we actually get stuck then there should be no "message not understood" problem when the processes communicate. So the proof system is *infinitary*. Even infinite derivations are sufficient to guarantee that there is no finite counterexample.

Another way to express this is to say that the proof rules here are interpreted *coinductively*. A proof is valid if we can always proceed further along all the open branches. This is in contrast to the proof systems we have seen so far, where proofs are defined *inductively*: we are only satisfied if we have a finite proof constructed from the rules.

In general, coinductive proofs system are more difficult to work with because we cannot actually write down infinite proofs. But they can still serve a useful purpose when they capture an intuitive notion. Here, and in some other cases I am aware of, they capture the absence of a counterexample. Constructively, this is not the same as a direct proof, but a refutation of its negation and therefore in some sense "weaker" than a (constructive inductive) proof.

In this particular example, we actually have a finitary representation of an infinitary proof since we have reached a cycle: the judgment at the top is the same as one lower in the same proof branch. In that case we can mark it as a loop and not explore this branch further. This way to proceed is sound since we could always unfold the looping proof into an infinite one. Or we can say that if there were a counterexample, there would be a shortest one. But the shortest one wouldn't go through the same judgment more than once.

In lecture we noted this with arcs, but in LaTeX we just label the lower judgment in the proof and then use this to justify the leaf.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{1 \leq 1} \quad \mathsf{nat} \leq \mathsf{nat}\ (x)}{\oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\} \leq \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\}}}{\mathsf{nat} \leq \mathsf{nat} \quad (x)}}{\oplus\{\mathsf{succ} : \mathsf{nat}\} \leq \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\}}}{\mathsf{pos} \leq \mathsf{nat}}$$

In this particular example it seems like we should have been able to avoid the loop altogether by promoting reflexivity to be a rule. In other example, this is not possible. For example:

$$\begin{array}{l}\mathsf{nat} = \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\} \\ \mathsf{even} = \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{odd}\} \\ \mathsf{odd} = \oplus\{\mathsf{succ} : \mathsf{even}\}\end{array}$$

We start to construct circular proof of $\mathsf{even} \leq \mathsf{nat}$:

$$\dfrac{\dfrac{}{\mathbf{1} \leq \mathbf{1}} \quad \begin{array}{c}\vdots \\ \mathsf{odd} \leq \mathsf{nat}\end{array}}{\dfrac{\oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{odd}\} \leq \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\}}{\mathsf{even} \leq \mathsf{nat}}}$$

We see that we have "reduced" the question of $\mathsf{even} \leq \mathsf{nat}$ to the question if $\mathsf{odd} \leq \mathsf{nat}$. We go on until we can complete all branches in the proof.

$$\dfrac{\dfrac{}{\mathbf{1} \leq \mathbf{1}} \quad \dfrac{\begin{array}{c}(x) \\ \mathsf{even} \leq \mathsf{nat}\end{array}}{\dfrac{\oplus\{\mathsf{succ} : \mathsf{even}\} \leq \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\}}{\mathsf{odd} \leq \mathsf{nat}}}}{\dfrac{\oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{odd}\} \leq \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\}}{\mathsf{even} \leq \mathsf{nat} \quad (x)}}$$

In order to explore a failure of subtyping, consider the judgment $\mathsf{nat} \leq \mathsf{even}$. Clearly, this should not be provable.

$$\dfrac{\dfrac{}{\mathbf{1} \leq \mathbf{1}} \quad \dfrac{\begin{array}{c}\text{fails, since } \{\mathsf{zero}, \mathsf{succ}\} \not\subseteq \{\mathsf{succ}\} \\ \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\} \leq \oplus\{\mathsf{succ} : \mathsf{even}\}\end{array}}{\mathsf{nat} \leq \mathsf{odd}}}{\dfrac{\oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\} \leq \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{odd}\}}{\mathsf{nat} \leq \mathsf{even}}}$$

Here we observe that this particular *failed* derivation contains enough information to extract a sequence of messages where the last one is not expected by the recipient: succ followed by zero. And, indeed, this sequence of message is (the start of) the number 1 which is not even.

4 Example: Binary Numbers in Standard Form

As another example we consider binary numbers. Their representation is not uniquely determined because of the possibility of leading zeros. For example, $(0100)_2 = (100)_2 = 4$. We say a number is in standard form if it has no leading zeros. We can define this using positive numbers as an additional type.

$$\begin{aligned}\mathsf{bin} &= \oplus\{\mathsf{b0} : \mathsf{bin}, \mathsf{b1} : \mathsf{bin}, \mathsf{e} : \mathbf{1}\}\\ \mathsf{std} &= \oplus\{\mathsf{b0} : \mathsf{pos}, \mathsf{b1} : \mathsf{std}, \mathsf{e} : \mathbf{1}\}\\ \mathsf{pos} &= \oplus\{\mathsf{b0} : \mathsf{pos}, \mathsf{b1} : \mathsf{std}\}\end{aligned}$$

Then $\mathsf{pos} \leq \mathsf{std}$:

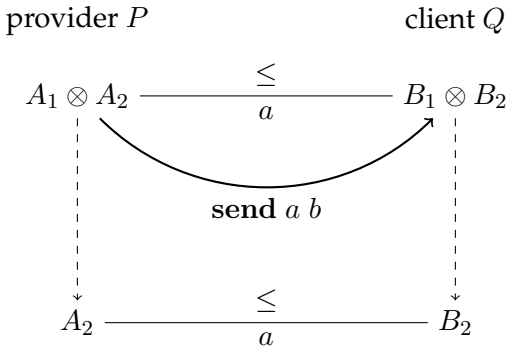
$$\dfrac{\dfrac{\dfrac{(x)}{\mathsf{pos} \leq \mathsf{pos}} \quad \vdots \quad \mathsf{std} \leq \mathsf{std}}{\oplus\{\mathsf{b0} : \mathsf{pos}, \mathsf{b1} : \mathsf{std}\} \leq \oplus\{\mathsf{b0} : \mathsf{pos}, \mathsf{b1} : \mathsf{std}\}} \quad \vdots}{\dfrac{\mathsf{pos} \leq \mathsf{pos} \quad (x) \qquad \mathsf{std} \leq \mathsf{std}}{\oplus\{\mathsf{b0} : \mathsf{pos}, \mathsf{b1} : \mathsf{std}\} \leq \oplus\{\mathsf{b0} : \mathsf{pos}, \mathsf{b1} : \mathsf{std}, \mathsf{e} : \mathbf{1}\}}}{\mathsf{pos} \leq \mathsf{std}}$$

This example is a (mild) illustration of a concern about circular proofs: we do not transfer what we learn on one branch to another. One technique to deal with this is to turn an infinitary (circular) proof system into a saturation procedure that works with forward inference. In such a system there is more reuse. DeYoung et al. [2023] then justify the saturating rules with respect to the infinitary rules.

In our system for subtyping (incomplete, at this point), attempts are constructing circular proofs will always either fail finitely or end up with a (finite) circular proof. The reason is that in a signature in which n syntactically different types occur, there can be at most n^2 pairs $A \leq B$ that might appear on a branch in the proof. This will continue to be the case when our set of rules is complete.

5 Tensor

Let's consider the interaction when $a : A_1 \otimes A_2$. The provider will send a channel $b : A_1$ and the channel a will afterwards have type A_2.



From this we see that $A_2 \leq B_2$ is required for the connection to continue to be well-typed. But what is the situation with the channel b, provided by, say R? We have the following chain of reasoning:

1. P was the original client of b at type A_1.
2. Q will be its new client of b at type B_1.
3. R provided the channel to P at some type C_1, so $C_1 \leq A_1$.

So for R and Q to be properly connected over the channel b we must have $A_1 \leq B_1$ because then $C_1 \leq B_1$ follows by transitivity.

Actually, we can be more lenient that what we just described. Provider P can send a channel $b : A_1'$ as long as $A_1' \leq A_1$. Then we get:

1. P was the original client of b at type A_1'.
2. Q is the new client of b at type B_1.
3. $A_1' \leq A_1$ (the condition for P to send b along $a : A_1 \otimes A_2$)
4. R is the provider at some type C_1, so $C_1 \leq A_1'$.

Still, $A_1 \leq B_1$ is sufficient to guarantee the chain of subtyping from the provider R to the new client Q: $C_1 \leq A_1' \leq A_1 \leq B_1$. Without the condition, if $C_1 = A_1' = A_1 \not\leq B_1$ an incorrect situation would arise, leading to the potential of a "message not understood" error along channel b later. So our rule is just:

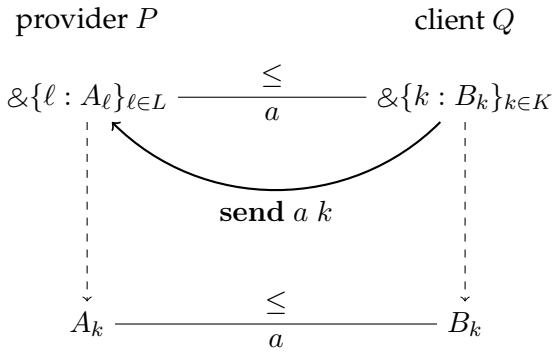
$$\frac{A_1 \leq B_1 \quad A_2 \leq B_2}{A_1 \otimes A_2 \leq B_1 \otimes B_2}$$

We can exploit subtyping as sketched above to generalize the $\otimes R^*$ rule.

$$\frac{A_1' \leq A_1 \quad \Delta \vdash P :: (x : A_2)}{\Delta, y : A_1' \vdash \mathbf{send}\ x\ y\ ;\ P :: (x : A_1 \otimes A_2)}\ \otimes R^*$$

6 Negative Types

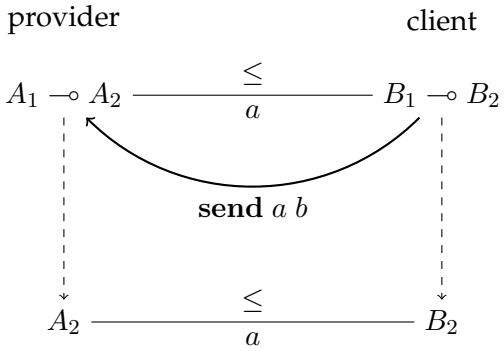
For negative types, the role of sender and receive are reversed, so we need to reexamine the situation carefully. We start with external choice, that should be analogous to internal choice.



We see that for the label k to be understood, we need that $k \in L$, so we must require that $L \supseteq K$, the opposite inclusion from the internal choice. However, provider and client remain the same, so the continuation types must be related in the same order.

$$\frac{L \supseteq K \quad A_k \leq B_k\ (\forall k \in K)}{\&\{\ell : A_\ell\}_{\ell \in L} \leq \&\{k : A_k\}_{k \in K}}$$

In the dynamics of linear implication a channel b is received by the *provider* along a.



The handoff of the channel b leads to the following reasoning.

1. A process R provides b at type $C_1 \leq B_1'$.
2. The client Q sees b at type $B_1' \leq B_1$.
3. The new client P sees b at type A_1.

So for the new connection to be well-typed, we need $C_1 \leq A_1$. We can get this by $C_1 \leq B_1' \leq B_1 \leq A_1$, so we should require $B_1 \leq A_1$. This is the manifestation of *contravariance of subtyping for arguments at function type* in this context.

$$\frac{B_1 \leq A_1 \quad A_2 \leq B_2}{A_1 \multimap A_2 \leq B_1 \multimap B_2}$$

We have already anticipated the generalization of the typing rule for sending a channel.

$$\frac{A_1' \leq A_1 \quad \Delta, x : A_2 \vdash P :: (z : C)}{\Delta, y : A_1', x : A_1 \multimap A_2 \vdash \mathbf{send}\ x\ y\ ;\ P :: (z : C)}\ \multimap L^*$$

We can look among the rules for opportunities for generalization. The only other place where types are compared for equality in the rules so far is the identity rule and, depending on how one looks at it, the cut rule. We generalize identity.

$$\frac{A' \leq A}{y : A' \vdash \mathsf{fwd}\ x\ y :: (x : A)}\ \mathsf{id}$$

You should convince yourself that this rule is correct by simple transitivity reasoning.

We do not generalize cut, because due to the admissibility of left and right subsumption, this would complicate the syntax without changing the set of well-typed processes. However, we do generalize the call rule (see Figure 2).

7 Example: Subtyping of Stores

As an example of subtyping with negative types we consider the store interface from before.

$$\begin{aligned}\mathsf{store} = \&\{\ &\mathsf{ins} : \mathsf{bin} \multimap \mathsf{store},\\ &\mathsf{del} : \oplus\{\ \mathsf{none} : \mathbf{1}, \mathsf{some} : \mathsf{bin} \otimes \mathsf{store}\ \}\ \}\end{aligned}$$

There are uses of a stack where it is important that we only insertions followed only by deletions until the stack is empty. For example, the amortized analysis of queues, implemented by two stacks, relies on a property along these lines [Okasaki, 1998].

This is an example of an interaction protocol with a data structure prescribed by a type. In object-oriented programming related techniques have been referred to as *typestate analysis* Strom and Yemini [1986].

We have two phases of communication, store^1 where we only insert (until the first deletion) and store^2 where we only delete. This is expressed in the following two types.

$$\begin{aligned}\mathsf{store}^1 &= \&\{\, \mathsf{ins} : \mathsf{bin} \multimap \mathsf{store}^1, \\ &\qquad \mathsf{del} : \oplus\{\, \mathsf{none} : \mathbf{1}, \mathsf{some} : \mathsf{bin} \otimes \mathsf{store}^2 \,\}\,\} \\ \mathsf{store}^2 &= \&\{\, \mathsf{del} : \oplus\{\, \mathsf{none} : \mathbf{1}, \mathsf{some} : \mathsf{bin} \otimes \mathsf{store}^2 \,\}\,\}\end{aligned}$$

What are the expected subtyping relationships between store, store^1 and store^2? We suggest you work this out for yourself before you read on.

Maybe, like me, you got it wrong and conjectured, for examples, that $\mathsf{store}^2 \le \mathsf{store}^1$. Let's see if we can prove this or find a counterexample.

$$\dfrac{\begin{array}{c}\text{fails, since } \{\mathsf{del}\} \not\supseteq \{\mathsf{del}, \mathsf{ins}\} \\ \&\{\mathsf{del} : \ldots\} \le \&\{\mathsf{del} : \ldots, \mathsf{ins} : \ldots\}\end{array}}{\mathsf{store}^2 \le \mathsf{store}^1}$$

Oops! And, indeed, if the client sees the type with both insert and delete options, it could send del. This message will not be understood by a provider in the phase 2, expecting only deletions.

So the relationship is the other way around. We skip some intermediate unfolding of type definitions. The missing part is a simple instance of reflexivity, which we have marked as an instance of an admissible rule.

$$\dfrac{\{\mathsf{ins}, \mathsf{del}\} \supseteq \{\mathsf{del}\} \qquad \dfrac{\cdots}{\oplus\{\,\mathsf{none} : \mathbf{1}, \mathsf{some} : \mathsf{bin} \otimes \mathsf{store}^2\,\} \le \oplus\{\,\mathsf{none} : \mathbf{1}, \mathsf{some} : \mathsf{bin} \otimes \mathsf{store}^2\,\}}}{\mathsf{store}^1 \le \mathsf{store}^2}$$

We also have $\mathsf{store} \le \mathsf{store}^1$ by a derivation that should not be surprising at this point.

$$\dfrac{\dfrac{\dfrac{\cdots}{\mathsf{bin} \le \mathsf{bin}} \qquad \dfrac{(x)}{\mathsf{store} \le \mathsf{store}^1}}{\mathsf{bin} \multimap \mathsf{store} \le \mathsf{bin} \multimap \mathsf{store}^1} \qquad \dfrac{\dfrac{}{\mathbf{1} \le \mathbf{1}} \qquad \dfrac{\dfrac{\cdots}{\mathsf{bin} \le \mathsf{bin}} \qquad \dfrac{\mathcal{D}}{\mathsf{store} \le \mathsf{store}^2}}{\mathsf{bin} \otimes \mathsf{store} \le \mathsf{bin} \otimes \mathsf{store}^2}}{\oplus\{\,\mathsf{none} : \mathbf{1}, \mathsf{some} : \mathsf{bin} \otimes \mathsf{store}\,\} \le \oplus\{\,\mathsf{none} : \mathbf{1}, \mathsf{some} : \mathsf{bin} \otimes \mathsf{store}^2\,\}}}{\mathsf{store} \le \mathsf{store}^1 \quad (x)}$$

$$\mathcal{D} = \dfrac{\dfrac{\dfrac{}{\mathbf{1} \le \mathbf{1}} \qquad \dfrac{\dfrac{\cdots}{\mathsf{bin} \le \mathsf{bin}} \qquad \dfrac{(y)}{\mathsf{store} \le \mathsf{store}^2}}{\mathsf{bin} \otimes \mathsf{store} \le \mathsf{bin} \otimes \mathsf{store}^2}}{\oplus\{\,\mathsf{none} : \mathbf{1}, \mathsf{some} : \mathsf{bin} \otimes \mathsf{store}\,\} \le \oplus\{\,\mathsf{none} : \mathbf{1}, \mathsf{some} : \mathsf{bin} \otimes \mathsf{store}^2\,\}}}{\mathsf{store} \le \mathsf{store}^2 \quad (y)}$$

8 Subtyping in MPASS

Subtyping is implemented in MPASS and will be used when it is called with `--subtyping`, or `-s` for short.

There is no separate declaration to test subtyping, but we can use forwarding because $x : A \vdash \mathbf{fwd}\ y\ x :: (y : B)$ is well-typed if and only if $A \le B$. Examples can be found in the file lecture8.mps; excerpts are in Listing 1 and Listing 2.

```
1   type bin = +{'b0 : bin, 'b1 : bin, 'e : 1}
2
3   type std = +{'b0 : pos, 'b1 : std, 'e : 1}
4   type pos = +{'b0 : pos, 'b1 : std        }
5
6   proc pos_std (y : std) (x : pos) = fwd y x  % pos <: std
7   proc std_bin (y : bin) (x : std) = fwd y x  % std <: bin
8
9   fail
10  proc bin_std (y : std) (x : bin) = fwd y x  % bin </: std
```

Listing 1: Subtyping for some subsets of binary numbers

The last declaration in Listing 2 illustrates how we can test that A is *not* a subtype of B. We do this by using the construct **`fail`** `<dec>` which succeeds if the declaration `<dec>` fails. If you run this through MPASS with the `-d` flag it will still show the error message it would have printed if the declaration were not preceded by **`fail`**.

The only example of testing subtyping on programs are the `list2store` and `store2list` processes. The first, only *inserts* numbers into a store, while the second only *deletes* them, so they use the store^1 and store^2 types from this lecture. You can find the code in Listing 2.

9 Summary

We summarize the rules for subtyping, interpreted coinductively, in Figure 1 and the updated rules for process typing in Figure 2. The other rules remain unchanged.

References

Henry DeYoung, Andreia Mordido, Frank Pfenning, and Ankush Das. Parametric subtyping for structural parametric polymorphism. *CoRR*, abs/2307.13661, July 2023. URL https://arxiv.org/abs/2307.13661. Submitted.

Simon J. Gay and Malcolm Hole. Subtyping for session types in the π-calculus. *Acta Informatica*, 42(2–3):191–225, 2005.

Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12 (1):157–171, 1986.

```
type list = +{'nil : 1, 'cons : bin * list}

type store =  &{'ins : bin -o store,
                'del : +{'none : 1, 'some : bin * store}}
type store1 = &{'ins : bin -o store1,
                'del : +{'none : 1, 'some : bin * store2}}
type store2 = &{'del : +{'none : 1, 'some : bin * store2}}

(* note use of 'store1' below! *)
proc list2store (s : store1) (l : list) (t : store1) =
     recv l ( 'nil => recv l (() => fwd s t)
            | 'cons => recv l (x =>
                       send t 'ins ;
                       send t x ;
                       call list2store s l t) )

(* note use of 'store2' below! *)
proc store2list (l : list) (s : store2) =
     send s 'del ;
     recv s ( 'none => recv s (() => send l 'nil ; send l ())
            | 'some => recv s (x => send l 'cons ; send l x ;
                                    call store2list l s) )

proc roundtrip (l : list) (k : list) =
     e <- call empty e ;           % start with empty store
     s <- call list2store s k e ;  % add all elements from k
     call store2list l s           % retrieve all element from s
```

Listing 2: Phase 1 and 2 store typing

$$\frac{L \subseteq K \quad A_\ell \leq B_\ell \ (\forall \ell \in L)}{\oplus\{\ell : A_\ell\}_{\ell \in L} \leq \oplus\{k : B_k\}_{k \in K}}$$

$$\frac{}{\mathbf{1} \leq \mathbf{1}} \qquad \frac{A_1 \leq B_1 \quad A_2 \leq B_2}{A_1 \otimes A_2 \leq B_1 \otimes B_2}$$

$$\frac{L \supseteq K \quad A_k \leq B_k \ (\forall k \in K)}{\&\{\ell : A_\ell\}_{\ell \in L} \leq \&\{k : A_k\}_{k \in K}} \qquad \frac{B_1 \leq A_1 \quad A_2 \leq B_2}{A_1 \multimap A_2 \leq B_1 \multimap B_2}$$

Figure 1: Subtyping, rules interpreted coinductively

$$\frac{A' \leq A}{y : A' \vdash \mathsf{fwd}\ x\ y :: (x : A)}\ \mathsf{id}$$

$$\frac{f\ (x : A')\ \overline{(y_i : B'_i)} = P \in \Sigma \quad B_i \leq B'_i\ (\forall i) \quad A' \leq A}{\overline{y_i : B_i} \vdash \mathbf{call}\ f\ x\ \overline{y_i} :: (x : A)}\ \mathsf{call}$$

$$\frac{A'_1 \leq A_1 \quad \Delta \vdash P :: (x : A_2)}{\Delta, y : A'_1 \vdash \mathbf{send}\ x\ y\ ;\ P :: (x : A_1 \otimes A_2)}\ \otimes R^*$$

$$\frac{A'_1 \leq A_1 \quad \Delta, x : A_2 \vdash P :: (z : C)}{\Delta, y : A'_1, x : A_1 \multimap A_2 \vdash \mathbf{send}\ x\ y\ ;\ P :: (z : C)}\ \multimap L^*$$

Figure 2: Process typing, extended for subtyping

Lecture Notes on Validity

15-836: Substructural Logics
Frank Pfenning

Lecture 9
September 26, 2023

1 Introduction

So far we have drawn strict boundaries between ordered, linear, and structural logic. To make linear logic more expressive we have used recursion because it is quite natural from the programming perspective. In Lecture 2 we briefly mentioned and showed rules for the exponential modality $!A$ which is subject to weakening and contraction.

In this lecture we explain that $!A$ is the result of a general construction that can be carried out for other logics as well. In structural logics, for example, it is usually written as $\Box A$, expressing that A is necessarily true or A is true in all possible worlds. The proof theory for $!A$ when viewed from this perspective is somewhat more pleasant, but ultimately still not entirely satisfactory. We will come back to this in the next lecture to improve and in the process generalize it even further.

2 Girard's Exponential

Girard's [1987] exponential modality, in the intuitionistic setting [Girard and Lafont, 1987], can be defined in the sequent calculus with the following rules.

$$\frac{!\Delta \vdash A}{!\Delta \vdash !A}\ !R \qquad \frac{\Delta, A \vdash C}{\Delta, !A \vdash C}\ !L$$

$$\frac{\Delta, !A, !A \vdash C}{\Delta, !A \vdash C}\ \text{contract} \qquad \frac{\Delta \vdash C}{\Delta, !A \vdash C}\ \text{weaken}$$

Here, $!\Delta$ means that every antecedent in Δ has the form $!B$. With these rules we can obtain as many copies of A from $!A$ as we want.

3 Andreoli's Exponential

Andreoli [1992] introduced what he calls a *dyadic* system for linear logic where we have two distinct forms of antecedents (later also given in its intuitionistic version [Barber, 1996]). From our perspective, one collection of antecedents is *structural* and the other is *linear*. We write such a sequent as

$$\Gamma \,;\, \Delta \vdash A$$

Here, Γ represents a set and Δ a multiset. For Andreoli, this was mostly a technical device; we will justify it as the result of studying *validity*.

What does it mean for a proposition A to be *valid* as opposed to merely *true*? A slogan from an earlier lecture may be helpful:

> Truth is ephemeral, validity forever.

A proposition such as "it is raining" may be true in a particular state and false in others, while a proposition such as "A implies A" should be true for all propositions A in all states. In linear logic we can capture this with

$$\frac{\cdot \vdash A\ \mathit{true}}{\vdash A\ \mathit{valid}}$$

It expresses that if A is true without using any hypotheses, then A is valid.

From the perspective of linear logic, A being valid means we should be able to produce as many proofs of A as we wish. This in turn allows us to use A as many times as we wish in a proof. After all, a proof of A requires no resources. Using cut:

$$\frac{\cdot \vdash A \quad \Delta, A \vdash C}{\Delta \vdash C}\ \mathsf{cut}$$

So we should treat *antecedents* $A\ \mathit{valid}$ as *structural*.

The judgment we end up with has the form anticipated at the beginning of this section.

$$\underbrace{\Gamma}_{\mathit{valid}} \,;\, \underbrace{\Delta}_{\mathit{true}} \vdash A\ \mathit{true}$$

An interesting property of this formulation is that we do not change (yet) our language of propositions at all: they are all linear, with the usual linear connectives. This means that propositional inference rules are applied only to the succedent $A\ \mathit{true}$ and the antecedents in Δ, not those in Γ.

Once we have the judgment $A\ \mathit{valid}$ we also obtain a second judgment

$$\underbrace{\Gamma}_{\mathit{valid}} \vdash A\ \mathit{valid}$$

but there is only a single rule that applies here because the meaning of the connectives arises entirely from their linear nature. An antecedent $A\ \mathit{valid}$ allows us to use it whenever we wish.

$$\frac{\Gamma\ ;\ \cdot \vdash A\ \mathit{true}}{\Gamma \vdash A\ \mathit{valid}}\ \mathsf{valid}_R \qquad \frac{\Gamma, A\ \mathit{valid}\ ;\ \Delta, A\ \mathit{true} \vdash C\ \mathit{true}}{\Gamma, A\ \mathit{valid}\ ;\ \Delta \vdash C\ \mathit{true}}\ \mathsf{valid}_L$$

These rules are not regular right or left rules, because validity is a judgment, not a proposition. Just to remind ourselves of this we write R and L as a subscript.

In some ways these rules are similar to cut and identity in the sense that they apply to arbitrary propositions A. So rather than defining the nature of the connectives, they define the nature of the judgments. Cut and identity explain the nature of the hypothetical judgment, while valid_R and valid_L explain the nature of validity.

The next question is about how to express validity, internally, as a proposition. At this point this has become easy!

$$\frac{\Gamma \vdash A\ \mathit{valid}}{\Gamma\ ;\ \cdot \vdash\ !A\ \mathit{true}}\ !R \qquad \frac{\Gamma, A\ \mathit{valid}\ ;\ \Delta \vdash C\ \mathit{true}}{\Gamma\ ;\ \Delta, !A \vdash C\ \mathit{true}}\ !L$$

Since there is only one rule to conclude $\Gamma \vdash A\ \mathit{valid}$ (namely valid_R) the system is a little less symmetric but a little more streamlined if we combine $!R$ with valid_R into the rule

$$\frac{\Gamma\ ;\ \cdot \vdash A\ \mathit{true}}{\Gamma\ ;\ \cdot \vdash\ !A\ \mathit{true}}\ !R'$$

As discussed in lecture, we couldn't allow a nonempty Δ in the conclusion of $!R'$ (or $!R$, for that matter) because in the premise it must definitely be empty, and then none of the supposedly linear antecedents in Δ would actually be used.

As one might expect, things *do* go horribly wrong without this restriction. Since Γ is structural, it is just added parametrically to all the right and left rules and we have and also allowed for cut and identity.

$$\frac{}{\Gamma\ ;\ A \vdash A}\ \mathsf{id} \qquad \frac{\Gamma\ ;\ \Delta \vdash A \quad \Gamma\ ;\ \Delta' \vdash C}{\Gamma\ ;\ \Delta, \Delta' \vdash C}\ \mathsf{cut}$$

Without the restriction on $!R'$, we could prove:

$$\dfrac{\dfrac{\dfrac{}{\cdot\ ;\ A \vdash A}\ \mathsf{id}}{\cdot\ ;\ A \vdash\ !A}\ !R'??}{\cdot\ ;\ A \vdash A \otimes A}\ \mathsf{cut} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A\ ;\ A \vdash A}\ \mathsf{id}}{A\ ;\ \cdot \vdash A}\ \mathsf{valid}_L \quad \dfrac{\dfrac{}{A\ ;\ A \vdash A}\ \mathsf{id}}{A\ ;\ \cdot \vdash A}\ \mathsf{valid}_L}{A\ ;\ \cdot \vdash A \otimes A}\ \otimes R}{\cdot\ ;\ !A \vdash A \otimes A}\ !L$$

and similarly $\cdot\ ;\ A \vdash 1$. For these it is an easy step to show that weakening and contraction for all *linear* antecedents would be admissible. In other words, the logic would no longer be linear!

4 Examples

As mentioned in the introduction, the construction of validity is quite generic. For example, it could be carried out even if the base logic were already structural, in which case we obtain a version of the intuitionistic modal logic S4 [Pfenning and Davies, 2001], where $!A$ would be written $\Box A$. So we can test some laws of modal logic here. The first three judgments below are derivable; the last one isn't.

$$\vdash\ !(A \multimap B) \multimap (!A \multimap !B)$$
$$\vdash\ !A \multimap A$$
$$\vdash\ !A \multimap !!A$$
$$\not\vdash\ P \multimap !P$$

Let's write out the first one, which indicates that linear logic is a "normal" modal logic because the exponential distributes over implication.

$$\dfrac{\dfrac{\vdots}{\cdot\ ;\ !(A \multimap B), !A \vdash !B}}{\cdot\ ;\ \cdot \vdash !(A \multimap B) \multimap (!A \multimap !B)}\ \multimap R \times 2$$

At this point we cannot apply $!R$ because there are linear antecedents, so we have to shuffle them into the structural antecedents and then apply $!R'$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{A \multimap B, A\ ;\ \cdot \vdash B}}{A \multimap B, A\ ;\ \cdot \vdash !B}\ !R'}{\cdot\ ;\ !(A \multimap B), !A \vdash !B}\ !L \times 2}{\cdot\ ;\ \cdot \vdash !(A \multimap B) \multimap (!A \multimap !B)}\ \multimap R \times 2$$

Now we can copy $A \multimap B$ to the linear context since we would like to apply a left rule to it. The premises of $\multimap L$ then follow readily.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \multimap B, A\ ;\ A \vdash A}\ \mathsf{id}}{A \multimap B, A\ ;\ \cdot \vdash A}\ \mathsf{valid}_L \qquad \dfrac{}{A \multimap B, A\ ;\ B \vdash B}\ \mathsf{id}}{A \multimap B, A\ ;\ A \multimap B \vdash B}\ \multimap L}{A \multimap B, A\ ;\ \cdot \vdash B}\ \mathsf{valid}_L}{A \multimap B, A\ ;\ \cdot \vdash !B}\ !R'}{\cdot\ ;\ !(A \multimap B), !A \vdash !B}\ !L \times 2}{\cdot\ ;\ \cdot \vdash !(A \multimap B) \multimap (!A \multimap !B)}\ \multimap R \times 2$$

This illustrates a practical shortcoming of this system: since right and left rules are applied only to linear propositions, we frequently have to move structural propositions into and out of the linear antecedents, using valid_L and $!L$.

We can consider other questions. For example, does the exponential distribute over the tensor? Let's try:

$$\dfrac{\dfrac{\vdots}{A \otimes B \,;\, \cdot \vdash\, !A \otimes !B}}{\cdot \,;\, !(A \otimes B) \vdash\, !A \otimes !B}\, !L$$

At this point we could try $\otimes R$ or valid_L. After $\otimes R$ there are two remaining symmetric subgoals.

$$\dfrac{\dfrac{\dfrac{\text{fails}}{A \otimes B \,;\, \cdot \vdash\, A}}{A \otimes B \,;\, \cdot \vdash\, !A}\, !R' \qquad \dfrac{\vdots}{A \otimes B \,;\, \cdot \vdash\, !B}}{\dfrac{A \otimes B \,;\, \cdot \vdash\, !A \otimes !B}{\cdot \,;\, !(A \otimes B) \vdash\, !A \otimes !B}\, !L}\, \otimes R$$

We can prove neither of them, because whenever we copy $A \otimes B$ into the linear zone, followed by $\otimes L$, we get both A and B, linearly, but we only have A to prove.

If we try valid_L first, we also get stuck because we have linear A and B but the ultimately succedents are $!A$ and $!B$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{fails}}{A \otimes B \,;\, A \vdash\, !A} \qquad \dfrac{\text{fails}}{A \otimes B \,;\, B \vdash\, !B}}{A \otimes B \,;\, A, B \vdash\, !A \otimes !B}\, \otimes R}{A \otimes B \,;\, A \otimes B \vdash\, !A \otimes !B}\, \otimes L}{A \otimes B \,;\, \cdot \vdash\, !A \otimes !B}\, \mathsf{valid}_L}{\cdot \,;\, !(A \otimes B) \vdash\, !A \otimes !B}\, !L$$

The failure of these attempts doesn't mean much, but since this logic satisfies cut and identity elimination (see Section 6) it takes just a little more work to show that these are in fact not provable.

Perhaps we should have even seen intuitively that this entailment does not hold. It says that if we have both A and B together, as many times as we want, we can get, independently, A as often as we want and B as often as we want. That just couldn't be true.

But the exponential distributes over the additive conjunction $A \,\&\, B$ in an interesting way. Intuitively, $!(A \,\&\, B)$ means that we arbitrarily often have a choice between A and B. and $!A \otimes !B$ means that we have both A and B arbitrarily often,

separately. These are equivalent.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \,\&\, B \;;\, A \vdash A}\,\mathsf{id}}{A \,\&\, B \;;\, A \,\&\, B \vdash A}\,\&L_1}{A \,\&\, B \;;\, \cdot \vdash A}\,\mathsf{valid}_L}{A \,\&\, B \;;\, \cdot \vdash\, !A}\,!R' \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A \,\&\, B \;;\, B \vdash B}\,\mathsf{id}}{A \,\&\, B \;;\, A \,\&\, B \vdash B}\,\&L_2}{A \,\&\, B \;;\, \cdot \vdash B}\,\mathsf{valid}_L}{A \,\&\, B \;;\, \cdot \vdash\, !B}\,!R'}{\dfrac{A \,\&\, B \;;\, \cdot \vdash\, !A \otimes\, !B}{\cdot \;;\, !(A \,\&\, B) \vdash\, !A \otimes\, !B}\,!L}\,\otimes R$$

The other direction, $\cdot \;;\, !A \otimes\, !B \vdash\, !(A \,\&\, B)$ is perhaps even more straightforward.

5 Translation from Structural into Linear Logic

The whole endeavor of linear logic is to add expressive power to structural logic. This is clearly not the case without either recursion (which jeopardizes the logical reading altogether) or the exponential. Now that we have validity (which is structural) and the exponential modality, how do we translate ordinary intuitionistic logic into linear logic?

There seem to be fundamentally two translations, one "by name" and one "by value". They are so named because of what they mean operationally, under a functional interpretation. Girard [1987] provides a "by value" translation, so we develop that.

The basic idea guiding the translation $(A)^\vee$ is that if $\Gamma \vdash A$ then $(\Gamma)^\vee \;;\, \cdot \vdash (A)^\vee$. Using one of the judgments we have introduced, we could also have said $(\Gamma)^\vee \vdash (A)^\vee \; \mathit{valid}$. This should not be so surprising. The other direction is generally easy because we can just ignore the strictures of linearity.

Now we examine a few connectives in turn to see how they should translate.

$$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B}\,\supset R \qquad \begin{array}{c} \Gamma^\vee, A^\vee \;;\, \cdot \vdash B^\vee \\ \vdots \\ \Gamma^\vee \;;\, \cdot \vdash (A \supset B)^\vee \end{array}$$

We see that at least for the right rule, we can pick

$$(A \supset B)^\vee = !A^\vee \multimap\, !B^\vee$$

and then apply $!L$ and $!R'$ after $\multimap R$.

$$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B}\,\supset R \qquad \dfrac{\dfrac{\dfrac{\Gamma^\vee, A^\vee \;;\, \cdot \vdash B^\vee}{\Gamma^\vee, A^\vee \;;\, \cdot \vdash\, !B^\vee}\,!R'}{\Gamma^\vee \;;\, !A^\vee \vdash\, !B^\vee}\,!L}{\Gamma^\vee \;;\, \cdot \vdash\, !A^\vee \supset\, !B^\vee}\,\multimap R$$

What about the left rule?

$$\frac{\Gamma, A \supset B \vdash A \quad \Gamma, A \supset B, B \vdash C}{\Gamma, A \supset B \vdash C}\ \supset L \qquad \frac{\begin{array}{c}\Gamma^\vee, !A^\vee \multimap !B^\vee \,;\, \cdot \vdash A^\vee \quad \Gamma^\vee, !A^\vee \multimap !B^\vee, B^\vee \,;\, \cdot \vdash C^\vee \\ \vdots \\ \Gamma^\vee, !A^\vee \multimap !B^\vee \,;\, \cdot \vdash C^\vee\end{array}}{}$$

Note that because Γ and $\Gamma^\vee$ are both structural, we propagate them to all premises. Then we can just copy $!A^\vee \multimap !B^\vee$ to the linear antecedent, and apply the left rule, followed by some more administrative moves.

$$\frac{\dfrac{\dfrac{\Gamma^\vee, !A^\vee \multimap !B^\vee \,;\, \cdot \vdash A^\vee}{\Gamma^\vee, !A^\vee \multimap !B^\vee \,;\, \cdot \vdash !A^\vee}\ !R \quad \dfrac{\Gamma^\vee, !A^\vee \multimap !B^\vee, B^\vee \,;\, \cdot \vdash C^\vee}{\Gamma^\vee, !A^\vee \multimap !B^\vee \,;\, !B^\vee \vdash C^\vee}\ !L}{\Gamma^\vee, !A^\vee \multimap !B^\vee \,;\, !A^\vee \multimap !B^\vee \vdash C^\vee}\ \multimap L}{\Gamma^\vee, !A^\vee \multimap !B^\vee \,;\, \cdot \vdash C^\vee}\ \mathsf{valid}L$$

All the other connectives follow similar patterns, but let's also look at identity and cut.

$$\frac{}{\Gamma, A \vdash A}\ \mathsf{id} \qquad \frac{\dfrac{}{\Gamma^\vee, A^\vee \,;\, A^\vee \vdash A^\vee}\ \mathsf{id}}{\Gamma^\vee, A^\vee \,;\, \cdot \vdash A^\vee}\ \mathsf{valid}_L$$

$$\frac{\Gamma \vdash A \quad \Gamma, A \vdash C}{\Gamma \vdash C}\ \mathsf{cut} \qquad \frac{\begin{array}{c}\Gamma^\vee \,;\, \cdot \vdash A^\vee \quad \Gamma^\vee, A^\vee \,;\, \cdot \vdash C^\vee \\ \vdots \\ \Gamma^\vee \,;\, \cdot \vdash C^\vee\end{array}}{}$$

We can derive the conclusion in the case of a cut by introducing and then cutting $!A^\vee$

$$\frac{\dfrac{\dfrac{\Gamma^\vee \,;\, \cdot \vdash A^\vee}{\Gamma^\vee \,;\, \cdot \vdash !A^\vee}\ !R' \quad \dfrac{\Gamma^\vee, A^\vee \,;\, \cdot \vdash C^\vee}{\Gamma^\vee \,;\, !A^\vee \vdash C^\vee}\ !L}{\Gamma^\vee \,;\, \cdot \vdash C^\vee}\ \mathsf{cut}$$

As we will see in Section 6 is also makes sense to extend linear logic with additional rule that cuts valid antecedents directly.

$$\frac{\Gamma^\vee \,;\, \cdot \vdash A^\vee \quad \Gamma^\vee, A^\vee \,;\, \cdot \vdash C^\vee}{\Gamma^\vee \,;\, \cdot \vdash C^\vee}\ \mathsf{cut}!$$

To completing the translation we map the intuitionistic (structural) connectives to their linear counterparts and prefix every subformula with an exponential. In the reverse direction $A^\wedge$ we just map all linear connectives to their structural coun-

terparts and drop all exponentials.

$$
\begin{array}{lcllcl}
(A \supset B)^\vee & = & !A^\vee \multimap !B^\vee & (A \multimap B)^\wedge & = & A^\wedge \supset B^\wedge \\
(A \wedge B)^\vee & = & !A^\vee \otimes !B^\vee & (A \otimes B)^\wedge & = & A^\wedge \wedge B^\wedge \\
 & & & (A \mathbin{\&} B)^\wedge & = & A^\wedge \wedge B^\wedge \\
(\top)^\vee & = & \mathbf{1} & (\mathbf{1})^\wedge & = & \top \\
 & & & (\top)^\wedge & = & \top \\
(A \vee B)^\vee & = & !A^\vee \oplus !B^\vee & (A \oplus B)^\wedge & = & A^\wedge \vee B^\wedge \\
(\bot)^\vee & = & \mathbf{0} & (\mathbf{0})^\wedge & = & \bot \\
 & & & (!A)^\wedge & = & A^\wedge \\
(P)^\vee & = & P & (P)^\wedge & = & P
\end{array}
$$

We can summarize the correctness of the translation in the following theorem.

Theorem 1 (Correctness of Translation from Structural to Linear Logic)

(i) If $\Gamma \vdash A$ *then* $\Gamma^\vee \,;\, \cdot \vdash A^\vee$

(ii) If $\Gamma \,;\, \Delta \vdash A$ *then* $\Gamma^\wedge, \Delta^\wedge \vdash A^\wedge$

Proof: Part (i) follows by structural induction over the given derivation. In each case we directly construct the resulting derivation, preserving the essential structure while inserting rules concerning validity and the exponential. We showed some representative cases in this section.

Part (ii) also follows by structural induction over the given derivation. Some structural antecedents available for $\Gamma^\wedge, \Delta^\wedge \vdash A^\wedge$ will be unnecessary and can be dropped by weakening. □

There is an optimized translation where the subformulas of positive propositions ($\otimes$, $\oplus$) are not preceded by an exponential. I suspect the most straightforward way to prove the correctness of the optimized translation is to prove inversion of the left rules on the structural side and then mimic them with the linear left rules (which also happen to be invertible).

6 Cut and Identity Elimination[1]

Both cut and identity elimination carry over from the purely linear case, but with a few new wrinkles.

[1] not covered in lecture

Theorem 2 (Admissibility of Identity) *If we restrict the identity to atomic propositions, then*

$$\dfrac{}{\Gamma\ ;\ A \vdash A}\ \mathsf{id}_A$$

is admissible for arbitrary A.

Proof: As before, by structural induction on A. The only interesting case is $A = !A'$. We construct:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, A'\ ;\ A' \vdash A'}\ \mathsf{id}_{A'}}{\Gamma, A'\ ;\ \cdot \vdash A'}\ \mathsf{valid}_L}{\Gamma, A'\ ;\ \cdot \vdash\ !A'}\ !R}{\Gamma\ ;\ !A' \vdash\ !A'}\ !L$$

□

In order to prove admissibility of cut, it is helpful to simultaneously proof the admissibility of cut!. The induction measure is then somewhat more complicated, as we explain below. The first premise of the cut! rule expresses that A is valid, so we can cut out an antecedent of the form $A\ \mathit{valid}$ from the second premise.

Theorem 3 (Admissibility of Cut) *The rules*

$$\dfrac{\Gamma\ ;\ \Delta \vdash A \quad \Gamma\ ;\ \Delta' \vdash C}{\Gamma\ ;\ \Delta, \Delta' \vdash C}\ \mathsf{cut} \qquad \dfrac{\Gamma\ ;\ \cdot \vdash A \quad \Gamma, A\ ;\ \Delta' \vdash C}{\Gamma\ ;\ \Delta' \vdash C}\ \mathsf{cut!}$$

are admissible.

Proof: By a simultaneous nested induction in the following order

(1) the structure of the proposition A

(2) $\mathsf{cut!}_A$ is greater than cut_A

(3) either the first or the second derivation becomes smaller while the other remains the same

Item (2) is new here and necessary for the following case.

Case:

$$\dfrac{\mathcal{D}\qquad \dfrac{\mathcal{E}'}{\dfrac{\Gamma, A\ ;\ \Delta', A \vdash C}{\Gamma, A\ ;\ \Delta' \vdash C}\ \mathsf{valid}_L}}{\Gamma\ ;\ \cdot \vdash A \qquad \Gamma\ ;\ \Delta' \vdash C}\ \mathsf{cut!}_A$$

We have to construct a new derivation with two cuts, because there are now two copies of A among the antecedents of $\mathcal{E}'$.

$$\dfrac{\dfrac{\mathcal{D}}{\Gamma\,;\cdot\vdash A} \quad \dfrac{\mathcal{E}'}{\Gamma, A\,;\Delta', A\vdash C}\,\mathsf{cut!}_A}{\Gamma\,;\Delta'\vdash C}\,\mathsf{cut}_A$$

The problem here is that both cut_A and $\mathsf{cut!}_A$ are on the same cut formula A. Also, the derivation of the second premise of cut_A may be much larger than the original $\mathcal{E}$, since it is the result of the induction hypothesis on $\mathsf{cut!}_A$. So we need that cut_A is strictly smaller than $\mathsf{cut!}_A$. Fortunately, the other critical case (which necessitates $\mathsf{cut!}_A$ in the first place) requires an appeal to the induction hypothesis at a smaller proposition.

Case:

$$\dfrac{\dfrac{\dfrac{\mathcal{D}'}{\Gamma\,;\cdot\vdash A'}\,!R}{\Gamma\,;\cdot\vdash\,!A'} \quad \dfrac{\dfrac{\mathcal{E}'}{\Gamma, A'\,;\Delta'\vdash C}}{\Gamma\,;\Delta',!A'\vdash C}\,!L}{\Gamma\,;\Delta'\vdash C}\,\mathsf{cut}_{!A'}$$

We reduce this immediately to a $\mathsf{cut!}_{A'}$, which is a smaller formula. So even if $\mathsf{cut!}_A$ is greater than cut_A, the structure of the proposition takes precedence.

$$\dfrac{\dfrac{\mathcal{D}'}{\Gamma\,;\cdot\vdash A'} \quad \dfrac{\mathcal{E}'}{\Gamma, A'\,;\Delta'\vdash C}\,\mathsf{cut!}_{A'}}{\Gamma\,;\Delta'\vdash C}$$

□

References

Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):197–347, 1992.

Andrew Barber. Dual intuitionistic linear logic. Technical Report ECS-LFCS-96-347, Department of Computer Science, University of Edinburgh, September 1996.

Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

Jean-Yves Girard and Yves Lafont. Linear logic and lazy computation. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, volume 2, pages 52–66, Pisa, Italy, March 1987. Springer-Verlag LNCS 250.

Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999.

Lecture Notes on
A Mixed Linear/Nonlinear Logic

15-836: Substructural Logics
Frank Pfenning

Lecture 10
September 28, 2023

1 Introduction

As we have seen in the last lecture, introducing the judgment of validity and then internalizing it as the exponential modality $!A$ permits a compositional translation of structural (intuitionistic) logic into linear logic. At this point we could declare victory and see if similar techniques apply to other logics, e.g., if there is some embedding of linear logic into ordered logic.

However, there are some nagging issues, despite the elegance of the cut elimination proof.

1. The exponential $!A$ is neither positive nor negative in that it is invertible neither on the right nor on the left. On the right, in a judgment $\Gamma\ ;\ \Delta \vdash\ !A$ we may have to wait until Δ becomes empty before applying $!R$. On the left, we can move the linear antecedent $!A\ \mathit{true}$ to $A\ \mathit{valid}$, but we cannot necessarily move it back immediately to $A\ \mathit{true}$ because that renders the linear context nonempty (thereby possibly preventing $!R$).

2. If we have a particular interpretation of intuitionistic (structural) logic in mind, then translation to linear logic may mangle this interpretation. Among other concerns, a notion of observability may not be preserved, in which case the translation couldn't properly serve as a compiler.

So it is worth looking for a direct combination of logics, rather than translating one into the other. If structural logic represents functional programming, and linear logic message-passing, then we'd look for a way to combine functional and message-passing programming.

Today, we take a first step in this direction by developing a mixed linear/nonlinear logic (mostly) following Benton [1994]. Since ours is a minor variant, we also call it LNL. We will see that it repairs some of the noted issues. Moreover, it

presents a clear path towards further generalization in the form of *adjoint logic* [Reed, 2009, Pruiksma et al., 2018].

2 Shifting Between Logics

Given the desire to keep the native meaning of both structural and linear logic, we place them in two different strata and speculate that we may just go back and forth between them using two shift operators.

$$\begin{array}{lrcl} \text{Structural} & A_{\mathsf{S}} & ::= & P_{\mathsf{S}} \mid A_{\mathsf{S}} \supset B_{\mathsf{S}} \mid A_{\mathsf{S}} \wedge B_{\mathsf{S}} \mid \top \mid A_{\mathsf{S}} \vee B_{\mathsf{S}} \mid \bot \mid \uparrow A_{\mathsf{L}} \\ \text{Linear} & A_{\mathsf{L}} & ::= & P_{\mathsf{L}} \mid A_{\mathsf{L}} \multimap B_{\mathsf{L}} \mid A_{\mathsf{L}} \otimes B_{\mathsf{L}} \mid \mathbf{1} \mid A_{\mathsf{L}} \mathbin{\&} B_{\mathsf{L}} \mid \top \mid A_{\mathsf{L}} \oplus B_{\mathsf{L}} \mid \mathbf{0} \mid \downarrow A_{\mathsf{S}} \end{array}$$

We call S and L *modes*.

The key questions are: Which properties do we expect from the combination, and which laws should the two shifts satisfy so that these properties hold? At least, we the combination should satisfy cut and identity elimination. Beyond that, the combination should be *conservative* over each fragment in a strong sense: not only do we want purely structural or purely linear propositions to be true precisely if they are true in purely structural or purely linear logic, but we also want them to have essentially the same proofs.

A lesson from the study of validity is that we cannot allow validity (here: truth of a structural proposition) to depend on truth (here: truth of a linear proposition). This gives us two judgment forms for the mixed linear/nonlinear logic.

(1) $\Gamma_{\mathsf{S}} \,;\, \Delta_{\mathsf{L}} \vdash A_{\mathsf{L}}$

(2) $\Gamma_{\mathsf{S}} \vdash A_{\mathsf{S}}$

The significant differences to the judgments from dual intuitionistic linear logic is that (a) there no longer is an exponential $!A$, and (b) we can apply inference rules directly to structural propositions A_{S}, both in the antecedent and in the succedent.

3 Rules for Implication

Because we have two judgments and also two forms of implication, there are more rules concerning implication than one might at first suspect. This kind of redundancy is unfortunate, but, as we will see in the next lecture, it can be eliminated to obtain a quite streamlined system in which there are just a single right rule and a single left rule for implication.

Because we can tell, by notation and by position, when antecedents are structural or linear propositions, we omit the subscription Γ_{S} and Δ_{L}.

First, the right rules. Since the mode of the succedent is uniquely determined by the judgment (or vice versa, depending on how you look at it), there are just two rules.

$$\frac{\Gamma, A_{\mathsf{S}} \vdash B_{\mathsf{S}}}{\Gamma \vdash A_{\mathsf{S}} \supset B_{\mathsf{S}}}\ \supset R \qquad \frac{\Gamma\ ;\ \Delta, A_{\mathsf{L}} \vdash B_{\mathsf{L}}}{\Gamma\ ;\ \Delta \vdash A_{\mathsf{L}} \multimap B_{\mathsf{L}}}\ \multimap R$$

There is only one left rule for $A_{\mathsf{L}} \multimap B_{\mathsf{L}}$ because it can appear in only one of the two judgments.

$$\frac{\Gamma\ ;\ \Delta_1 \vdash A_{\mathsf{L}} \quad \Gamma\ ;\ \Delta_2, B_{\mathsf{L}} \vdash C_{\mathsf{L}}}{\Gamma\ ;\ \Delta_1, \Delta_2, A_{\mathsf{L}} \multimap B_{\mathsf{L}} \vdash C_{\mathsf{L}}}\ \multimap L$$

On the other hand, structural implication can appear in both judgment forms, so there are two left rules for $A_{\mathsf{S}} \supset B_{\mathsf{S}}$.

$$\frac{\Gamma, A_{\mathsf{S}} \supset B_{\mathsf{S}} \vdash A_{\mathsf{S}} \quad \Gamma, A_{\mathsf{S}} \supset B_{\mathsf{S}}, B_{\mathsf{S}}\ ;\ \Delta \vdash C_{\mathsf{L}}}{\Gamma, A_{\mathsf{S}} \supset B_{\mathsf{S}}\ ;\ \Delta \vdash C_{\mathsf{L}}}\ \supset L_{\mathsf{SL}}$$

$$\frac{\Gamma, A_{\mathsf{S}} \supset B_{\mathsf{S}} \vdash A_{\mathsf{S}} \quad \Gamma, A_{\mathsf{S}} \supset B_{\mathsf{S}}, B_{\mathsf{S}} \vdash C_{\mathsf{S}}}{\Gamma, A_{\mathsf{S}} \supset B_{\mathsf{S}} \vdash C_{\mathsf{S}}}\ \supset L_{\mathsf{SS}}$$

Note that the first premises of the two rules are the same. That's because A_{S} is structural, and thereby determines the judgment form needed to prove it. It is also forced that we sort B_{S} into Γ, just from its mode.

4 Rules for Shifts

Besides identifying the right judgment forms, the rules for the shifts are a key to understanding the mixed linear/nonlinear system. As guidance in this process, let's recall the rules in the dyadic system from last lecture. We show the original form of $!R$ with an explicit, albeit forced, valid_R rule.

$$\frac{\Gamma\ ;\ \cdot \vdash A\ \mathit{true}}{\Gamma \vdash A\ \mathit{valid}}\ \mathsf{valid}_R \qquad \frac{\Gamma, A\ \mathit{valid}\ ;\ \Delta, A\ \mathit{true} \vdash C\ \mathit{true}}{\Gamma, A\ \mathit{valid}\ ;\ \Delta \vdash C\ \mathit{true}}\ \mathsf{valid}_L$$

$$\frac{\Gamma \vdash A\ \mathit{valid}}{\Gamma\ ;\ \cdot \vdash\ !A\ \mathit{true}}\ !R \qquad \frac{\Gamma, A\ \mathit{valid}\ ;\ \Delta \vdash C\ \mathit{true}}{\Gamma\ ;\ \Delta, !A \vdash C\ \mathit{true}}\ !L$$

Here is a thought experiment: what if the structural layer was only occupied by $\uparrow A_{\mathsf{L}}$? Then for any proposition $\downarrow A_{\mathsf{S}}$ the proposition A_{S} must have the form $\uparrow A_{\mathsf{L}}$. Then we can try to gain insight from the decomposition $!A_L \triangleq \downarrow\uparrow A_{\mathsf{L}}$. Here, both A_{L} and $\downarrow\uparrow A_{\mathsf{L}}$ are at mode L. Therefore, $A_{\mathsf{S}}\ \mathit{valid}$ corresponds to $\uparrow A_{\mathsf{L}}\ \mathit{true}$ because this is the only possibility for A_{S}. From this we get the following rules (showing the prior

rule on the left, the LNL rule on the right):

$$\frac{\Gamma\ ;\cdot \vdash A\ \mathit{true}}{\Gamma \vdash A\ \mathit{valid}}\ \mathsf{valid}_R \qquad \frac{\Gamma\ ;\cdot \vdash A_{\mathsf{L}}}{\Gamma \vdash \uparrow A_{\mathsf{L}}}\ \uparrow R$$

$$\frac{\Gamma, A\ \mathit{valid}\ ;\ \Delta, A\ \mathit{true} \vdash C\ \mathit{true}}{\Gamma, A\ \mathit{valid}\ ;\ \Delta \vdash C\ \mathit{true}}\ \mathsf{valid}_L \qquad \frac{\Gamma, \uparrow A_{\mathsf{L}}\ ;\ \Delta, A_{\mathsf{L}} \vdash C_{\mathsf{L}}}{\Gamma, \uparrow A_{\mathsf{L}}\ ;\ \Delta \vdash C_{\mathsf{L}}}\ \uparrow L$$

$$\frac{\Gamma \vdash A\ \mathit{valid}}{\Gamma\ ;\cdot \vdash\ !A\ \mathit{true}}\ !R \qquad \frac{\Gamma \vdash A_{\mathsf{S}}}{\Gamma\ ;\cdot \vdash \downarrow A_{\mathsf{S}}}\ \downarrow R$$

$$\frac{\Gamma, A\ \mathit{valid}\ ;\ \Delta \vdash C\ \mathit{true}}{\Gamma\ ;\ \Delta, !A \vdash C\ \mathit{true}}\ !L \qquad \frac{\Gamma, A_{\mathsf{S}}\ ;\ \Delta \vdash C_{\mathsf{L}}}{\Gamma\ ;\ \Delta, \downarrow A_{\mathsf{S}} \vdash C_{\mathsf{L}}}\ \downarrow L$$

Some of the anomalies have disappeared. For example, every rule now concerns a particular logical connective rather than a judgment.

Furthermore, we have an explanation why $!A$ was neither positive nor negative. From writing out the proofs of identities we can see that $\uparrow$ is negative (invertible on the right) and $\downarrow$ is positive (invertible on the left). So $!A_{\mathsf{L}} = \downarrow\uparrow A_{\mathsf{L}}$ is neither positive nor negative because there is a shift in polarity between the two shifts. It is sometimes said that ! is "*positive on the outside and negative on the inside*", which reflects this decomposition precisely.

5 Identity and Cut

In LNL we have two forms of identity, and several forms of cut. This is because we can apply rules directly to structural propositions.

$$\frac{}{\Gamma, A_{\mathsf{S}} \vdash A_{\mathsf{S}}}\ \mathsf{id}_{\mathsf{S}} \qquad \frac{}{\Gamma\ ;\ A_{\mathsf{L}} \vdash A_{\mathsf{L}}}\ \mathsf{id}_{\mathsf{L}}$$

For the same reason we have multiple left rules, we also get multiple versions of cut (three, to be precise).

$$\frac{\Gamma \vdash A_{\mathsf{S}} \quad \Gamma, A_{\mathsf{S}} \vdash C_{\mathsf{S}}}{\Gamma \vdash C_{\mathsf{S}}}\ \mathsf{cut}_{\mathsf{SS}} \qquad \frac{\Gamma \vdash A_{\mathsf{S}} \quad \Gamma, A_{\mathsf{S}}\ ;\ \Delta' \vdash C_{\mathsf{L}}}{\Gamma\ ;\ \Delta' \vdash C_{\mathsf{L}}}\ \mathsf{cut}_{\mathsf{SL}}$$

$$\frac{\Gamma\ ;\ \Delta \vdash A_{\mathsf{L}} \quad \Gamma\ ;\ \Delta', A_{\mathsf{L}} \vdash C_{\mathsf{L}}}{\Gamma\ ;\ \Delta, \Delta' \vdash C_{\mathsf{L}}}\ \mathsf{cut}_{\mathsf{LL}}$$

The admissibility of cut and identity is not essentially different from before, and perhaps proof are even a bit simpler since we do not need to order different forms of cut among each other. But we need to prove it simultaneously for the two forms of identity and three three forms of cut since shifts will move us between these judgments.

Theorem 1 (Admissibility of Identity) *The rules of identity* id_S *and* id_L *are admissible in the system where they are restricted to atomic propositions.*

Proof: By simultaneous structural induction on A_S and A_L. □

Theorem 2 (Admissibility of Cut) *The three rules of cut are admissible in the system without cut.*

Proof: By simultaneous nested induction on all three forms of cut, first on the cut proposition A_L and A_S, and second on the first and second given derivation.

To account for the requirement that the structural context Γ be the same among the premises of the cut we may have to apply the admissibility of weakening for structural antecedents, that is, we can add a new antecedent B_S to every sequent in a proof. □

Once we have cut elimination, the conservative extension properties are almost immediate.

Theorem 3 (Conservative Extension)

(i) If Γ *and* A_S *do not contain any upshift* $\uparrow$ *then* $\Gamma \vdash A_\mathsf{S}$ *if and only if* $\Gamma \vdash A_\mathsf{S}$ *in structural (intuitionistic) logic.*

(ii) If Δ *and* A_L *do not contain any downshift* $\downarrow$ *then* $\cdot \,;\, \Delta \vdash A_\mathsf{L}$ *if and only if* $\Delta \vdash A_\mathsf{L}$ *in purely linear logic.*

Proof: All the rules except cut have the subformula property, and the rules can be read as rules from the purely structural (part (i)) or purely linear (part (ii)) sequent calculus. Therefore conservative extension follows directly from cut elimination. □

6 Examples

We can now experiment with some properties. For example, we see that $\uparrow$ distributes over implication, we just have to be careful to pick the *correct* kind of implication. We elide some structural antecedents when they are no longer needed.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{- \,;\, A_\mathsf{L} \vdash A_\mathsf{L}}\,\mathsf{id} \quad \dfrac{}{- \,;\, B_\mathsf{L} \vdash B_\mathsf{L}}\,\mathsf{id}}{- \,;\, A_\mathsf{L} \multimap B_\mathsf{L}, A_\mathsf{L} \vdash B_\mathsf{L}}\,{\multimap}L}{\uparrow(A_\mathsf{L} \multimap B_\mathsf{L}), \uparrow A_\mathsf{L} \,;\, \cdot \vdash B_\mathsf{L}}\,\uparrow L \times 2}{\uparrow(A_\mathsf{L} \multimap B_\mathsf{L}), \uparrow A_\mathsf{L} \vdash \uparrow B_\mathsf{L}}\,\uparrow R}{\cdot \vdash \uparrow(A_\mathsf{L} \multimap B_\mathsf{L}) \supset (\uparrow A_\mathsf{L} \supset \uparrow B_\mathsf{L})}\,{\supset}R \times 2$$

The downshift also distributes over implication.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{-, A_\mathsf{s} \vdash A_\mathsf{s}}\,\mathsf{id}_\mathsf{s} \quad \dfrac{}{-, B_\mathsf{s} \vdash B_\mathsf{s}}\,\mathsf{id}_\mathsf{s}}{A_\mathsf{s} \supset B_\mathsf{s}, A_\mathsf{s} \vdash B_\mathsf{s}}\,{\supset}L}{A_\mathsf{s} \supset B_\mathsf{s}, A_\mathsf{s} \,;\, \cdot \vdash {\downarrow}B_\mathsf{s}}\,{\downarrow}R}{\cdot \,;\, {\downarrow}(A_\mathsf{s} \supset B_\mathsf{s}), {\downarrow}A_\mathsf{s} \vdash {\downarrow}B_\mathsf{s}}\,{\downarrow}L \times 2}{\cdot \,;\, \cdot \vdash {\downarrow}(A_\mathsf{s} \supset B_\mathsf{s}) \multimap ({\downarrow}A_\mathsf{s} \multimap {\downarrow}B_\mathsf{s})}\,{\supset}R \times 2$$

The properties we proved last time using $!A$ carry over under the definition as $\downarrow\uparrow A$. The proofs remain essentially the same, with just a couple of additional administrative steps and different naming of the rules.

7 A Programming Example

One advantage of LNL is more direct expression of mixed programs. As an example that also illustrates parallelism, we show *mapreduce* over trees that have data only at the leaves. *mapreduce* is a fold operation over such trees, so the type tree_A is replaced by a fresh type variable B.

$$\mathsf{tree}_A = \oplus\{\mathsf{leaf} : A, \mathsf{node} : \mathsf{tree}_A \otimes \mathsf{tree}_A\}$$
$$\mathit{mapreduce}_{A\,B}\,(r : B)\;(h : A \multimap B)\;(f : B \multimap (B \multimap B))\;(t : \mathsf{tree}_A) = \ldots \;?$$

In the type of *mapreduce*, r stands for the result, that is, the channel to deliver the result at. We have curried the type of the function f for ease of programming.

A problem is that we will not be able to write a recursive *mapreduce* with this type since h and f are channels of linear type. h will be used for every leaf, and f will be used for every node, so both will be used multiple times. We could try to rewrite *mapreduce* as an iterator—here we want to write it directly. This means that both h and f should be structural: we need upshifts.

$$\mathit{mapreduce}_{A\,B}\,(r : B)\;(h : {\uparrow}(A \multimap B))\;(f : {\uparrow}(B \multimap (B \multimap B)))\;(t : \mathsf{tree}_A) = \ldots$$

At this point it would be relatively straightforward to write the code, if we only had the *dynamics* of the shifts. Let's develop this and then come back to the example.

8 Dynamics of the Shifts

We start with $\uparrow A$. Since it is negative it will receive, but what? Let's look at the rule and annotate with channels.

$$\dfrac{\Gamma \,;\, \cdot \vdash A_\mathsf{L}}{\Gamma \vdash {\uparrow}A_\mathsf{L}}\,{\uparrow}R \qquad \dfrac{\Gamma \,;\, \cdot \vdash (y_\mathsf{L} : A_\mathsf{L})}{\Gamma \vdash (x_\mathsf{S} : {\uparrow}A_\mathsf{L})}\,{\uparrow}R$$

We need to transition from a structural channel x_S to a linear channel y_L, so that's what we need to receive! We write $\langle y \rangle$ for receiving a channel of a different mode (linear or structural). Then we have:

$$\frac{\Gamma \,;\, \cdot \vdash P(y_\mathsf{L}) :: (y_\mathsf{L} : A_\mathsf{L})}{\Gamma \vdash \mathbf{recv}\ x_\mathsf{S}\ (\langle y_\mathsf{L} \rangle \Rightarrow P(y_\mathsf{L})) :: (x_\mathsf{S} : \uparrow A_\mathsf{L})}\ \uparrow R$$

Next, the left rule $\uparrow L$. To match $\uparrow R$, we clearly need to send a channel but which one? We annotate the rule and then think of a process notation.

$$\frac{\Gamma, \uparrow A_\mathsf{L} \,;\, \Delta, A_\mathsf{L} \vdash C_\mathsf{L}}{\Gamma, \uparrow A_\mathsf{L} \,;\, \Delta \vdash C_\mathsf{L}}\ \uparrow L \qquad \frac{\Gamma, x_\mathsf{S} : \uparrow A_\mathsf{L} \,;\, \Delta, y_\mathsf{L} : A_\mathsf{L} \vdash (z : C_\mathsf{L})}{\Gamma, x_\mathsf{S} : \uparrow A_\mathsf{L} \,;\, \Delta \vdash (z : C_\mathsf{L})}\ \uparrow L$$

We see that what we have to send is a fresh linear channel y_L. This is a new phenomenon since there is no cut involved, and so far only cut would create a fresh channel. So we have to make up some new form of syntax.

$$\frac{\Gamma, x_\mathsf{S} : \uparrow A_\mathsf{L} \,;\, \Delta, y_\mathsf{L} : A_\mathsf{L} \vdash Q(y_\mathsf{L}) :: (z : C_\mathsf{L})}{\Gamma, x_\mathsf{S} : \uparrow A_\mathsf{L} \,;\, \Delta \vdash \mathbf{send}\ x_\mathsf{S}\ (\langle y_\mathsf{L} \rangle \Rightarrow Q(y_\mathsf{L})) :: (z : C_\mathsf{L})}\ \uparrow L$$

What happens operationally? The client sends a fresh channel and the provider continues with the fresh channel. So the first approximation would be

$$\begin{array}{lll} & \mathsf{proc}(\mathbf{recv}\ a_\mathsf{S}\ (\langle y_\mathsf{L} \rangle \Rightarrow P(y_\mathsf{L}))), & \mathsf{proc}(\mathbf{send}\ a_\mathsf{S}\ (\langle y_\mathsf{L} \rangle \Rightarrow Q(y_\mathsf{L}))) \\ \longrightarrow & \mathsf{proc}(P(b_\mathsf{L})), & \mathsf{proc}(Q(b_\mathsf{L})) \qquad b_\mathsf{L}\ \text{fresh} \end{array}$$

This does not quite work, however, since structural channels may have multiple clients. In the rule above the provider of a_S disappears after interacting with one client, so the other clients will be left dangling without a provider.

A similar practically occurring scenario is a web server. When a client sends an HTTP request, the server spawns a fresh one-to-one connection and meanwhile continues to serve other requests. What this means here is that the provider of a structural channel spawns a fresh linear process *but remains in the configuration* to receive further requests.

We model this with a new feature of multiset rewriting: we combine linear and structural inference. The state consists of a set (the persistent propositions) and a multiset (the ephemeral propositions). We just *underline* the persistent proposition, because the alternative notation $!A$ may be misleading. Then the rule reads:

$$\begin{array}{lll} & \underline{\mathsf{proc}}(\mathbf{recv}\ a_\mathsf{S}\ (\langle y_\mathsf{L} \rangle \Rightarrow P(y_\mathsf{L}))), & \mathsf{proc}(\mathbf{send}\ a_\mathsf{S}\ (\langle y_\mathsf{L} \rangle \Rightarrow Q(y_\mathsf{L}))) \\ \longrightarrow & \mathsf{proc}(P(b_\mathsf{L})), & \mathsf{proc}(Q(b_\mathsf{L})) \qquad b_\mathsf{L}\ \text{fresh} \end{array}$$

The persistent semantic objects originate in the $\mathsf{cut}_{\mathsf{SL}}$ rule (we omit the $\mathsf{cut}_{\mathsf{SS}}$ rule). With process terms:

$$\frac{\Gamma \vdash P(x_\mathsf{S}) :: (x_\mathsf{S} : A_\mathsf{S}) \quad \Gamma, x_\mathsf{S} : A_\mathsf{S} \,;\, \Delta' \vdash Q(x_\mathsf{S}) :: (z_\mathsf{L} : C_\mathsf{L})}{\Gamma \,;\, \Delta' \vdash x_\mathsf{S} \leftarrow P(x_\mathsf{S}) \,;\, Q(x_\mathsf{S}) :: (z_\mathsf{L} : C_\mathsf{L})}\ \mathsf{cut}_{\mathsf{SL}}$$

And the dynamics:

$$\mathsf{proc}(x_\mathsf{s} \leftarrow P(x_\mathsf{s}) \,;\, Q(x_\mathsf{s})) \quad \longrightarrow \quad \underline{\mathsf{proc}}(P(a_\mathsf{s})), \mathsf{proc}(Q_\mathsf{s}(a_\mathsf{s})) \quad (a_\mathsf{s} \text{ fresh})$$

Because it is not needed for the example, for the downshifts we jump directly to the annotated versions and dynamics. As with other symmetric pairs of connectives ($\multimap$ / $\otimes$ and $\&$ / $\oplus$) we'd like to reuse the syntax for a streamlined language, just swapping provider and client roles.

$$\frac{\Gamma \vdash P(y_\mathsf{s}) :: (y_\mathsf{s} : A_\mathsf{s})}{\Gamma \,;\, \cdot \vdash \mathbf{send}\ x_\mathsf{L}\ (\langle y_\mathsf{s}\rangle \Rightarrow P(y_\mathsf{s})) :: (x_\mathsf{L} : {\downarrow}A_\mathsf{s})}\ {\downarrow}R$$

$$\frac{\Gamma, y_\mathsf{s} : A_\mathsf{s} \,;\, \Delta \vdash Q(y_\mathsf{s}) :: (z_\mathsf{L} :: C_\mathsf{L})}{\Gamma \,;\, \Delta, x_\mathsf{L} : {\downarrow}A_\mathsf{s} \vdash \mathbf{recv}\ x_\mathsf{L}\ (\langle y_\mathsf{s}\rangle \Rightarrow P(y_\mathsf{s})) :: (z_\mathsf{L} : C_\mathsf{L})}\ {\downarrow}L$$

The dynamics is similar to the one for ${\uparrow}A_\mathsf{L}$, except that different processes are persistent and channels go from linear to structural instead of vice versa.

$$\begin{array}{lll} & \mathsf{proc}(\mathbf{send}\ a_\mathsf{L}\ (\langle y_\mathsf{s}\rangle \Rightarrow P(y_\mathsf{s}))), & \mathsf{proc}(\mathbf{recv}\ a_\mathsf{L}\ (\langle y_\mathsf{s}\rangle \Rightarrow Q(y_\mathsf{s}))) \\ \longrightarrow & \underline{\mathsf{proc}}(P(b_\mathsf{s})), & \mathsf{proc}(Q(b_\mathsf{s})) \qquad\qquad b_\mathsf{s} \text{ fresh} \end{array}$$

9 Example Continued

Returning to the example, we can now write the process for *mapreduce*. Both h and f are structural channels; we omit the subscript on the linear channels.

$$\begin{array}{l} \mathsf{tree}_A = \oplus\{\mathsf{leaf} : A, \mathsf{node} : \mathsf{tree}_A \otimes \mathsf{tree}_A\} \\ \mathit{mapreduce}_{AB}\ (r : B)\ (h_\mathsf{s} : {\uparrow}(A \multimap B))\ (f_\mathsf{s} : {\uparrow}(B \multimap (B \multimap B)))\ (t : \mathsf{tree}_A) = \\ \quad \mathbf{recv}\ t\ (\ \mathsf{leaf} \Rightarrow \mathbf{send}\ h_\mathsf{s}\ (\langle h'\rangle \Rightarrow \mathbf{send}\ h'\ t\ ;\ \mathbf{fwd}\ r\ h') \\ \qquad\quad \mid \mathsf{node} \Rightarrow \mathbf{recv}\ t\ (s \Rightarrow \\ \qquad\qquad\qquad x \leftarrow \mathbf{call}\ \mathit{mapreduce}_{AB}\ x\ h_\mathsf{s}\ f_\mathsf{s}\ s\ ; \\ \qquad\qquad\qquad y \leftarrow \mathbf{call}\ \mathit{mapreduce}_{AB}\ y\ h_\mathsf{s}\ f_\mathsf{s}\ t\ ; \\ \qquad\qquad\qquad \mathbf{send}\ f_\mathsf{s}\ (\langle f'\rangle \Rightarrow \\ \qquad\qquad\qquad \mathbf{send}\ f'\ x\ ; \\ \qquad\qquad\qquad \mathbf{send}\ f'\ y\ ; \\ \qquad\qquad\qquad \mathbf{fwd}\ r\ f')\,)\,) \end{array}$$

The structural nature of h_s and f_s is crucial here because they are indeed used multiple or zero times in the two branches.

This may also be a good time to think about parallelism. We see that the two recursive calls to *mapreduce* proceed entirely independently. But the parallelism even goes further: the computation of f' on x and y can proceed while *mapreduce* is still computing. So x and y are shared between the providers (the recursive calls

to $mapreduce$) and the client (f', a linear instance of f_s), synchronization between these two processes only takes place during input or output on those channels. This phenomenon of *pipelining* can improve the asymptotic complexity of some parallel algorithms, as observed by Blelloch and Reid-Miller [1999].

Since it came up during lecture: if we change the type of f_s to take a pair of channels, we have to create this pair in the code which is minimally more complicated. The result is below.

$$
\begin{array}{l}
\mathsf{tree}_A = \oplus\{\mathsf{leaf} : A, \mathsf{node} : \mathsf{tree}_A \otimes \mathsf{tree}_A\} \\
mapreduce_{AB}\,(r : B)\;(h_\mathsf{s} : \uparrow(A \multimap B))\;(f_\mathsf{s} : \uparrow((B \otimes B) \multimap B))\;(t : \mathsf{tree}_A) = \\
\quad \mathbf{recv}\ t\ (\,\mathsf{leaf} \Rightarrow \mathbf{send}\ h_\mathsf{s}\ (\langle h' \rangle \Rightarrow \mathbf{send}\ h'\ t\ ;\ \mathbf{fwd}\ r\ h') \\
\qquad\quad \mid \mathsf{node} \Rightarrow \mathbf{recv}\ t\ (s \Rightarrow \\
\qquad\qquad\qquad x \leftarrow \mathbf{call}\ mapreduce_{AB}\ x\ h_\mathsf{s}\ f_\mathsf{s}\ s\ ; \\
\qquad\qquad\qquad y \leftarrow \mathbf{call}\ mapreduce_{AB}\ y\ h_\mathsf{s}\ f_\mathsf{s}\ t\ ; \\
\qquad\qquad\qquad \mathbf{send}\ f_\mathsf{s}\ (\langle f' \rangle \Rightarrow \\
\qquad\qquad\qquad p_{B\otimes B} \leftarrow (\mathbf{send}\ p\ x\ ;\ \mathbf{fwd}\ p\ y)\ ; \\
\qquad\qquad\qquad \mathbf{send}\ f'\ p\ ; \\
\qquad\qquad\qquad \mathbf{fwd}\ r\ f')\,)\,)
\end{array}
$$

10 Summary

We have presented LNL [Benton, 1994], a mixed linear/nonlinear logic that arises from the logic of validity from the last lecture by decomposing $!A$ into two shifts: one ($\uparrow$) from linear to structural and one ($\downarrow$) from structural to linear. Benton makes the point that there is an adjunction between the two shifts, which results in their composition $\downarrow\uparrow$ being a comonad and the opposite composition $\uparrow\downarrow$ being a monad.

This approach solves some small issues with the dyadic formulation from last lecture. For example, $\uparrow$ is negative and $\downarrow$ is positive, which means their composition $!$ is neither.

Since we view LNL mainly as a stepping stone to full adjoint logic[1] we do not summarize the rules here.

References

Nick Benton. A mixed linear and non-linear logic: Proofs, terms and models. In Leszek Pacholski and Jerzy Tiuryn, editors, *Selected Papers from the 8th International Workshop on Computer Science Logic (CSL'94)*, pages 121–135, Kazimierz, Poland, September 1994. Springer LNCS 933. An extended version appears as Technical Report UCAM-CL-TR-352, University of Cambridge.

[1] briefly previewed in this lecture, but covered in the notes for the next lecture

G. E. Blelloch and M. Reid-Miller. Pipeling with futures. *Theory of Computing Systems*, 32:213–239, 1999.

Klaas Pruiksma, William Chargin, Frank Pfenning, and Jason Reed. Adjoint logic. Unpublished manuscript, April 2018. URL http://www.cs.cmu.edu/~fp/papers/adjoint18b.pdf.

Jason Reed. A judgmental deconstruction of modal logic. Unpublished manuscript, May 2009. URL http://www.cs.cmu.edu/~jcreed/papers/jdml2.pdf.

Lecture Notes on Adjoint Logic

15-836: Substructural Logics
Frank Pfenning

Lecture 11
October 3, 2023

1 Introduction

In the last lecture we introduced LNL, a mixed linear/nonlinear logic that directly contains linear and nonlinear propositions instead of the exponential $!A$ (which could be defined as $\downarrow\uparrow A$). This eliminates some drawbacks of coding all nonlinear propositions via the exponential, but it has some of its own issues. For example, we have seen that there are two right rules for implication (one for $A \multimap B$ and one for $A \supset B$), and three left rules for implication (one for $A \multimap B$ and two for $A \supset B$).

One question is if we can streamline this so we would only have two rules implication (one right and one left rule) rather than five as in LNL. Another is if we can generalize the LNL approach to combine different logics, rather than just intuitionistic structural and linear logics. One answer to both questions is provided by *adjoint logic*, a general schema for combining certain classes of logics based on simple principles. The idea was first sketched by Reed [2009] and further developed by Pruiksma et al. [2018] and others (e.g., [Chargin, 2017, Licata and Shulman, 2016, Licata et al., 2017, Pruiksma and Pfenning, 2021]).

The generality of adjoint logic then yields a number of familiar logics such as *lax logic* Fairtlough and Mendler [1997] which is related to *computational monads* Moggi [1991], or the intuitionistic modal logic S4 [Pfenning and Davies, 2001] which is related to staged computation and metaprogramming [Davies and Pfenning, 2001].

In this and the following lecture we assume that exchange is always present as a structural property, although this is not necessary. We leave further discussion of ordered logic in the adjoint context either to a future lecture or a miniproject.

Mostly, it seems, we like to use weakening and contraction together. Occasionally, it is suitable to postulate just weakening or contraction in isolation. For example, if we want to allow failure and process cancelation, then weakening may be appropriate (not every process providing a channel may actually be used). The type system of Rust is also affine, that is, permits weakening but not contraction in

certain ways that are not entirely captured here, but analogous. Another example is expressing *strictness analysis* in a language such as Haskell as a type system. When a function definitely uses its argument then it is strict, that is, it can (explicitly or implicitly) employ contraction but not weakening.

2 Adjoint Logic: The Basics

In adjoint logic every proposition has an intrinsic *mode of truth*, where each mode may or may not admit weakening and contraction. We define the meaning of the *connectives* uniformly at all modes by their right and left rules. So ultimately they are distinguished only by the structural properties their mode satisfies.

As an example, LNL as an instance of the adjoint logic framework would have two modes: S for structural propositions and L for linear propositions.

We also have a preorder $m \geq k$ on modes which expresses that the proof of a proposition A_k may depend on an antecedent B_m. Conversely, when $m \not\geq k$ then B_m may *not* be among the antecedents of a proof of m. We call this the *principle of independence*. A necessary use of this is in LNL, where a proof of a structural proposition A_{S} may not depend on a linear proposition A_{L}. So we have $\mathsf{S} > \mathsf{L}$ as our preorder (and, in particular $\mathsf{L} \not\geq \mathsf{S}$).

In addition to the usual connectives, adjoint logic also generalizes the shifts from LNL to go between two arbitrary modes. For $\uparrow_k^m A_k$ we require $m \geq k$ and for $\downarrow_m^\ell A_\ell$ we require $\ell \geq m$.

$$A_m ::= P_m \mid A_m \to B_m \mid A_m \times B_m \mid \mathbf{1} \mid A_m \,\&\, B_m \mid \top \mid A_m + B_m \mid \mathbf{0} \mid \uparrow_k^m A_k \mid \downarrow_m^\ell A_\ell$$

We chose a new syntax, partially based on the reading of propositions as types. So $A \to B$ unifies $A \supset B$ and $A \multimap B$, $A \times B$ stands for $A \wedge B$ and $A \otimes B$, and $A + B$ stands for $A \vee B$ and $A \oplus B$. Even the logical constants $\mathbf{1}$, $\top$ and $\mathbf{0}$ should be thought of as having an intrinsic mode, even if we don't write them this way in the grammar.

We write $\sigma(m)$ for the set of structural properties satisfied by mode m, where $\sigma(m) \subseteq \{\mathsf{W}, \mathsf{C}\}$. As mentioned in the introduction, we always assume exchange. Based on the experience with cut elimination and validity (or the exponential), we require:

If $m \geq k$ then $\sigma(m) \supseteq \sigma(k)$.

And, indeed cut elimination fails if we omit this requirement. Furthermore, any dependence in a sequent must be allowed by the preorder among modes.

Whenever we write $\Delta \vdash A_m$ we require $\Delta \geq m$.

This *presupposition* means we can never ask a question $\Delta \vdash A_m$ unless for all $B_\ell \in \Delta$ we have $\ell \geq m$. We write Δ here for the antecedents because we treat the antecedents as a multiset. This means, weakening and contraction must be explicit rules for those modes that permit it.

Like Gentzen [1935], we read the inference rules of the sequent calculus bottom-up, as a means to construct a proof. When viewed in this direction, we need to ensure there are sufficient preconditions in the rule to ensure the premises satisfy our presupposition when the conclusion does.

As a start, write out the structural rules. Weakening applies to antecedents that permit it explicitly, and similarly for contraction.

$$\frac{\mathsf{W} \in \sigma(m) \quad \Delta \vdash C_r}{\Delta, A_m \vdash C_r}\ \mathsf{weaken} \qquad \frac{\mathsf{C} \in \sigma(m) \quad \Delta, A_m, A_m \vdash C_r}{\Delta, A_m \vdash C_r}\ \mathsf{contract}$$

Also generic are the rules of cut and identity. First, identity.

$$\frac{}{A_m \vdash A_m}\ \mathsf{id}$$

Cut requires a bit of thought. As usual, we start by writing down what we know directly, and then think about what else may be needed.

$$\frac{\Delta \vdash A_m \quad \Delta', A_m \vdash C_r}{\Delta, \Delta' \vdash C_r}\ \mathsf{cut?}$$

At first glance it might seem this should be it, but we remember that in LNL we needed three rules. For example, if $m = \mathsf{S}$ (that is, m is structural) then Δ may not contain any linear antecedents (that is, B_L). This issue surfaces here when we reason about our presupposition. Let's write in blue what we know and in red what we need to know for the premises.

$$\frac{\begin{array}{c}\color{red}{\Delta \geq m?} \quad \color{red}{\Delta' \geq r, m \geq r?} \\ \Delta \vdash A_m \quad \Delta', A_m \vdash C_r\end{array}}{\begin{array}{c}\Delta, \Delta' \vdash C_r \\ \color{blue}{\Delta, \Delta' \geq r}\end{array}}\ \mathsf{cut?}$$

We see that we already know $\Delta' \geq r$ from the presupposition for the conclusion, but we know neither $\Delta \geq m$ nor $m \geq r$. These two conditions thus need to be explicitly enforced and we obtain:

$$\frac{\Delta \geq m \geq r \quad \Delta \vdash A_m \quad \Delta', A_m \vdash C_r}{\Delta, \Delta' \vdash C_r}\ \mathsf{cut}$$

You should convince yourself that in the case where we have just two modes, L and S, this gives rise exactly to the three forms of cut in LNL.

3 Logical Rules

Next we can define the logical rules, uniformly across the modes. We start with implication, which is almost a worst case for its complexity.

$$\frac{\Delta, A_m \vdash B_m}{\Delta \vdash A_m \to B_m} \to R$$

The presupposition tells us that $\Delta \geq m$, which is sufficient to ensure that $(\Delta, A_m) \geq m$. It is good there is no condition: since implication is negative, we would expect it to be right invertible and therefore not be subject to any conditions. For the left rule, matters are not quite as simple.

$$\frac{\begin{array}{c}\Delta \geq m? \quad \Delta' \geq r, m \geq r? \\ \Delta \vdash A_m \qquad \Delta', B_m \vdash C_r\end{array}}{\begin{array}{c}\Delta, \Delta', A_m \to B_m \vdash C_r \\ (\Delta, \Delta') \geq r, m \geq r\end{array}} \to L?$$

We see that $\Delta' \geq r$ and $m \geq r$ is already known, but $\Delta \geq m$ is not and must be added as a condition.

$$\frac{\Delta \geq m \quad \Delta \vdash A_m \quad \Delta', B_m \vdash C_r}{\Delta, \Delta', A_m \to B_m \vdash C_r} \to L$$

It turns out there isn't much of interest in the other rules. We only show the ones for tensor and unit.

$$\frac{\Delta \vdash A_m \quad \Delta' \vdash B_m}{\Delta, \Delta' \vdash A_m \times B_m} \times R \qquad \frac{\Delta, A_m, B_m \vdash C_r}{\Delta, A_m \times B_m \vdash C_r} \times L$$

$$\frac{}{\cdot \vdash \mathbf{1}} \mathbf{1}R \qquad \frac{\Delta \vdash C_r}{\Delta, 1 \vdash C_r} \mathbf{1}L$$

It is easy to see that for these (and the remaining rules except shifts) the presupposition for the conclusion immediately entails the presupposition for the premises and no additional conditions are needed. The rules are summarized in Figure 1.

4 The Shifts

The shifts generalize those from LNL, where $\uparrow$ would be replaced by $\uparrow^{\mathsf{S}}_{\mathsf{L}}$ and $\downarrow$ by $\downarrow^{\mathsf{S}}_{\mathsf{L}}$. Based on the polarity of the shifts in LNL we would expect the annotated shifts of adjoint logic to have the same polarities: $\uparrow$ should be negative and $\downarrow$ should be positive.

We can confirm this two ways: analyze the mode constraints in detail, and also derive the admissibility of the identity. Neither of these is proof, of course, which will be come back to in the next lecture.

We show first the rule with the known (blue) constraints and then the necessary constraints in red. The known constraint $m \geq k$ comes from the nature of the shift, since the upper mode must always be greater or equal to the lower mode.

$$\begin{array}{c} \Delta \geq k? \\ \dfrac{\Delta \vdash A_k}{\Delta \vdash \uparrow_k^m A_k}\uparrow R \\ \Delta \geq m, m \geq k \end{array} \qquad \dfrac{\Delta \vdash A_k}{\Delta \vdash \uparrow_k^m A_k}\uparrow R$$

As we might have predicted from the negative nature of the upshift, the presupposition of the premise follows from the presupposition of the conclusion.

In contrast, we expect some mode condition on the left rule for the upshift.

$$\begin{array}{c} \Delta \geq r, k \geq r? \\ \dfrac{\Delta, A_k \vdash C_r}{\Delta, \uparrow_k^m A_k \vdash C_r}\uparrow L \\ \Delta \geq r, m \geq k, m \geq r \end{array} \qquad \dfrac{k \geq r \quad \Delta, A_k \vdash C_r}{\Delta, \uparrow_k^m A_k \vdash C_r}\uparrow L$$

We show the rules for downshift in a similar form: collecting and checking dependence constraints and synthesizing the rule from them. If you try this yourself first, you will see that there is no leeway: the rules are uniquely determined.

$$\begin{array}{c} \Delta \geq \ell? \\ \dfrac{\Delta \vdash A_\ell}{\Delta \vdash \downarrow_m^\ell A_\ell}\downarrow R \\ \Delta \geq m, \ell \geq m \end{array} \qquad \dfrac{\Delta \geq \ell \quad \Delta \vdash A_\ell}{\Delta \vdash \downarrow_m^\ell A_\ell}\downarrow R$$

And finally the $\downarrow L$ rule, which requires no conditions.

$$\begin{array}{c} \Delta \geq r, \ell \geq r? \\ \dfrac{\Delta, A_\ell \vdash C_r}{\Delta, \downarrow_m^\ell A_\ell \vdash C_r}\downarrow L \\ \Delta \geq r, \ell \geq m, m \geq r \end{array} \qquad \dfrac{\Delta, A_\ell \vdash C_r}{\Delta, \downarrow_m^\ell A_\ell \vdash C_r}\downarrow L$$

5 Specific Logics as Instances of the Adjoint Schema

We obtain specific logics in the literature by specifying the modes, their dependence relation, and their structural properties.

Linear Logic. We obtain intuitionistic linear logic [Girard, 1987, Chang et al., 2003] with two modes, S and L where $\mathsf{S} > \mathsf{L}$ where $\sigma(\mathsf{S}) = \{\mathsf{W}, \mathsf{C}\}$ and $\sigma(\mathsf{L}) = \{\,\}$. Furthermore, we restrict the propositions at mode S as

$$A_{\mathsf{S}} ::= \uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}}$$

Then $!A_{\mathsf{L}} \triangleq \downarrow^{\mathsf{S}}_{\mathsf{L}} \uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}}$. We also rule out the shifts $\uparrow^m_m$ and $\downarrow^m_m$.

LNL. We obtain LNL [Benton, 1994] with the same modes, but the structural layer has a full set of connectives. We only rule out $\uparrow^m_m$ and $\downarrow^m_m$.

Intuitionistic S4. We obtain intuitionistic S4 [Pfenning and Davies, 2001] with two modes, V (validity) and T (truth), with $\sigma(\mathsf{V}) = \sigma(\mathsf{T}) = \{\mathsf{W}, \mathsf{C}\}$. The mode V is restricted analogously to S in linear logic:

$$A_{\mathsf{V}} ::= \uparrow^{\mathsf{V}}_{\mathsf{T}} A_{\mathsf{T}}$$

We also rule out $\uparrow^m_m$ and $\downarrow^m_m$. We define $\Box A_{\mathsf{T}} \triangleq \downarrow^{\mathsf{V}}_{\mathsf{T}} \uparrow^{\mathsf{V}}_{\mathsf{T}} A_{\mathsf{T}}$. As a composition of the two adjoint shift operators, $\Box$ is a *comonad*.

Perhaps somewhat surprisingly, we do not obtain the monad $\Diamond A_{\mathsf{T}}$ from a composition of shifts, although we can obtain $\bigcirc A_{\mathsf{T}}$, a strong monad and the basis for lax logic.

Lax Logic. We obtain *lax logic* [Fairtlough and Mendler, 1997] with two modes, T (truth) and X (lax truth), with $\mathsf{T} > \mathsf{X}$ and $\sigma(\mathsf{T}) = \sigma(\mathsf{X}) = \{\mathsf{W}, \mathsf{C}\}$. The mode X is restricted to

$$A_{\mathsf{X}} ::= \downarrow^{\mathsf{T}}_{\mathsf{X}} A_{\mathsf{T}}$$

and then $\bigcirc A_{\mathsf{T}} \triangleq \uparrow^{\mathsf{T}}_{\mathsf{X}} \downarrow^{\mathsf{T}}_{\mathsf{X}} A_{\mathsf{T}}$. The $\bigcirc$ modality is a (strong) monad.

It is now easy to extend and combine these. For examples, we can have a language for staged computation [Davies and Pfenning, 2001] where quoted expressions are drawn directly from the layer for validity. Or we can have a language that has both a monad and a comonad, with different structural properties.

6 Summary

The rules for adjoint logic are summarized in Figure 1.

Syntax with $\ell \geq m \geq k$ and $\sigma(m) \subseteq \{\mathsf{W}, \mathsf{C}\}$.

$$A_m ::= P_m \mid A_m \to B_m \mid A_m \times B_m \mid \mathbf{1} \mid A_m \,\&\, B_m \mid \top \mid A_m + B_m \mid \mathbf{0} \mid \uparrow_k^m A_k \mid \downarrow_m^\ell A_\ell$$

Rules.

$$\frac{\mathsf{W} \in \sigma(m) \quad \Delta \vdash C_r}{\Delta, A_m \vdash C_r}\ \mathsf{weaken} \qquad \frac{\mathsf{C} \in \sigma(m) \quad \Delta, A_m, A_m \vdash C_r}{\Delta, A_m \vdash C_r}\ \mathsf{contract}$$

$$\frac{}{A_m \vdash A_m}\ \mathsf{id} \qquad \frac{\Delta \geq m \geq r \quad \Delta \vdash A_m \quad \Delta', A_m \vdash C_r}{\Delta, \Delta' \vdash C_r}\ \mathsf{cut}$$

$$\frac{\Delta \vdash A_k}{\Delta \vdash \uparrow_k^m A_k}\ \uparrow R \qquad \frac{k \geq r \quad \Delta, A_k \vdash C_r}{\Delta, \uparrow_k^m A_k \vdash C_r}\ \uparrow L$$

$$\frac{\Delta \geq \ell \quad \Delta \vdash A_\ell}{\Delta \vdash \downarrow_m^\ell A_\ell}\ \downarrow R \qquad \frac{\Delta, A_\ell \vdash C_r}{\Delta, \downarrow_m^\ell A_\ell \vdash C_r}\ \downarrow L$$

$$\frac{\Delta, A_m \vdash B_m}{\Delta \vdash A_m \to B_m}\ \to R \qquad \frac{\Delta \geq m \quad \Delta \vdash A_m \quad \Delta', B_m \vdash C_r}{\Delta, \Delta', A_m \to B_m \vdash C_r}\ \to L$$

$$\frac{\Delta \vdash A_m \quad \Delta' \vdash B_m}{\Delta, \Delta' \vdash A_m \times B_m}\ \times R \qquad \frac{\Delta, A_m, B_m \vdash C_r}{\Delta, A_m \times B_m \vdash C_r}\ \times L$$

$$\frac{}{\cdot \vdash \mathbf{1}}\ \mathbf{1}R \qquad \frac{\Delta \vdash C_r}{\Delta, 1 \vdash C_r}\ \mathbf{1}L$$

$$\frac{\Delta \vdash A_m \quad \Delta \vdash B_m}{\Delta \vdash A_m \,\&\, B_m}\ \&R \qquad \frac{\Delta, A_m \vdash C_r}{\Delta, A_m \,\&\, B_m \vdash C_r}\ \&L_1 \qquad \frac{\Delta, B_m \vdash C_r}{\Delta, A_m \,\&\, B_m \vdash C_r}\ \&L_2$$

$$\frac{}{\Delta \vdash \top}\ \top R \qquad \text{no } \top L \text{ rule}$$

$$\frac{\Delta \vdash A_m}{\Delta \vdash A_m + B_m}\ +R_1 \qquad \frac{\Delta \vdash B_m}{\Delta \vdash A_m + B_m}\ +R_2 \qquad \frac{\Delta, A_m \vdash C_r \quad \Delta, B_m \vdash C_r}{\Delta, A_m + B_m \vdash C_r}\ +L$$

$$\text{no } \mathbf{0}R \text{ rule} \qquad \frac{}{\Delta, \mathbf{0} \vdash C_r}\ \mathbf{0}L$$

Figure 1: Adjoint Logic

References

Nick Benton. A mixed linear and non-linear logic: Proofs, terms and models. In Leszek Pacholski and Jerzy Tiuryn, editors, *Selected Papers from the 8th International Workshop on Computer Science Logic (CSL'94)*, pages 121–135, Kazimierz, Poland, September 1994. Springer LNCS 933. An extended version appears as Technical Report UCAM-CL-TR-352, University of Cambridge.

Bor-Yuh Evan Chang, Kaustuv Chaudhuri, and Frank Pfenning. A judgmental analysis of linear logic. Technical Report CMU-CS-03-131R, Carnegie Mellon University, Department of Computer Science, December 2003.

William Chargin. A general system of adjoint logic. Honors thesis, Carnegie Mellon University, December 2017.

Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, May 2001.

M. Fairtlough and M.V. Mendler. Propositional lax logic. *Information and Computation*, 137(1):1–33, August 1997.

Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

Daniel R. Licata and Michael Shulman. Adjoint logic with a 2-category of modes. In *International Symposium on Logical Foundations of Computer Science (LFCS)*, pages 219–235. Springer LNCS 9537, January 2016.

Daniel R. Licata, Michael Shulman, and Mitchell Riley. A fibrational framework for substructural and modal logics. In Dale Miller, editor, *Proceedings of the 2nd International Conference on Formal Structures for Computation and Deduction (FSCD'17)*, pages 25:1–25:22, Oxford, UK, September 2017. LIPIcs.

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999.

Klaas Pruiksma and Frank Pfenning. A message-passing interpretation of adjoint logic. *Journal of Logical and Algebraic Methods in Programming*, 120(100637), 2021.

Klaas Pruiksma, William Chargin, Frank Pfenning, and Jason Reed. Adjoint logic. Unpublished manuscript, April 2018. URL http://www.cs.cmu.edu/~fp/papers/adjoint18b.pdf.

Jason Reed. A judgmental deconstruction of modal logic. Unpublished manuscript, May 2009. URL http://www.cs.cmu.edu/~jcreed/papers/jdml2.pdf.

Lecture Notes on Focusing

15-836: Substructural Logics
Frank Pfenning

Lecture 12
October 5, 2023

1 Introduction

In this lecture we first revisit cut elimination by adapting it for adjoint logic. Then we introduce Andreoli's focusing [1992], a calculus for proof construction that exploits the negative and positive nature of the connectives to an extreme. Andreoli calls them asynchronous and synchronous connectives, and investigates *classical linear logic*, but his work seems nevertheless the beginning of the study of polarity. This was later adapted to various other logics [Liang and Miller, 2009], including adjoint logic [Pruiksma et al., 2018].

2 Cut Elimination Revisited

So far, we have studied cut elimination just in the ordered case, in some ways the most particular form. The argument for the linear connectives does not differ much, but the structural cases introduce some new considerations. Since we have introduced adjoint logic with explicit rules for weakening and contraction (mode permitting), we examine it in this case.

Recall the rule of cut, in this case formulated as a (hopefully) admissible property.

$$\frac{\Delta \geq m \geq r \quad \overset{\mathcal{D}}{\Delta \vdash A_m} \quad \overset{\mathcal{E}}{\Delta', A_m \vdash C_r}}{\Delta, \Delta' \vdash C_r}\ \mathsf{cut}_{A_m}$$

The case we are interested in here is where A_m is the result of contraction.

$$\dfrac{\Delta \geq m \geq r \quad \Delta \vdash A_m \quad \dfrac{\mathsf{C} \in \sigma(m) \quad \Delta', A_m, A_m \vdash C_r}{\Delta', A_m \vdash C_r}\,\mathsf{contract}}{\Delta, \Delta' \vdash C_r}\,\mathsf{cut}_{A_m}$$

A natural reduction is the cut out the two copies of A_m in sequence.

$$\longrightarrow \quad \dfrac{\Delta \geq m \geq r \quad \Delta \vdash A_m \quad \dfrac{\Delta \geq m \geq r \quad \Delta \vdash A_m \quad \Delta', A_m, A_m \vdash C_r}{\Delta, \Delta', A_m \vdash C_r}\,\mathsf{cut}_{A_m}}{\Delta, \Delta, \Delta' \vdash C_r}\,\mathsf{cut}_{A_m}$$

First we see that the conditions that $\Delta \geq m \geq r$ are known from the derivation before the reduction, so they are satisfied. Second, we observe that this reduction *does not preserve the conclusion* because there are now two copies of Δ!

Fortunately, the monotonicity condition on the structural properties comes to the rescue. We know $\mathsf{C} \in \sigma(m)$ from the given derivation, and we also know $\Delta \geq m$. This means for every $B_\ell \in \Delta$ we have $\ell \geq m$ and therefore $\mathsf{C} \in \sigma(\ell)$. This is sufficient to apply contraction to all antecedents in Δ.

$$\longrightarrow \quad \dfrac{\dfrac{\Delta \geq m \geq r \quad \Delta \vdash A_m \quad \dfrac{\Delta \geq m \geq r \quad \Delta \vdash A_m \quad \Delta', A_m, A_m \vdash C_r}{\Delta, \Delta', A_m \vdash C_r}\,\mathsf{cut}_{A_m}}{\Delta, \Delta, \Delta' \vdash C_r}\,\mathsf{cut}_{A_m}}{\Delta, \Delta' \vdash C_r}\,\mathsf{contract}^*$$

One bullet dodged!

Now the second bullet. The upper of the two cuts is a valid appeal to the induction hypothesis since the cut formula A_m remains the same, $\mathcal{D}$ remains the same, and $\mathcal{E}' < \mathcal{E}$. So the upper cut yields a derivation $\mathcal{F}$ of $\Delta, \Delta', A_m \vdash C_r$. Unfortunately, $\mathcal{F}$ may be much larger that $\mathcal{E}$, and the cut formula A_m is still the same, so we cannot appeal to our induction hypothesis!

While this particular reduction may have some merit if used in a process dynamics, it actually does not lead to a correct proof of cut admissibility.

At this point there are two main options. One is to rewrite the sequent calculus for adjoint logic so that contraction is implicit, as in our early formulas of structural logic. That is, antecedents subject to contraction are treated as a set. A second option is to generalize the rule of cut to eliminate multiple antecedents at once. We call this *multicut* even if this name is not universally agreed upon. We follow

this latter path because it illustrates a general technique when an inference system becomes more complex: we "absorb" additional rules into the cut by making it more general. Of course, this means that in some way we have to start from scratch and restart our overall proof, but the hope is we can follow the general inductive structure from before.

Our new, more general rule uses the notation $(A)^n$ to mean n copies of A.

$$\frac{\Delta \geq m \geq r \quad n \in \mu(m) \quad \Delta \vdash A_m \quad \Delta', (A_m)^n \vdash C_r}{\Delta, \Delta' \vdash C_r}\ \mathsf{multicut}_{A_m}$$

The new condition $n \in \mu(m)$ ensures that neither more nor fewer copies of A_m are cut out than allowed by the mode m. It is defined by

$$\begin{array}{lcll} \mu(m) & = & \{1\} & \text{if } \sigma(m) = \{\,\} \\ \mu(m) & = & \{0,1\} & \text{if } \sigma(m) = \{\mathsf{W}\} \\ \mu(m) & = & \{1,2,\ldots\} & \text{if } \sigma(m) = \{\mathsf{C}\} \\ \mu(m) & = & \{0,1,2,\ldots\} & \text{if } \sigma(m) = \{\mathsf{W},\mathsf{C}\} \end{array}$$

An interesting observation is that weakening and contraction are special cases of multicut. For contraction, the condition that $2 \in \mu(m)$ implies that $\mathsf{C} \in \sigma(m)$.

$$\frac{2 \in \mu(m) \quad \dfrac{}{A_m \vdash A_m}\ \mathsf{id} \quad \Delta', (A_m)^2 \vdash C_r}{\Delta', A_m \vdash C_r}\ \mathsf{multicut}$$

Weakening uses the odd special case where $n = 0$. The condition $0 \in \mu(m)$ implies that $\mathsf{W} \in \sigma(m)$.

$$\frac{m \geq r \quad 0 \in \mu(m) \quad \dfrac{}{A_m \vdash A_m}\ \mathsf{id} \quad \Delta', (A_m)^0 \vdash C_r}{\Delta', A_m \vdash C_r}\ \mathsf{multicut}$$

The rule of contraction is now a trivial case in the proof of admissibility of multicut. In fact, due to the admissibility of contraction it wouldn't even be necessary any more. But if we allow it, there must be at least one copy of A_m in the conclusion, which we capture by writing $(A)^{n+1}$.

$$\frac{\Delta \geq m \geq r \quad n+1 \in \mu(m) \quad \overset{\mathcal{D}}{\Delta \vdash A_m} \quad \dfrac{\mathsf{C} \in \sigma(m) \quad \overset{\mathcal{E}'}{\Delta', (A_m)^{n+2} \vdash C_r}}{\Delta', (A_m)^{n+1} \vdash C_r}\ \mathsf{contract}}{\Delta, \Delta' \vdash C_r}\ \mathsf{multicut}$$

$$\longrightarrow \quad \frac{\Delta \geq m \geq r \quad n+2 \in \mu(m) \quad \overset{\mathcal{D}}{\Delta \vdash A_m} \quad \overset{\mathcal{E}'}{\Delta, (A)^{n+2} \vdash C_r}}{\Delta, \Delta' \vdash C_r}\ \mathsf{multicut}$$

The condition that $n+2 \in \mu(m)$ follows from $\mathsf{C} \in \sigma(m)$.

The price for generalizing cut has to be paid somewhere, if not in contraction. We show only one such case, which illustrates the new form of the principal cases. We omit some conditions on dependence and multiplicity for the sake of brevity.

$$\dfrac{\dfrac{\mathcal{D}_1 \quad \mathcal{D}_2}{\Delta \vdash A_m \quad \Delta \vdash B_m}\,\&R \qquad \dfrac{\mathcal{E}'}{\Delta', (A_m \mathbin{\&} B_m)^n, A_m \vdash C_r}\,\&L_1}{\Delta, \Delta' \vdash C_r}\,\mathsf{multicut}_{A_m \& B_m}$$

$$\longrightarrow \quad \dfrac{\mathcal{D}_1 \quad \dfrac{\dfrac{\mathcal{D}_1 \quad \mathcal{D}_2}{\Delta \vdash A_m \quad \Delta \vdash B_m}\,\&R \qquad \mathcal{E}'}{\Delta, (A_m \mathbin{\&} B_m)^n, A_m \vdash C_r}\,\mathsf{multicut}_{A_m \& B_m}}{\Delta \vdash A_m \qquad \Delta, \Delta', A_m \vdash C_r}\,\mathsf{multicut}_{A_m}$$

$$\Delta, \Delta, \Delta' \vdash C_r$$

What saves us here is that the upper of the two multicuts has the same proposition $(A_m \mathbin{\&} B_m)$ and the same first premise $\mathcal{D}$, but a smaller second premise $\mathcal{E}' < \mathcal{E}$. The lower multicut has a potentially much larger second premise, but is only on A_m so is smaller by our lexicographic induction ordering (first on the structure of the cut formula, and then on the structure of the left and right derivations).

We see that with multicut, every left rule applies to one of $n+1$ copies of a principal proposition, after which are n copies remaining.

In the case of weakening (or cut of zero propositions) we can directly construct a proof of the conclusion with using $\mathcal{D}$.

$$\dfrac{\Delta \geq m \geq r \quad 0 \in \mu(m) \quad \mathcal{D} \quad \Delta \vdash A_m \quad \mathcal{E} \quad \Delta', (A_m)^0 \vdash C_r}{\Delta, \Delta' \vdash C_r}\,\mathsf{multicut}_{A_m}$$

$$\longrightarrow \quad \dfrac{\mathcal{E}}{\Delta', (A_m)^0 \vdash C_r}\,\mathsf{weaken}^*$$

$$\Delta, \Delta' \vdash C_r$$

For the correctness we see that $0 \in \mu(m)$ implies that $\mathsf{W} \in \sigma(m)$, and since $\Delta \geq m$ we also have $\mathsf{W} \in \sigma(\ell)$ for every B_ℓ in Δ. Furthermore, $(A_m)^0$ means zero copies of A_m.

3 Inversion

We have talked about right and left invertibility of connectives a lot in this course. Not only is it important for proof search, but it also affects the operational interpretation. For example, channels of invertible type will receive under our message-passing interpretation.

We now want to refine the proof system for the sequent calculus so that inversion is *forced*, that is, inversion *must* be applied during search. This is not immediately relevant to the computational interpretation of proof reduction since it limits program expression.

It is a recurring theme of this course that we express ideas via rules of inference. We will do so here. Fundamentally, the idea is to take away all choice during proof search as long as invertible rules apply. We also remind ourselves of positive and negative proposition.

$$\begin{array}{lrcl}
\text{Negatives} & A^-_m, B^-_m & ::= & P^-_m \mid A_m \to B_m \mid A_m \mathbin{\&} B_m \mid \top \mid \uparrow^m_k A_k \; [\mid \langle P^+_m \rangle] \\
\text{Positives} & A^+_m, B^+_m & ::= & P^+_m \mid A_m \times B_m \mid \mathbf{1} \mid A_m + B_m \mid \mathbf{0} \mid \downarrow^\ell_m A_\ell \; [\mid \langle P^-_m \rangle] \\
\text{Propositions} & A_m, B_m & ::= & A^-_m \mid B^-_m
\end{array}$$

We have not fully *polarized* the propositions which would mean to continue with negative or positive subformulas until there is an explicit change of polarity via a polarity-changing modality. We do this to reduce syntactic complexity since we already use shifts for different purpose (namely switching between modes). We explain *suspended atomic propositions* $\langle P^-_m \rangle$ and $\langle P^+_m \rangle$ later.

We walk through the inference rules the way you might discover them. We start with negative propositions in the succedent. We write $\xrightarrow{\mathsf{IR}} A_m$ for a judgment that forces inversion to be applied to the succedent until it is no longer possible. We have omitted the antecedents until we see what they might need to look like.

$$\frac{\xrightarrow{\mathsf{IR}} A_m \quad \xrightarrow{\mathsf{IR}} B_m}{\xrightarrow{\mathsf{IR}} A_m \mathbin{\&} B_m} \mathbin{\&}R \qquad \frac{}{\xrightarrow{\mathsf{IR}} \top} \top R$$

$$\frac{A_m \xrightarrow{\mathsf{IR}} B_m}{\xrightarrow{\mathsf{IR}} A_m \to B_m} \to R$$

We see that $\to R$ introduces an antecedent that may or may not be invertible. We want to make sure that there is no choice so we force right inversion and accumulate antecedents until right rules can no longer be applied. The accumulator is *ordered* (rather than linear) so we can process it deterministically during the left

inversion phase. We also add the upshift.

$$\frac{\Omega \xrightarrow{\mathsf{IR}} A_m \quad \Omega \xrightarrow{\mathsf{IR}} B_m}{\Omega \xrightarrow{\mathsf{IR}} A_m \mathbin{\&} B_m}\ \&R \qquad \frac{}{\Omega \xrightarrow{\mathsf{IR}} \top}\ \top R$$

$$\frac{A_m\, \Omega \xrightarrow{\mathsf{IR}} B_m}{\Omega \xrightarrow{\mathsf{IR}} A_m \to B_m}\ \to R \qquad \frac{\Omega \xrightarrow{\mathsf{IR}} A_k}{\Omega \xrightarrow{\mathsf{IR}} \uparrow_k^m A_k}\ \uparrow R$$

At this point we miss negative atoms P_m^- and positive propositions A_m^+. In both cases, the right inversion phase comes to an end and we switch to perform possible inversions on Ω. For negative atoms, in a way, we *should* still invert but we can't since there are no subformulas. So we *suspend* this proposition and count it, for the purpose of our judgment, as a negative proposition. It is important to recognize that the angle brackets $\langle P_m^- \rangle$ are only a syntactic ("judgmental") marker and not a propositional modality.

$$\frac{\Omega \xrightarrow{\mathsf{IL}} C_m^+}{\Omega \xrightarrow{\mathsf{IR}} C_m^+}\ \mathsf{IL/IR}^+ \qquad \frac{\Omega \xrightarrow{\mathsf{IL}} \langle P_m^- \rangle}{\Omega \xrightarrow{\mathsf{IR}} P_m^-}\ \mathsf{IL/IR}^*$$

Now left inversion peels off left invertible propositions from the left end of Ω. These are, of course, the positive propositions.

$$\frac{A_m\, B_m\, \Omega \xrightarrow{\mathsf{IL}} C_r^+}{(A_m \times B_m)\, \Omega \xrightarrow{\mathsf{IL}} C_r^+}\ \times L \qquad \frac{\Omega \xrightarrow{\mathsf{IL}} C_r^+}{\mathbf{1}\, \Omega \xrightarrow{\mathsf{IL}} C_r^+}\ \mathbf{1}L$$

$$\frac{A_m\, \Omega \xrightarrow{\mathsf{IL}} C_r^+ \quad B_m\, \Omega \xrightarrow{\mathsf{IL}} C_r^+}{(A_m + B_m)\, \Omega \xrightarrow{\mathsf{IL}} C_r^+}\ +L \qquad \frac{}{\mathbf{0}\, \Omega \xrightarrow{\mathsf{IL}} C_r^+}\ \mathbf{0}L$$

What happens when we reach a negative proposition or a positive atom? We have to postpone dealing with them because we want to force inversion, so we need another linear zone Δ^- consisting only of negative propositions (including positive atoms). We move negative propositions into Δ' when they pop up in Ω.

$$\frac{\Delta^-, A_m^- \,;\, \Omega \xrightarrow{\mathsf{IL}} C_r^+}{\Delta^- \,;\, A_m^-\, \Omega \xrightarrow{\mathsf{IL}} C_r^+}\ \mathsf{IL}^+ \qquad \frac{\Delta^-, \langle P_m^+ \rangle \,;\, \Omega \xrightarrow{\mathsf{IL}} C_r^+}{\Delta^- \,;\, P_m^+\, \Omega \xrightarrow{\mathsf{IL}} C_r^+}\ \mathsf{IL}^*$$

Now we have to add Δ^- to all the earlier rules and propagate them unchanged from conclusion to premise. Sigh.

When the ordered antecedents become empty the inversion phase comes to an end and we have to make an actual choice. We represent this judgment as $\Delta^- \xrightarrow{\mathsf{C}} C_r^+$.

$$\frac{\Delta^- \xrightarrow{\mathsf{C}} C_r^+}{\Delta^- \,;\, \cdot \xrightarrow{\mathsf{IL}} C_r^+}\ \mathsf{C/IL}$$

4 Chaining

Once the inversion phase is over, we could just make a choice of applying a left rule to a proposition in Δ^- or a right rule to C_r^+. This would be captured by recapping all the noninvertible rules for positive propositions on the right and negative propositions on the left.

Nondeterminism is further drastically reduced if we *focus* on a proposition and then chain the rules on this particular proposition until we switch back to an invertible one.

Unfortunately, I made a significant error in lecture in that the rules I showed only work as given in the case where there are no structural rules are allowed. Or, to put it another way, if all modes are linear and do not allow weakening or contraction, only exchange (which is always implicit). We show these rules and fix our mistake in a future lecture.

For the remainder of this section, all modes must be linear.

The first step is to select a negative antecedent or the succedent for focus. We indicate focus by using [square brackets]. Note that only a single proposition can be in focus in a given sequent.

$$\frac{\Delta^- \xrightarrow{\mathsf{FR}} [C_m^+]}{\Delta^- \xrightarrow{\mathsf{C}} C_m^+}\ \mathsf{FR/C} \qquad \frac{\Delta^- \,;\, [A^-] \xrightarrow{\mathsf{FL}} C_m^+}{\Delta^-, A^- \xrightarrow{\mathsf{C}} C_m^+}\ \mathsf{FL/C}$$

We start with the right rules. They are the usual right rules, but they retain focus

on the subformulas.

$$\frac{\Delta_1^- \xrightarrow{\mathsf{FR}} [A_m] \quad \Delta_2^- \xrightarrow{\mathsf{FR}} [B_m]}{\Delta_1^-, \Delta_2^- \xrightarrow{\mathsf{FR}} [A_m \times B_m]}\ \times R \qquad \frac{}{\cdot \xrightarrow{\mathsf{FR}} [\mathbf{1}]}\ \mathbf{1}R$$

$$\frac{\Delta^- \xrightarrow{\mathsf{FR}} [A_m]}{\Delta^- \xrightarrow{\mathsf{FR}} [A_m + B_m]}\ +R_1 \quad \frac{\Delta^- \xrightarrow{\mathsf{FR}} [B_m]}{\Delta^- \xrightarrow{\mathsf{FR}} [A_m + B_m]}\ +R_2 \qquad \text{no } \mathbf{0}R \text{ rule}$$

$$\frac{\Delta^- \geq \ell \quad \Delta^- \xrightarrow{\mathsf{FR}} [A_\ell]}{\Delta^- \xrightarrow{\mathsf{FR}} [\downarrow_m^\ell A_\ell]}\ \downarrow R$$

$$\frac{\Delta^- \,;\, \cdot \xrightarrow{\mathsf{IR}} A_m^-}{\Delta^- \xrightarrow{\mathsf{FR}} [A_m^-]}\ \mathsf{FR}^- \qquad \frac{}{\langle P_m^+ \rangle \xrightarrow{\mathsf{FR}} [P_m^+]}\ \mathsf{id}^+$$

The left rules are also the usual left rules, but the principal formula must be in focus.

$$\frac{\Delta_1^- \geq m \quad \Delta_1^- \xrightarrow{\mathsf{FR}} [A_m] \quad \Delta_2^- \,;\, [B_m] \xrightarrow{\mathsf{FL}} C_r^+}{\Delta_1^-, \Delta_2^- \,;\, [A_m \to B_m] \xrightarrow{\mathsf{FL}} C_r^+}\ \to L$$

$$\frac{\Delta^- \,;\, [A_m] \xrightarrow{\mathsf{FL}} C_r^+}{\Delta^- \,;\, [A_m \,\&\, B_m] \xrightarrow{\mathsf{FL}} C_r^+}\ \&L_1 \quad \frac{\Delta^- \,;\, [B_m] \xrightarrow{\mathsf{FL}} C_r^+}{\Delta^- \,;\, [A_m \,\&\, B_m] \xrightarrow{\mathsf{FL}} C_r^+}\ \&L_2 \qquad \text{no } \top L \text{ rule}$$

$$\frac{k \geq r \quad \Delta^- \,;\, [A_k] \xrightarrow{\mathsf{FL}} C_r^+}{\Delta^- \,;\, [\uparrow_k^m A_k] \xrightarrow{\mathsf{FL}} C_r^+}\ \uparrow L$$

$$\frac{\Delta^- \,;\, A_m^+ \xrightarrow{\mathsf{IL}} C_r^+}{\Delta^- \,;\, [A_m^+] \xrightarrow{\mathsf{FL}} C_r^+}\ \mathsf{FL}^+ \qquad \frac{}{\Delta^- \,;\, [P_m^-] \xrightarrow{\mathsf{FL}} \langle P_m^- \rangle}\ \mathsf{id}^-$$

The noninvertible rules constitute a phase of *chaining*. Taken together with *inversion* this proof search strategy is called *focusing*.

5 A Simple Example

As a simple example, consider

$$(P_m \to Q_m) \to ((Q_m \to R_m) \to (P_m \to R_m))$$

We can assign any polarity to the atoms we like and we choose all of them to be positive. The we start with right inversion until we hit the positive atom.

$$\dfrac{\dfrac{\vdots}{\cdot\,;\, P_m^+\ (Q_m^+ \to R_m^+)\ (P_m^+ \to Q_m^+) \xrightarrow{\mathsf{IR}} R_m^+}}{\cdot\,;\,\cdot \xrightarrow{\mathsf{IR}} (P_m^+ \to Q_m^+) \to ((Q_m^+ \to R_m^+) \to (P_m^+ \to R_m^+))}\ {\to}R \times 3$$

Now we switch to left inversion until we complete inversion and reach a choice.

$$\dfrac{\dfrac{\dfrac{\dfrac{\langle P_m^+\rangle, Q_m^+ \to R_m^+, P_m^+ \to Q_m^+ \xrightarrow{\mathsf{C}} R_m^+}{\cdots}}{\cdot\,;\, P_m^+\ (Q_m^+ \to R_m^+)\ (P_m^+ \to Q_m^+) \xrightarrow{\mathsf{IL}} R_m^+}\ \mathsf{IL/IR}}{\cdot\,;\, P_m^+\ (Q_m^+ \to R_m^+)\ (P_m^+ \to Q_m^+) \xrightarrow{\mathsf{IR}} R_m^+}}{\cdot\,;\,\cdot \xrightarrow{\mathsf{IR}} (P_m^+ \to Q_m^+) \to ((Q_m^+ \to R_m^+) \to (P_m^+ \to R_m^+))}\ {\to}R \times 3$$

This is a critical point in the search.

1. We cannot focus on R_m^- because $\langle R_m^+\rangle$ is not among the antecedents.
2. We cannot focus on $Q_m^+ \to R_m^+$ because Q_m^+ is not among the antecedents.
3. We cannot focus on $\langle P_m^+\rangle$ because it is a suspended atom.

The only choice that remains is to focus on $P_m^+ \to Q_m^+$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\langle Q_m^+\rangle, Q_m^+ \to R_m^+ \xrightarrow{\mathsf{C}} R_m^+}\ \mathsf{C/IL}}{\langle Q_m^+\rangle, Q_m^+ \to R_m^+\,;\,\cdot \xrightarrow{\mathsf{IL}} R_m^+}\ \mathsf{IL}^*}{\langle P_m^+\rangle\,;\,\cdot \xrightarrow{\mathsf{FR}} [P_m^+]}\ \mathsf{id}^+ \qquad \dfrac{Q_m^+ \to R_m^+\,;\, Q_m^+ \xrightarrow{\mathsf{IL}} R_m^+}{Q_m^+ \to R_m^+\,;\, [Q_m^+] \xrightarrow{\mathsf{FL}} R_m^+}\ \mathsf{IL/FL}}{\langle P_m^+\rangle, Q_m^+ \to R_m^+\,;\, [P_m^+ \to Q_m^+] \xrightarrow{\mathsf{FL}} R_m^+}\ {\to}L}{\langle P_m^+\rangle, Q_m^+ \to R_m^+, P_m^+ \to Q_m^+ \xrightarrow{\mathsf{C}} R_m^+}\ \mathsf{FL/C}}{\cdots}}{\cdot\,;\, P_m^+\ (Q_m^+ \to R_m^+)\ (P_m^+ \to Q_m^+) \xrightarrow{\mathsf{IL}} R_m^+}\ \mathsf{IL/IR}}{\cdot\,;\, P_m^+\ (Q_m^+ \to R_m^+)\ (P_m^+ \to Q_m^+) \xrightarrow{\mathsf{IR}} R_m^+}}{\cdot\,;\,\cdot \xrightarrow{\mathsf{IR}} (P_m^+ \to Q_m^+) \to ((Q_m^+ \to R_m^+) \to (P_m^+ \to R_m^+))}\ {\to}R \times 3$$

At this point once again the only possibility is to focus on $Q_m^+ \to R_m^+$, after which we can focus on R_m^+ in the succedent.

Even though it looks complex, there is just one proof and (excepting shallow backtracking) only one way to construct this proof.

The mistake I made in lecture, by the way, is that I claimed the FL/C rule was the only one where weakening and contraction came into play. There are actually several others, so we postpone a full discussion to a future lecture.

References

Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):197–347, 1992.

Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, November 2009.

Klaas Pruiksma, William Chargin, Frank Pfenning, and Jason Reed. Adjoint logic. Unpublished manuscript, April 2018. URL http://www.cs.cmu.edu/~fp/papers/adjoint18b.pdf.

Lecture Notes on Quantifiers

15-836: Substructural Logics
Frank Pfenning

Lecture 13
October 10, 2023

1 Introduction

In the development of logical inference, we used propositions such as $\mathsf{path}(x, y)$ or $\mathsf{proc}(P)$ in the rules, but we did not explicitly quantify over them. That's because the schematic variables in inference rules are *implicitly* universally quantified. The lack of quantifiers was then a problem when we tried to internalize the rules as logical propositions. For example, we'd like to translate the rule on the left to the proposition on the right.

$$\frac{\mathsf{path}(x, y) \quad \mathsf{path}(y, z)}{\mathsf{path}(x, z)} \qquad\qquad \forall x.\, \forall y.\, \forall z.\, \mathsf{path}(x, y) \land \mathsf{path}(y, z) \supset \mathsf{path}(x, z)$$

On the logical side, introducing quantifiers is interesting and not particularly disturbing. When we think of propositions as types (and proofs as programs), then there are multiple ways to think about quantifiers, and none are particularly canonical unless we go all the way to type theory in which we can talk (and therefore quantify) over proofs. Even there, matters are complicated specifically when we consider substructural type theories.

Today, we'll stay comfortably in logic and logical inference. We'll go with ordered logic as our base because this is the context we have thought the most about cut elimination. Our considerations carry over to linear, structural, and adjoint logics.

2 Universal Quantification

One tricky aspect of quantification is the *domain* we quantify over. For example, in the reachability example of the introduction, the quantifiers range over nodes in a graph. Or we may quantify over natural numbers, or trees, etc. When we go

to a full type theory, this issue must be faced and deeply considered. In predicate calculus (or first-order logic, as it is often called) it is convenient to consider the laws of the quantifiers independently from the domain of quantification. In other words, we'd like to investigate the rules of logical reasoning *independently* of the individuals we quantify over. This is not unlike the step we took when we investigated the logical connectives. We postulated some atomic propositions P, but the our reasoning did not depend on them. Once we fix a domain of interest, say, the natural numbers, we are no longer in the pure predicate calculus; instead we are reasoning in *arithmetic*, or the theory of lists, or trees, etc.

Diving in now: when can we prove $\forall i.\, A(i)$, not considering the domain of quantification? We can prove it if we can prove $A(a)$ for an arbitrary individual n. So we could write

$$\frac{\Omega \vdash A(n)}{\Omega \vdash \forall i.\, A(i)}\ \forall R^n$$

where the superscript n means that n must be *fresh*: it cannot already occur in Ω or $\forall i.\, A(i)$. This condition is crucial. Otherwise, we could for example prove that $A(n) \supset \forall i.\, A(i)$, that is, if A holds for some individual n then it holds for all individuals. This concept is so important it has its own name: n is an *eigenvariable* of the inference, and the proof of the premise is *parametric in* n.

Similar freshness conditions applied to proof terms that introduced variables, and we managed them through explicit naming of all antecedents. We stick here to the same form of variable hygiene and introduce a structural context naming all the individuals that may appear in a sequent.

$$\underbrace{i_1\ ind, \ldots, i_k\ ind}_{\Gamma}\ ;\ \Omega \vdash A$$

Our *presupposition* is that all variables $i_1, \ldots, i_k$ are distinct, and that all variables occurring in Ω and A are declared in Γ. We then obtain the rule

$$\frac{\Gamma, i\ ind\ ;\ \Omega \vdash A(i)}{\Gamma\ ;\ \Omega \vdash \forall i.\, A(i)}\ \forall R$$

Since the condition on the eigenvariable is now enforced by our presupposition we no longer annotate the rule. As for proof terms, we write the same variable name i, but bound variables (as in $\forall i.\, A(i)$) can be silently renamed in order to make the rule applicable in the form presented.

Next, we need to consider the matching left rule. If we know $A(i)$ is true for an arbitrary individual i, we should be able to instantiate it with any individual.

$$\frac{\Gamma \vdash t\ ind \quad \Gamma\ ;\ \Omega_L\ A(t)\ \Omega_R \vdash C}{\Gamma\ ;\ \Omega_L\ (\forall i.\, A(i))\ \Omega_R \vdash C}\ \forall L$$

Here, t is a term in our logical language denoting an individual that uses only variable from Γ. This condition is necessary because otherwise the premise with $A(t)$ may no longer satisfy our presupposition. The minimal choice for the judgment $\Gamma \vdash t\ ind$ is that the term t is a variable declared in Γ. Depending on our intentions, we could also have other terms denoting individuals, such as $\mathsf{zero}, \mathsf{succ}(\mathsf{zero}), \mathsf{succ}(i)$, etc.

Also, we see that we treat Γ *structurally* rather than linearly or in an ordered fashion. Intuitively, that's because individuals are truly *used* in a proof, they are merely *mentioned* inside proposition. Therefore, terms that are meaningful (that is, in scope and well-formed) can be mentioned arbitrarily and are themselves not subject to a substructural discipline even if we are reasoning within one.

Of course, now we need to check for harmony.

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}'\\ \Gamma, i\ ind\ ;\ \Omega \vdash A(i)\end{array}}{\Gamma\ ;\ \Omega \vdash \forall i.\, A(i)}\ \forall R \qquad \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2\\ \Gamma \vdash t\ ind & \Gamma\ ;\ \Omega_L\ A(t)\ \Omega_R \vdash C\end{array}}{\Gamma\ ;\ \Omega_L\ (\forall i.\, A(i))\ \Omega_R \vdash C}\ \forall L}{\Gamma \vdash \Omega_L\ \Omega\ \Omega_R \vdash C}\ \mathsf{cut}_{\forall i.\, A(i)}$$

It is not immediately clear how to proceed because the propositions $A(i)$ and $A(t)$ in the premises do not match. This is precisely why we had to choose i to be fresh: so we could substitute t for i in the sequent and in fact in the whole derivation. Assuming for the moment this is possible, we obtain

$$\longrightarrow \qquad \dfrac{\begin{array}{cc}[t/i]\mathcal{D}' & \mathcal{E}_2\\ \Gamma\ ;\ \Omega \vdash A(t) & \Gamma\ ;\ \Omega_L\ (A(t))\ \Omega_R \vdash C\end{array}}{\Gamma \vdash \Omega_L\ \Omega\ \Omega_R \vdash C}\ \mathsf{cut}_{A(t)}$$

First, we should see why $A(t)$ is smaller than $\forall i.\, A(i)$. The term t could contain constructors as indicated above so it could be arbitrarily large. On the other hand, in a predicate calculus/first-order logic we distinguish between individuals and propositions, so the term t cannot contain any propositions. Therefore, if we count quantifiers and logical connectives, then $A(t)$ is smaller than $\forall i.\, A(i)$.

Second, we should verify that $[t/i]\mathcal{D}'$ can always be constructed as a proof of $\Gamma\ ;\ \Omega \vdash A(t)$. This follows from the *substitution principle* for individuals:

$$\dfrac{\begin{array}{cc}\Gamma \vdash t\ ind & \Gamma, i\ ind\ ;\ \Omega(i) \vdash A(i)\end{array}}{\Gamma\ ;\ \Omega(t) \vdash A(t)}\ \mathsf{subst}$$

We call this a substitution principle because we obtain the resulting derivation simply by substituting t for i. It is proved by induction over the second given deriva-

tion. The particular application of this rule here is:

$$\frac{\mathcal{E}_2 \qquad\qquad \mathcal{D}' \\ \Gamma \vdash t\ \mathit{ind} \quad \Gamma, i\ \mathit{ind}\ ; \Omega \vdash A(i)}{\Gamma\ ; \Omega \vdash A(t)}\ \mathsf{subst}$$

We know from the the shape of $\mathcal{D}$ and our presuppositions that the ordered antecedents in $\mathcal{D}'$ do not depend in i.

We can check that our transcription of the inference rules is correct. Recall that top-down inference is turned into bottom-up inference among the antecedents. In this example, we ignore issues of order.

$$\frac{\mathsf{path}(a,b) \quad \mathsf{path}(b,c)}{\mathsf{path}(a,c)}\ \mathsf{trans}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{path}(a,b) \vdash \mathsf{path}(a,b)}\ \mathsf{id} \quad \dfrac{}{\mathsf{path}(b,c) \vdash \mathsf{path}(b,c)}\ \mathsf{id}}{\mathsf{path}(a,b), \mathsf{path}(b,c) \vdash \mathsf{path}(a,b) \wedge \mathsf{path}(b,c)}\ \wedge R \qquad \mathsf{path}(a,c) \vdash C}{\mathsf{path}(a,b) \wedge \mathsf{path}(b,c) \supset \mathsf{path}(a,c), \quad \mathsf{path}(a,b), \mathsf{path}(b,c), \vdash C}\ \supset L}{\forall z.\, \mathsf{path}(a,b) \wedge \mathsf{path}(b,z) \supset \mathsf{path}(a,z), \quad \mathsf{path}(a,b), \mathsf{path}(b,c) \vdash C}\ \forall L}{\forall y.\, \forall z.\, \mathsf{path}(a,y) \wedge \mathsf{path}(y,z) \supset \mathsf{path}(a,z), \quad \mathsf{path}(a,b), \mathsf{path}(b,c), \vdash C}\ \forall L}{\forall x.\, \forall y.\, \forall z.\, \mathsf{path}(x,y) \wedge \mathsf{path}(y,z) \supset \mathsf{path}(x,z), \quad \mathsf{path}(a,b), \mathsf{path}(b,c) \vdash C}\ \forall L$$

3 Existential Quantification

We expect the rules for existential quantification to mirror those for universal quantification, reversing the role of the antecedent and succedent.

$$\frac{\Gamma \vdash t\ \mathsf{ind} \quad \Gamma\ ; \Omega \vdash A(t)}{\Gamma\ ; \Omega \vdash \exists i.\, A(i)}\ \exists R \qquad \frac{\Gamma, i\ \mathit{ind}\ ; \Omega_L\ A(i)\ \Omega_R \vdash C}{\Gamma\ ; \Omega_L\ (\exists i.\, A(i))\ \Omega_R \vdash C}\ \exists L$$

We show the reduction, even if it is straightforward after what we discussed for the universal.

$$\dfrac{\dfrac{\mathcal{D}_1 \qquad \mathcal{D}_2}{\Gamma \vdash t\ \mathsf{ind} \quad \Gamma\ ;\ \Omega \vdash A(t)}\ \exists R \qquad \dfrac{\mathcal{E}'}{\Gamma, i\ ind\ ;\ \Omega_L\ A(i)\ \Omega_R \vdash C}\ \exists L}{\Gamma\ ;\ \Omega_L\ \Omega\ \Omega_R \vdash C}\ \mathsf{cut}_{\exists i.\ A(i)}$$

$$\longrightarrow \quad \dfrac{\mathcal{D}_2 \qquad \dfrac{\mathcal{D}_1 \qquad \mathcal{E}'}{\Gamma \vdash t\ \mathsf{ind} \quad \Gamma, i\ \mathsf{ind}\ ;\ \Omega_L\ A(i)\ \Omega_R \vdash C}\ \mathsf{subst}}{\Gamma\ ;\ \Omega \vdash A(t) \qquad \Gamma\ ;\ \Omega_L\ A(t)\ \Omega_R \vdash C}\ \mathsf{cut}_{A(t)}$$
$$\Gamma\ ;\ \Omega_L\ \Omega\ \Omega_R \vdash C$$

So where do we use the existential? It turns out it allows us to express freshness conditions in multiset rewriting rules. We consider the dynamics of cut, written

$$\frac{\mathsf{proc}(x \leftarrow P(x)\ ;\ Q(x))}{\mathsf{proc}(P(a)) \quad \mathsf{proc}(Q(a))}\ (a\ \text{fresh})$$

As mentioned in an earlier lecture, the freshness condition here is somewhat strange: a must be *globally fresh* for the whole configuration, not just with respect to $P(x)$ and $Q(x)$. When transcribed into a logical proposition, this freshness condition turns into an *existential* quantifier.

$$\forall P.\, \forall Q.\, \mathsf{proc}(x \leftarrow P(x)\ ;\ Q(x)) \multimap \exists a.\, \mathsf{proc}(P(a)) \otimes \mathsf{proc}(Q(a))$$

Here, P and Q act as abstractions over channels, a detail we ignore until maybe a future lecture. The point is that when using this proposition as an antecedent, we will arrive at a sequent

$$\Gamma\ ;\ \exists a.\, \mathsf{proc}(P(a)) \otimes \mathsf{proc}(Q(a)), \Delta \vdash C$$

where $\Gamma = (a_1\ ind, \ldots, a_k\ ind)$ covers all the channels that might occur in P, Q, Δ, or C. The variable a is still bound, and now the left rule will have to pick a globally fresh parameter for it and then break down the tensor.

$$\dfrac{\dfrac{\Gamma, a\ ind\ ;\ \mathsf{proc}(P(a)), \mathsf{proc}(Q(a)), \Delta \vdash C}{\Gamma, a\ ind\ ;\ \mathsf{proc}(P(a)) \otimes \mathsf{proc}(Q(a)), \Delta \vdash C}\ \otimes L}{\Gamma\ ;\ \exists a.\, \mathsf{proc}(P(a)) \otimes \mathsf{proc}(Q(a)), \Delta \vdash C}\ \exists L$$

This reasoning points out several things. First, the freshness condition in our dynamics is a manifestation of the existential quantifier at the propositional level. Second, in our formulation of the dynamics of MPASS we could have been more explicit by keeping, on the side, a collection of all the channels in the configuration.

4 Polarities

We can apply our quick test for the polarities, which is to check the first step in the identity expansion.

$$\dfrac{\dots\dots\dots\dots\dots\dots\dots\dots}{\Gamma\ ;\ \forall j.\ A(j) \vdash \forall i.\ A(i)}\ \mathsf{id}_{\forall i.\ A(i)} \quad \longrightarrow_E \quad \dfrac{\dfrac{\dfrac{}{\Gamma, i\ ind \vdash i\ ind} \quad \dfrac{\dots\dots\dots\dots\dots\dots\dots\dots}{\Gamma, i\ ind\ ;\ A(i) \vdash A(i)}\ \mathsf{id}_{A(i)}}{\Gamma, i\ ind\ ;\ \forall j.\ A(j) \vdash A(i)}\ \forall L}{\Gamma\ ;\ \forall j.\ A(j) \vdash \forall i.\ A(i)}\ \forall R$$

While not proof, this indicates that the universal quantifier is *negative*. It does show that it is not positive, because the left rules for universal quantification is not invertible: we cannot instantiate the quantifier in the antecedent with a term until we have such a term.

By the way, we had to add Γ to the statement of the identity because our presupposition requires that all free variables in the sequent are collected in Γ. Without Γ the identity would be restricted to closed propositions A, which is far from general enough.

We expect that the existential, somehow symmetric to the universal, would be positive and invertible on the left, and this is indeed the case although we don't bother showing the details here.

Lecture Notes on
Semi-Axiomatic Sequent Calculus

15-836: Substructural Logics
Frank Pfenning

Lecture 14
October 26, 2023

1 Introduction

Message-passing communication in MPASS, which is based on the linear sequent calculus, is *synchronous* in the sense that both sending and receiving are potentially blocking actions. This is often a convenient abstraction, but under the hood communication is usually *asynchronous* in the sense that sending does not block but receiving does. In other formalisms for concurrency such as the π-calculus, we have synchronous [Milner et al., 1992] and asynchronous [Boudol, 1992] versions. So it is natural to look for an *asynchronous* calculus based on the interpretation of linear propositions as session types. It turns out that such a calculus exists and uncovers several new connections, in particular to *futures* in functional programming languages [Halstead, 1985] that are usually thought of as a form of shared memory concurrency. In this lecture we will develop an asynchronous message-passing calculus.

We know that in the untyped setting, the synchronous π-calculus is more expressive than the asynchronous one [Palamidessi, 2003]. In the setting of session types, they turn out to have the same expressive power [Pfenning and Griffith, 2015], so we have to decide which formulation we would like to take as fundamental. Because of its (relative) proximity to an implementation and its connection to futures, we prefer the asynchronous version as long as we can still relate it to proof theory. It turns out that going down this path will also allow us to generalize from linear to structural and then general adjoint types, which seems difficult to do directly for the synchronous version.

2 The Origin of Synchronous Communication

We refresh our memory about synchronous communication in MPASS. We use internal choice as an example. When the provider sends a label k, the type of the channel changes from $\oplus\{\ell : A_\ell\}_{\ell\in L}$ to A_k.

$$\begin{array}{lll} & \mathsf{proc}(\mathbf{send}\ a\ k\ ;\ P), & \mathsf{proc}(\mathbf{recv}\ a\ (\ell \Rightarrow Q_\ell)_{\ell\in L}) \\ \longrightarrow & \mathsf{proc}(P) & \mathsf{proc}(Q_k) \qquad\qquad (k \in L) \end{array}$$

This communication is synchronous because sender and receive proceed to their respective continuations at once. This ultimately comes from the linear sequent calculus where the principal cases of cut reduction replace a cut of proposition $A \oplus B$ either by a cut of A or of B, with subderivations on both premises of the cut.

Let's also recall the typing rules where the change in type of the communication channel is clearly visible.

$$\frac{k \in L \quad \Delta \vdash P :: (x : A_k)}{\Delta \vdash \mathbf{send}\ x\ k\ ;\ P :: (x : \oplus\{\ell : A_\ell\}_{\ell\in L})}\ \oplus R$$

$$\frac{\Delta, x : A_\ell \vdash Q_\ell \quad (\forall \ell \in L)}{\Delta, x : \oplus\{\ell : A_\ell\}_{\ell\in L} \vdash \mathbf{recv}\ k\ (\ell \Rightarrow Q_\ell)_{\ell\in L} :: (z : C)}\ \oplus L$$

We confirm that the synchronous nature of communication directly derives from the synchronous nature of cut reduction. Writing out a binary case (as is customary in logic):

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}' \\ \Delta \vdash A\end{array}}{\Delta \vdash A \oplus B}\ \oplus R_1 \qquad \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ \Delta', A \vdash C & \Delta', B \vdash C\end{array}}{\Delta', A \oplus B \vdash C}\ \oplus L}{\Delta, \Delta' \vdash C}\ \mathsf{cut}_{A\oplus B}$$

$$\longrightarrow \qquad \dfrac{\begin{array}{cc}\mathcal{D}' & \mathcal{E}_1 \\ \Delta \vdash A & \Delta', A \vdash C\end{array}}{\Delta, \Delta' \vdash C}\ \mathsf{cut}_A$$

3 Continuation Channels instead of Continuation Processes

We cannot simply drop the continuation process P to make communication asynchronous, because messages could be received out of order and progress would be violated. For example:

$\mathsf{bin} = \oplus\{\mathsf{b0} : \mathsf{bin}, \mathsf{b1} : \mathsf{bin}, \mathsf{e} : \mathbf{1}\}$
$\mathit{one}\ (x : \mathsf{bin}) = \mathbf{send}\ x\ \mathsf{b1}\ ;\ \mathbf{send}\ x\ \mathsf{e}\ ;\ \mathbf{send}\ x\ (\,)$

If we now match up a process that sends a binary 1 and one that receives a binary number, all several kinds of mismatches can occur.

$$\begin{array}{l}\mathsf{proc}(\mathbf{call}\ \mathit{one}\ a), \mathsf{proc}(\mathbf{recv}\ a\ (\mathsf{b0} \Rightarrow Q_0 \mid \mathsf{b1} \Rightarrow Q_1 \mid \mathsf{e} \Rightarrow Q_e) \\ \longrightarrow^* \mathsf{proc}(\mathbf{send}\ a\ \mathsf{b1}), \mathsf{proc}(\mathbf{send}\ a\ \mathsf{e}), \mathsf{proc}(\mathbf{send}\ a\ (\,)), \\ \qquad \mathsf{proc}(\mathbf{recv}\ a\ (\mathsf{b0} \Rightarrow Q_0 \mid \mathsf{b1} \Rightarrow Q_1 \mid \mathsf{e} \Rightarrow Q_e))\end{array}$$

Each of the messages could interact with the receiver, which could be an immediate "message not understood" problem (when the message is $(\,)$), or a later one (when the message is e).

The way we solve this problem is to replace the *continuation process* of the sender by a *continuation channel*. Ignoring for the moment where these continuation channels would come from, we might write

$$\begin{array}{l}\mathsf{bin} = \oplus\{\mathsf{b0} : \mathsf{bin}, \mathsf{b1} : \mathsf{bin}, \mathsf{e} : \mathbf{1}\} \\ \mathit{one}\ (x : \mathsf{bin}) = \mathbf{send}\ x\ \mathsf{b1}(x')\ ;\ \mathbf{send}\ x'\ \mathsf{e}(x'')\ ;\ \mathbf{send}\ x''\ (\,) \qquad \text{\% approximately}\end{array}$$

We need to fix this later to account for the creation of the continuation channels. The receiver then not only selects the branch, but also receives a continuation channel for further communication.

$$\mathsf{proc}(\mathbf{recv}\ x\ (\mathsf{b0}(x') \Rightarrow Q_0(x') \mid \mathsf{b1}(x') \Rightarrow Q_1(x') \mid \mathsf{e}(x') \Rightarrow Q_e(x'))$$

The idea is that x is used in only one place, and then x' next, and then x'', etc. so channels and their types cannot be confused. This technique is due to Kobayashi et al. [1996].

Revisiting our rules, they now become:

$$\frac{k \in L}{x' : A_k \vdash \mathbf{send}\ x\ k(x') :: (x : \oplus\{\ell : A_\ell\}_{\ell \in L})}\ \oplus X$$

$$\frac{\Delta, x' : A_\ell \vdash Q_\ell(x') \quad (\forall \ell \in L)}{\Delta, x : \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \mathbf{recv}\ k\ (\ell(x') \Rightarrow Q_\ell(x'))_{\ell \in L} :: (z : C)}\ \oplus L$$

Note that the right rule has become an axiom, that is, it has no logical premises. This makes sense intuitively because when a message is received it should be consumed by the recipient. The left rule only changes in the sense that x in the premises has become the continuation channel x'.

In the reduction rule we see that the channel a no longer changes type, but communication is transferred from $a : \oplus\{\ell : A_\ell\}_{\ell \in L}$ to the continuation channel $a' : A_k$.

$$\begin{array}{rll} & \mathsf{proc}(\mathbf{send}\ a\ k(a')), & \mathsf{proc}(\mathbf{recv}\ a\ (\ell(x') \Rightarrow Q_\ell(x'))_{\ell \in L}) \\ \longrightarrow & & \mathsf{proc}(Q_k(a')) \qquad (k \in L)\end{array}$$

4 Back to Logic

We have seen that the dynamics, if it holds up to scrutiny in the logic, would allow for asynchronous communication. Extracting the binary logical rules we have:

$$\frac{}{A \vdash A \oplus B}\ \oplus X_1 \qquad \frac{}{B \vdash A \oplus B}\ \oplus X_2 \qquad\qquad \frac{\Delta, A \vdash C \quad \Delta, B \vdash C}{\Delta, A \oplus B \vdash C}\ \oplus L$$

The reductions:

$$\frac{\dfrac{}{A \vdash A \oplus B}\ \oplus X_1 \qquad \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ \Delta', A \vdash C & \Delta', B \vdash C\end{array}}{\Delta', A \oplus B \vdash C}\ \oplus L}{\Delta', A \vdash C}\ \mathsf{cut}_{A \oplus B} \quad \longrightarrow \quad \begin{array}{c}\mathcal{E}_1 \\ \Delta', A \vdash C\end{array}$$

$$\frac{\dfrac{}{B \vdash A \oplus B}\ \oplus X_2 \qquad \dfrac{\begin{array}{cc}\mathcal{E}_1 & \mathcal{E}_2 \\ \Delta', A \vdash C & \Delta', B \vdash C\end{array}}{\Delta', A \oplus B \vdash C}\ \oplus L}{\Delta', A \vdash C}\ \mathsf{cut}_{A \oplus B} \quad \longrightarrow \quad \begin{array}{c}\mathcal{E}_2 \\ \Delta', B \vdash C\end{array}$$

In both cases, the cut disappears (which corresponds to a message receipt) and only the recipient continues computation. The channel substitution is hidden in this proof notation. For example, in first of the two reduction we would replace the $x : A$ resulting from the case split by the $x' : A$ label in the conclusion which is the same as in the first premise. A symmetric case arises for the second reduction.

5 Generalizing to Other Connectives

We have seen that for internal choice the right rules turned into axioms (representing messages) and the left rule remained unchanged. In general, those rules that carry information should become messages (and thus axioms) while invertible rules should remain as they are. This means for positive connectives the right rules become axioms while the left rules remain.

We call the result the *semi-axiomatic sequent calculus* (SAX) because half the rules are turned into axioms while the other half remains the same.

$$\frac{}{A, B \vdash A \otimes B}\ \otimes X \qquad\qquad \frac{\Delta, A, B \vdash C}{\Delta, A \otimes B \vdash C}\ \otimes L$$

$$\frac{}{\cdot \vdash \mathbf{1}}\ \mathbf{1}X \qquad\qquad \frac{\Delta \vdash C}{\Delta, \mathbf{1} \vdash C}\ \mathbf{1}L$$

Because $\mathbf{1}R$ is already an axiom, it does not change but we have given it a new name, for uniformity. The negatives are symmetric in the sense that the left rules become axioms (they are the ones carrying information) while the right rules remain (they are invertible).

$$\frac{\Delta \vdash A \quad \Delta \vdash B}{\Delta \vdash A \mathbin{\&} B}\ \&R \qquad \frac{}{A \mathbin{\&} B \vdash A}\ \&X_1 \quad \frac{}{A \mathbin{\&} B \vdash B}\ \&X_2$$

$$\frac{\Delta, A \vdash B}{\Delta \vdash A \multimap B}\ \multimap R \qquad \frac{}{A, A \multimap B \vdash B}\ \multimap X$$

We also have identity and cut as usual.

$$\frac{}{A \vdash A}\ \mathsf{id} \qquad \frac{\Delta \vdash A \quad \Delta', A \vdash C}{\Delta, \Delta' \vdash C}\ \mathsf{cut}$$

Before we return to the computational meaning of these rules, we should ask the obvious questions: (1) if we replace the positive right and negative left rules by axioms do the same judgments hold, and (2) do cut and identity elimination still hold?

6 Relating Sequent Calculus to SAX

We conjecture that the ordinary and semi-axiomatic sequent calculi prove the same sequents. To show this, we will demonstrate how to derive the rules of each calculus in the other.

First, translating from SAX to the sequent calculus. By showing that the SAX rules are derivable in the sequent calculus we can conclude that SAX is *sound*. We only show two examples.

$$\frac{\dfrac{}{A \vdash A}\ \mathsf{id}_A}{A \vdash A \oplus B}\ \oplus R_1 \qquad \frac{\dfrac{}{A \vdash A}\ \mathsf{id}_A \quad \dfrac{}{B \vdash B}\ \mathsf{id}_B}{A, A \multimap B \vdash B}\ \multimap L$$

So, in general to derive the new axioms we just need identity, while the negative right and positive left rules remain unchanged.

Second, translating from the sequent calculus to SAX. We show that the sequent calculus rules are derivable in SAX. Again the rules that don't change are trivial. We show two other examples.

$$\frac{\Delta \vdash A \quad \dfrac{}{A \vdash A \oplus B}\ \oplus X_1}{\Delta \vdash A \oplus B}\ \mathsf{cut}_A \qquad \frac{\dfrac{\Delta \vdash A \quad \dfrac{}{A, A \multimap B \vdash B}\ \multimap X}{\Delta, A \multimap B \vdash B}\ \mathsf{cut}_A \qquad \Delta', B \vdash C}{\Delta, \Delta', A \multimap B \vdash C}\ \mathsf{cut}_B$$

So for this direction we need cut. We formulate this as a theorem.

Theorem 1 (Soundness and Completeness of SAX) $\Delta \vdash A$ *in the sequent calculus iff* $\Delta \vdash A$ *in SAX.*

Proof: We prove in each direction that the rules in the other calculus are derivable. This could be formalized as in induction over the structure of the given derivation.

From left to right we insert suitable cuts (as exemplified above) and from right to left we insert suitable identities (as exemplified above). □

7 Cut Elimination for SAX

The fact that the translation requires us to insert cuts suggests that SAX does not satisfy a traditional cut elimination result. You may want to construct a counterexample for yourself before moving on.

Here is a simple one for distinct atoms P, Q, and R.

$$Q \vdash (P \oplus Q) \oplus R$$

While easily provable in the sequent calculus, in SAX we are stuck right away. It is not the form of an axiom, so we can only proceed with cut.

This is profoundly saddening if you are a logical fundamentalist and proof-theorist. But it turns out that it is also an opportunity for new discoveries!

If you look at the sample cases where cut had to be inserted in the previous section, you see that the cuts have a special form: they only eliminate a *subformula* of the sequent we are trying to prove. In the first example we have A as a subformula of $A \oplus B$, in the second we have both A and B as subformulas of $A \multimap B$. In general such cuts are called *analytic cuts*. Here we call them *snips*, which is an even more restricted class than analytic cuts in the sense that one of the two premises of a snip must be an axiom or another snip. We elide a precise definition for now, but we will come back to it in a future lecture.

We can easily see that we can eliminate all cuts that are not snips. We do this by translating a SAX derivation to the sequent calculus, eliminating cut, and then translating back. This back translation will only require snips if the given sequent derivation is cut-free to start with.

This, however, is not fully satisfying since we would like to relate the rules of computation to cut reduction. Looking ahead, there is indeed a direct cut elimination algorithm utilizing the cut reductions in SAX that we have shown, leaving only snips. For a structural version of SAX, this is proved by DeYoung et al. [2020]—the linear version is significantly easier.

8 Completing Process Assignment and Dynamics

We now take the logical rules back into typing rules, adding continuation channels to all messages. Our convention to assign the name x' to the continuation of a

channel x. First, the positive connectives.

$$\frac{k \in L}{x' : A_k \vdash \mathbf{send}\ x\ k(x') :: (x : \oplus\{\ell : A_\ell\}_{\ell \in L})}\ \oplus X$$

$$\frac{\Delta, x' : A_\ell \vdash Q_\ell(x') \quad (\forall \ell \in L)}{\Delta, x : \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \mathbf{recv}\ x\ (\ell(x') \Rightarrow Q_\ell(x'))_{\ell \in L} :: (z : C)}\ \oplus L$$

$$\frac{}{y : A, x' : B \vdash \mathbf{send}\ x\ (y, x') :: (x : A \otimes B)}\ \otimes X$$

$$\frac{\Delta, y : A, x' : B \vdash Q(y, x') :: (z : C)}{\Delta, x : A \otimes B \vdash \mathbf{recv}\ x\ ((y, x') \Rightarrow Q(y, x')) :: (z : C)}\ \otimes L$$

$$\frac{}{\cdot \vdash \mathbf{send}\ x\ (\,) :: (x : \mathbf{1})}\ \mathbf{1}X \qquad \frac{\Delta \vdash Q :: (z : C)}{\Delta, x : \mathbf{1} \vdash \mathbf{recv}\ x\ (() \Rightarrow Q) :: (z : C)}\ \mathbf{1}L$$

Now the negatives.

$$\frac{\Delta \vdash P_\ell(x') :: (x' : A_\ell) \quad (\forall \ell \in L)}{\Delta \vdash \mathbf{recv}\ x\ (\ell(x') \Rightarrow P_\ell(x')) :: (x : \&\{\ell : A_\ell\}_{\ell \in L})}\ \&R$$

$$\frac{k \in L}{x : \&\{\ell : A_\ell\}_{\ell \in L} \vdash \mathsf{send}\ x\ k(x') :: (x' : A_k)}\ \&X$$

$$\frac{\Delta, y : A \vdash P(y, x') :: (x' : B)}{\Delta \vdash \mathbf{recv}\ x\ ((y, x') \Rightarrow P(y, x')) :: (x : A \multimap B)}\ \multimap R$$

$$\frac{}{y : A, x : A \multimap B \vdash \mathbf{send}\ x\ (y, x') :: (x' : B)}\ \multimap X$$

Cut and identity do not change from the sequent calculus.

$$\frac{}{y : A \vdash \mathbf{fwd}\ x\ y :: (x : A)}\ \mathsf{id} \qquad \frac{\Delta \vdash P(x) :: (x : A) \quad \Delta', x : A \vdash Q(x) :: (z : C)}{\Delta, \Delta' \vdash x_A \leftarrow P(x)\ ;\ Q(x) :: (z : C)}\ \mathsf{cut}$$

We refactor the dynamics as before.

$$
\begin{array}{llcll}
\text{Messages} & M & ::= & k(x') \mid (y, x') \mid ( ) & \\
\text{Continuations} & K & ::= & (\ell(x') \Rightarrow P_\ell(x'))_{\ell \in L} & (\oplus, \&) \\
 & & \mid & ((y, x') \Rightarrow P(y, x')) & (\otimes, \multimap) \\
 & & \mid & (( ) \Rightarrow P) & (\mathbf{1}) \\
\text{Processes} & P & ::= & x \leftarrow P(x) \,;\, Q(x) & \mathsf{cut} \\
 & & \mid & \mathbf{fwd}\ x\ y & \mathsf{id} \\
 & & \mid & \mathbf{send}\ x\ M & \\
 & & \mid & \mathbf{recv}\ x\ K & \\
 & & \mid & \mathbf{call}\ p\ x\ y_1 \ldots y_n &
\end{array}
$$

The dynamics in this refactored form relies on the $M \triangleright K$ operation defined just below. Recall that a global signature Σ contains (possibly mutually recursive) type and process definitions.

$$
\begin{array}{lcll}
\mathsf{proc}(x \leftarrow P(x) \,;\, Q(x)) & \longrightarrow & \mathsf{proc}(P(a)), \mathsf{proc}(Q(a)) & (a \text{ fresh}) \\
\mathsf{proc}(P(b)), \mathsf{proc}(\mathbf{fwd}\ a\ b) & \longrightarrow & \mathsf{proc}(P(a)) & \\
\mathsf{proc}(\mathbf{send}\ a\ M), \mathsf{proc}(\mathbf{recv}\ a\ K) & \longrightarrow & \mathsf{proc}(M \triangleright K) & \\
\mathsf{proc}(\mathbf{call}\ p\ a\ b_1 \ldots b_n) & \longrightarrow & \mathsf{proc}(P(a, b_1, \ldots, b_n)) & \\
 & & \text{for } p\ x\ y_1 \ldots y_n = P(x, y_1, \ldots, y_n) \in \Sigma &
\end{array}
$$

$$
\begin{array}{lcll}
k(a') \triangleright (\ell(x') \Rightarrow P_\ell(x'))_{\ell \in L} & = & P_k(a') & (k \in L) \\
(b, a') \triangleright ((y, x') \Rightarrow P(y, x')) & = & P(b, a') & \\
( ) \triangleright (( ) \Rightarrow P) & = & P &
\end{array}
$$

9 Example Revisited

Recall the earlier example, which wasn't quite right because we could not explain where the continuation channels would come from.

$$
\begin{array}{l}
\mathsf{bin} = \oplus\{\mathsf{b0} : \mathsf{bin}, \mathsf{b1} : \mathsf{bin}, \mathsf{e} : \mathbf{1}\} \\
\mathit{one}\ (x : \mathsf{bin}) = \mathbf{send}\ x\ \mathsf{b1}(x') \,;\, \mathbf{send}\ x'\ \mathsf{e}(x'') \,;\, \mathbf{send}\ x''\ ( ) \qquad \text{\% approximately}
\end{array}
$$

In SAX we have to explicitly allocate the continuation channels via cut. Because of the orientation of the cut, this requires us to reverse the textual order of the send actions.

$$
\begin{array}{l}
\mathit{one}\ (x : \mathsf{bin}) = x'' \leftarrow \mathbf{send}\ x''\ ( ) \,; \\
\qquad\qquad\qquad\quad x' \leftarrow \mathbf{send}\ x'\ \mathsf{e}(x'') \,; \\
\qquad\qquad\qquad\quad \mathbf{send}\ x\ \mathsf{b1}(x')
\end{array}
$$

Because of the concurrent nature of cut, the order is not significant for the computation of this process and we are left with three messages. These messages form a queue, with the continuation channels acting as "pointers".

$$\mathsf{proc}(\mathbf{call}\ \mathit{one}\ a) \longrightarrow^* \mathsf{proc}(\mathbf{send}\ a''\ (\,)), \mathsf{proc}(\mathbf{send}\ a'\ \mathsf{e}(a'')), \mathsf{proc}(\mathbf{send}\ a\ \mathsf{b1}(a'))$$

Syntactically, the opposite order of sends in the definition would be closer to MPASS. This can be achieved with a "reverse cut" where the client $Q(x)$ precedes the provider $P(x)$.

$$\frac{\Delta', x : A \vdash Q(x) :: (z : C) \quad \Delta \vdash P(x) :: (x : A)}{\Delta, \Delta' \vdash x \rightarrow Q(x)\ ;\ P(x) :: (z : C)}\ \mathsf{cut}^R$$

Since we just reverse the premises of the cut, this is just a syntactic convenience and does not change the essence of the language. Then we could write:

$$\begin{array}{l} \mathit{one}\ (x : \mathsf{bin}) = x' \rightarrow \mathbf{send}\ x\ \mathsf{b1}(x')\ ; \\ \qquad\qquad\quad\ x'' \rightarrow \mathbf{send}\ x'\ \mathsf{e}(x'')\ ; \\ \qquad\qquad\quad\ \mathbf{send}\ x''\ (\,) \end{array}$$

We might decide to pick a different concrete syntax for this, to be discussed in the next lecture.

During lecture, we also briefly discussed an alternative where x' is somehow computed from x, and that the sender and recipient agree on this computation. This could be concrete "address arithmetic" (like: from x we go to $x + 1$) or more abstract (like: from x we go to $x.\overline{\mathsf{b1}}$). This actually has a quite sensible *logical* interpretation in terms of *snips* from Section 7 so we will come back to it, probably two lectures from now. With this case we might write the process one as follows:

$$\mathit{one}\ (x : \mathsf{bin}) = \mathbf{send}\ x\ \mathsf{b1}(\_)\ ;\ \mathbf{send}\ x.\overline{\mathsf{b1}}\ \mathsf{e}(\_)\ ;\ \mathbf{send}\ x.\overline{\mathsf{b1}}.\overline{\mathsf{e}}\ (\,) \qquad \%\ \text{with snips}$$

We have elided the continuation channels in the send actions because they can be computed from the channel x.

References

Gérard Boudol. Asynchrony and the π-calculus. Rapport de Recherche 1702, INRIA, Sophia-Antipolis, 1992.

Henry DeYoung, Frank Pfenning, and Klaas Pruiksma. Semi-axiomatic sequent calculus. In Z. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, pages 29:1–29:22, Paris, France, June 2020. LIPIcs 167.

Robert H. Halstead. Multilisp: A language for parallel symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–539, October 1985.

Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In H.-J. Boehm and G. Steele, editors, *Proceedings of the 23rd Symposium on Principles of Programming Languages (POPL'96)*, pages 358–371, St. Petersburg Beach, Florida, USA, January 1996. ACM.

Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–77, September 1992. Parts I and II.

Catuscia Palamidessi. Comparing the expressive power of the synchronous and the asynchronous π-calculus. *Mathematical Structures in Computer Science*, 13(5): 685–719, 2003.

Frank Pfenning and Dennis Griffith. Polarized substructural session types. In A. Pitts, editor, *Proceedings of the 18th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2015)*, pages 3–22, London, England, April 2015. Springer LNCS 9034. Invited talk.

Lecture Notes on Adjoint SAX

15-836: Substructural Logics
Frank Pfenning

Lecture 15
October 31, 2023

1 Introduction

The version of SAX we introduced in the last lecture is still purely linear, although with recursion available when considered as a programming language. Not every function we want to write is linear, however, so we pursue the adjoint approach to mix linear with nonlinear types. This can be generalized further to a preorder of modes as in Lecture 11. With nonlinear types we can then express *multicast* (one message is sent to multiple recipients) and *shared servers* (a provider has multiple clients). Making communication *asynchronous* is a critical to this generalization. For example, it is difficult to conceptualize what synchronous delivery of a message to multiple clients might mean without at least a notion of (logical) time.

In MPASS we had a natural notion of channel that remained stable throughout communication, with a changing type. In SAX all messages (except unit) contain a continuation channel. Is there still a stable underlying notion of channel? We explore this using *messages sequences* that avoid many instances of allocating fresh continuation channels. While not formalized in this lecture, message sequences allow us implement channels as queues and, in some cases, calculate a precise bound on maximal size of queue [Willsey et al., 2016]. Message sequences in the syntax also allow some programs to be written more compactly.

2 Adding Adjoint Modalities to SAX

We recall the syntax of of SAX; the typing rules and dynamics can be found at the end of Lecture 14. Even if not formally distinguished, we use x' to denote a

continuation channel.

$$
\begin{array}{llcll}
\text{Messages} & M & ::= & k(x') & (\oplus, \&) \\
 & & \mid & (y, x') & (\otimes, \multimap) \\
 & & \mid & (\,) & (\mathbf{1}) \\
\text{Continuations} & K & ::= & (\ell(x') \Rightarrow P_\ell(x'))_{\ell \in L} & (\oplus, \&) \\
 & & \mid & ((y, x') \Rightarrow P(y, x')) & (\otimes, \multimap) \\
 & & \mid & ((\,) \Rightarrow P) & (\mathbf{1}) \\
\text{Processes} & P & ::= & x \leftarrow P(x) \,;\, Q(x) & \mathsf{cut} \\
 & & \mid & \mathbf{fwd}\ x\ y & \mathsf{id} \\
 & & \mid & \mathbf{send}\ x\ M & \\
 & & \mid & \mathbf{recv}\ x\ K & \\
 & & \mid & \mathbf{call}\ p\ x\ y_1 \ldots y_n &
\end{array}
$$

We see that messages and continuations do double-duty for a pair of dual types. But here is no dual to $\mathbf{1}$—why? Actually, there is one we just haven't used it. See Section 4 for what it would mean.

To generalize to mixed linear/nonlinear logic we introduce a second layer of types and shifts that go between them.

$$
\begin{array}{llcl}
\text{Structural Types} & A_{\mathsf{S}} & ::= & \ldots \mid \uparrow A_{\mathsf{L}} \\
\text{Linear Types} & A_{\mathsf{L}} & ::= & \ldots \mid \downarrow A_{\mathsf{S}}
\end{array}
$$

Based on the symmetries we have seen so far, we might conjecture that they are dual in a way so that we just need a single new form of message and continuation, respectively, for both of these constructs. And that's indeed the case. We write the logical rules in the form of SAX based on their polarity and then assign program terms. How does this work? Recall that in the move from the sequent calculus to its semi-axiomatic form, the invertible rules remain the same and the noninvertible ones are turned into axioms. The up shift is negative, so its right rule stays intact. By our presupposition, Δ consists only of structural propositions. The left rule is turned into an axiom. We use here the *implicit* form without explicit rules for weaakening and contraction, so we allow structural antecedents in the axioms, denoted by Δ_{S}.

$$
\frac{\Delta \vdash A_{\mathsf{L}}}{\Delta \vdash \uparrow A_{\mathsf{L}}}\ \uparrow R
\qquad\qquad
\frac{}{\Delta_{\mathsf{S}}, \uparrow A_{\mathsf{L}} \vdash A_{\mathsf{L}}}\ \uparrow L
$$

The rules suggest a transition from a channel to its continuation channel at a different mode. We write $\langle x' \rangle$ for this form message.

$$
\frac{\Delta \vdash P(x'_{\mathsf{L}}) :: (x'_{\mathsf{L}} : A_{\mathsf{L}})}{\Delta \vdash \mathbf{recv}\ x_{\mathsf{S}}\ (\langle x'_{\mathsf{L}} \rangle \Rightarrow P(x'_{\mathsf{L}})) :: (x_{\mathsf{S}} : \uparrow A_{\mathsf{L}})}\ \uparrow R
$$

$$
\frac{}{\Delta_{\mathsf{S}}, x_{\mathsf{S}} : \uparrow A_{\mathsf{L}} \vdash \mathbf{send}\ x_{\mathsf{S}}\ \langle x'_{\mathsf{L}} \rangle :: (x'_{\mathsf{L}} : A_{\mathsf{L}})}\ \uparrow L
$$

The downshift works out symmetrically. First, the logical rules.

$$\frac{}{\Delta_\mathsf{S}, A_\mathsf{S} \vdash \downarrow A_\mathsf{S}}\ \downarrow R \qquad \frac{\Delta, A_\mathsf{S} \vdash C_r}{\Delta, \downarrow A_\mathsf{S} \vdash C_r}\ \downarrow L$$

In the left rule, the succedent C_r could be linear or structural. Annotating it with processes:

$$\frac{}{\Delta_\mathsf{S}, x'_\mathsf{S} : A_\mathsf{S} \vdash \mathbf{send}\ x_\mathsf{L}\ \langle x'_\mathsf{S}\rangle :: (x_\mathsf{L} :: \downarrow A_\mathsf{S})}\ \downarrow R$$

$$\frac{\Delta, x'_\mathsf{S} : A_\mathsf{S} \vdash Q(x'_\mathsf{S}) :: (z_r : C_r)}{\Delta, x_\mathsf{L} : \downarrow A_\mathsf{S} \vdash \mathbf{recv}\ x_\mathsf{L}\ (\langle x'_\mathsf{S}\rangle \Rightarrow Q(x'_\mathsf{S})) :: (z_r : C_r)}\ \downarrow L$$

The cut rule may introduce either a linear or a structural channel and structural channels may be shared between the two branches. Since we have just two modes, L and S with $\mathsf{S} > \mathsf{L}$, there are three versions of cut and only two versions of the identity.

$$\frac{(\Delta \geq m \geq r) \quad \Delta_\mathsf{S}, \Delta \vdash A_m \quad \Delta_\mathsf{S}, \Delta', A_m \vdash C_r}{\Delta_\mathsf{S}, \Delta, \Delta' \vdash C_r}\ \mathsf{cut} \qquad \frac{}{\Delta_\mathsf{S}, A_m \vdash A_m}\ \mathsf{id}$$

We retain a nice symmetry and language for messages, continuations, and channels. However, the dynamics becomes more complicated because some messages along shared channels should be persistent, and shared services may also need to be persistent. You can find the rules in a recent paper [Pfenning and Pruiksma, 2023][1]. We will come back to the mixed linear/nonlinear programs in the next lecture when we talk about *futures*.

3 An Example: mapreduce

As an example of a mixed linear/nonlinear program we write mapreduce on *linear* trees with data at the leaves.

$$\mathsf{tree}_A = \oplus\{\mathsf{node} : \mathsf{tree}_A \otimes \mathsf{tree}_A, \mathsf{leaf} : A\}$$

$$\mathit{mapreduce}_{AB}\ (r : B)\ (f_\mathsf{S} : \uparrow(B \otimes B \multimap B))\ (h_\mathsf{S} : \uparrow(A \multimap B))\ (t : \mathsf{tree}_A) = \ldots$$

Here, f_S and h_S are variables of structural type because f is used at every node and h is used at every leaf. We omit the annotation of the linear variables. The type parameters A and B themselves are *linear* so we did not write $f_\mathsf{S} : B \times B \to B$ but There is a corresponding version where A and B are structural types (which would require changing the type for trees). We start by receiving from t.

[1] Available at https://www.cs.cmu.edu/~fp/papers/coordination23.pdf

$$\mathsf{tree}_A = \oplus\{\mathsf{node} : \mathsf{tree}_A \otimes \mathsf{tree}_A, \mathsf{leaf} : A\}$$

$$\begin{array}{l} \mathit{mapreduce}_{AB}\ (y : B)\ (f_\mathsf{s} : \uparrow(B \otimes B \multimap B))\ (h_\mathsf{s} : \uparrow(A \multimap B))\ (t : \mathsf{tree}_A) = \\ \quad \mathbf{recv}\ t\ (\ \mathsf{node}(l, r) \Rightarrow \ldots \\ \qquad\qquad \mid \mathsf{leaf}(x) \Rightarrow \ldots\ ) \end{array}$$

In the case of a node we need to make two (hopefully parallel!) recursive calls. In order to type them it is important that f_s and h_s can be shared. And, yes, these two calls proceed independently, each given a fresh destination y_i.

$$\mathsf{tree}_A = \oplus\{\mathsf{node} : \mathsf{tree}_A \otimes \mathsf{tree}_A, \mathsf{leaf} : A\}$$

$$\begin{array}{l} \mathit{mapreduce}_{AB}\ (y : B)\ (f_\mathsf{s} : \uparrow(B \otimes B \multimap B))\ (h_\mathsf{s} : \uparrow(A \multimap B))\ (t : \mathsf{tree}_A) = \\ \quad \mathbf{recv}\ t\ (\ \mathsf{node}(l, r) \Rightarrow y_1 \leftarrow \mathbf{call}\ \mathit{mapreduce}_{AB}\ y_1\ f_\mathsf{s}\ h_\mathsf{s}\ l\ ; \\ \qquad\qquad\qquad\qquad\ y_2 \leftarrow \mathbf{call}\ \mathit{mapreduce}_{AB}\ y_2\ f_\mathsf{s}\ h_\mathsf{s}\ r\ ; \\ \qquad\qquad\qquad\qquad\ \ldots \\ \qquad\qquad \mid \mathsf{leaf}(x) \Rightarrow \ldots\ ) \end{array}$$

Next, we'd like to call f on y_1 and y_2 but before we do we need to "unwrap" f_s to obtain the underlying linear function f_L. For this purpose we need a send action because $\uparrow(B \otimes B \multimap B)$ is a negative type. The process providing f_s is waiting first for a linear continuation channel f_L and then a pair of following that.

$$\mathsf{tree}_A = \oplus\{\mathsf{node} : \mathsf{tree}_A \otimes \mathsf{tree}_A, \mathsf{leaf} : A\}$$

$$\begin{array}{l} \mathit{mapreduce}_{AB}\ (y : B)\ (f_\mathsf{s} : \uparrow(B \otimes B \multimap B))\ (h_\mathsf{s} : \uparrow(A \multimap B))\ (t : \mathsf{tree}_A) = \\ \quad \mathbf{recv}\ t\ (\ \mathsf{node}(l, r) \Rightarrow y_1 \leftarrow \mathbf{call}\ \mathit{mapreduce}_{AB}\ y_1\ f_\mathsf{s}\ h_\mathsf{s}\ l\ ; \\ \qquad\qquad\qquad\qquad\ y_2 \leftarrow \mathbf{call}\ \mathit{mapreduce}_{AB}\ y_2\ f_\mathsf{s}\ h_\mathsf{s}\ r\ ; \\ \qquad\qquad\qquad\qquad\ p : B \otimes B \leftarrow \mathbf{send}\ p\ (y_1, y_2)\ ; \\ \qquad\qquad\qquad\qquad\ f_\mathsf{L} \leftarrow \mathbf{send}\ f_\mathsf{s}\ \langle f_\mathsf{L} \rangle\ ; \\ \qquad\qquad\qquad\qquad\ \ldots \\ \qquad\qquad \mid \mathsf{leaf}(x) \Rightarrow \ldots\ ) \end{array}$$

Now we can send the pair p to f_L, but we also need to pass it a destination. But that's just the overall output channel y.

$$\mathsf{tree}_A = \oplus\{\mathsf{node} : \mathsf{tree}_A \otimes \mathsf{tree}_A, \mathsf{leaf} : A\}$$

$$\begin{array}{l} \mathit{mapreduce}_{AB}\ (y : B)\ (f_\mathsf{s} : \uparrow(B \otimes B \multimap B))\ (h_\mathsf{s} : \uparrow(A \multimap B))\ (t : \mathsf{tree}_A) = \\ \quad \mathbf{recv}\ t\ (\ \mathsf{node}(l, r) \Rightarrow y_1 \leftarrow \mathbf{call}\ \mathit{mapreduce}_{AB}\ y_1\ f_\mathsf{s}\ h_\mathsf{s}\ l\ ; \\ \qquad\qquad\qquad\qquad\ y_2 \leftarrow \mathbf{call}\ \mathit{mapreduce}_{AB}\ y_2\ f_\mathsf{s}\ h_\mathsf{s}\ r\ ; \\ \qquad\qquad\qquad\qquad\ p : B \otimes B \leftarrow \mathbf{send}\ p\ (y_1, y_2)\ ; \\ \qquad\qquad\qquad\qquad\ f_\mathsf{L} \leftarrow \mathbf{send}\ f_\mathsf{s}\ \langle f_\mathsf{L} \rangle\ ; \\ \qquad\qquad\qquad\qquad\ \mathbf{send}\ f_\mathsf{L}\ (p, y) \\ \qquad\qquad \mid \mathsf{leaf}(x) \Rightarrow \ldots\ ) \end{array}$$

The case of a leaf is simpler: we just unwrap the function h_s and apply it to the data of type A.

$$\mathsf{tree}_A = \oplus\{\mathsf{node} : \mathsf{tree}_A \otimes \mathsf{tree}_A, \mathsf{leaf} : A\}$$

$$\begin{array}{l}
\mathit{mapreduce}_{AB}\ (y : B)\ (f_{\mathsf{s}} : \uparrow(B \otimes B \multimap B))\ (h_{\mathsf{s}} : \uparrow(A \multimap B))\ (t : \mathsf{tree}_A) = \\
\quad \mathbf{recv}\ t\ (\ \mathsf{node}(l, r) \Rightarrow\ y_1 \leftarrow \mathbf{call}\ \mathit{mapreduce}_{AB}\ y_1\ f_{\mathsf{s}}\ h_{\mathsf{s}}\ l\ ; \\
\qquad\qquad\qquad\qquad\ \ y_2 \leftarrow \mathbf{call}\ \mathit{mapreduce}_{AB}\ y_2\ f_{\mathsf{s}}\ h_{\mathsf{s}}\ r\ ; \\
\qquad\qquad\qquad\qquad\ \ p : B \otimes B \leftarrow \mathbf{send}\ p\ (y_1, y_2)\ ; \\
\qquad\qquad\qquad\qquad\ \ f_{\mathsf{L}} \leftarrow \mathbf{send}\ f_{\mathsf{s}}\ \langle f_{\mathsf{L}} \rangle\ ; \\
\qquad\qquad\qquad\qquad\ \ \mathbf{send}\ f_{\mathsf{L}}\ (p, y) \\
\qquad\qquad\ \mid \mathsf{leaf}(x) \Rightarrow\quad h_{\mathsf{L}} \leftarrow \mathbf{send}\ h_{\mathsf{s}}\ \langle h_{\mathsf{L}} \rangle\ ; \\
\qquad\qquad\qquad\qquad\ \ \mathbf{send}\ h_{\mathsf{L}}\ (x, y)\ )
\end{array}$$

This process has significant parallelism beyond just the two recursive calls. The process f receives the results from the recursive calls and a destination and it can run in parallel with the recursive calls! This is a difference between fork/join parallelism and futures (to be explored in the next lecture). In fork/join we'd have to synchronize when the pair p is formed; here synchronization occurs when f needs to receive from its argument channels.

4 Bottom

What is dual to $\mathbf{1}$? Presumably, since $\mathbf{1}$ is positive, it would be negative. Let's recall the rules for the unit.

$$\frac{}{\cdot \vdash \mathbf{1}}\ \mathbf{1}R \qquad \frac{\Delta \vdash C}{\Delta, \mathbf{1} \vdash C}\ \mathbf{1}L$$

If we flip sides, we see that the *succedent* needs to be empty for the right rule.

$$\frac{\Delta \vdash \cdot}{\Delta \vdash \bot}\ \bot R \qquad \frac{}{\bot \vdash \cdot}\ \bot L$$

We know how to check that these are correct: cut reduction and identity expansion (locally), and cut and identity elimination globally. Locally, everything is fine.

$$\frac{\dfrac{\mathcal{D}'}{\Delta \vdash \cdot}\ \bot R \qquad \dfrac{}{\bot \vdash \cdot}\ \bot L}{\Delta \vdash \cdot}\ \mathsf{cut}_\bot \quad \longrightarrow_R \quad \frac{\mathcal{D}'}{\Delta \vdash \cdot}$$

$$\frac{}{\bot \vdash \bot}\ \mathsf{id}_\bot \quad \longrightarrow_E \quad \frac{\dfrac{}{\bot \vdash \cdot}\ \bot L}{\bot \vdash \bot}\ \bot R$$

As expected, the process assignment doesn't require anything new. We observe that the left rule is just renamed into an axiom, but doesn't change.

$$\frac{\Delta \vdash P :: \cdot}{\Delta \vdash \mathbf{recv}\ x\ (() \Rightarrow P) :: (x : \bot)}\ \bot R \qquad \frac{}{x : \bot \vdash \mathbf{send}\ x\ () :: \cdot}\ \bot X$$

The way we have biased the intuitionistic judgments, a process $\Delta \vdash P :: \cdot$ computes for its own sake, without a client. And it could not be closed $\cdot \vdash P :: \cdot$ is not provable (except perhaps by abusing recursion in some way) so it doesn't fit well into our applications.

5 Message Sequences

As mentioned in the introduction, when we moved from synchronous to asynchronous communication, we needed to introduce continuation channels. Creating fresh channels for every message is a good model for the theory, but not plausible for an implementation. Could we introduce *message sequences* that appear on the same channel as a way not only to make the communication model more realistic, but also write more compact programs?

Intuitively, a message sequence just replaces the continuation channel with another message. Conversely, when receiving along a channel we no longer match against a single message at a time, but a whole message sequence.

Message Sequence	$\overline{M}$	::=	$k(\overline{M})$	$(\oplus, \&)$
		$\mid$	$(y, \overline{M})$	$(\otimes, \multimap)$
		$\mid$	$()$	$(\mathbf{1})$
		$\mid$	x'	cont. channel
Continuations	$\overline{K}$	::=	$(\overline{M} \Rightarrow P \mid \overline{K}) \mid \cdot$	
Processes	P	::=	$x \leftarrow P(x)\ ;\ Q(x)$	cut
		$\mid$	$\mathbf{fwd}\ x\ y$	id
		$\mid$	$\mathbf{send}\ x\ \overline{M}$	
		$\mid$	$\mathbf{recv}\ x\ \overline{K}$	
		$\mid$	$\mathbf{call}\ p\ x\ y_1 \ldots y_n$	

Before we formalize that statics and dynamics of this extended language, we consider two examples to see where the formal development should lead us. We begin with `append`, which has a relatively simple use of pattern matching.

```
type bin = +{'b0 : bin, 'b1 : bin, 'e : 1}
type list = +{'cons : bin * list, 'nil : 1}

proc append (R : list) (L : list) (K : list) =
  recv L ( 'cons(x,L') => R' <- call append R' L' K ;
                          send R 'cons(x,R')
         | 'nil() => fwd R K )
```

This might expand to

```
proc append (R : list) (L : list) (K : list) =
  recv L ( 'cons(p) => recv p ((x,L') =>
              R' <- call append R' L' K ;
              p' : bin * list <- send p' (x,R') ;
              send R 'cons(p'))
         | 'nil(u) => recv u (() => fwd R K) )
```

The second is process to compute $\lfloor \frac{x}{2} \rfloor$ for x in unary form.

```
type nat = +{'zero : 1, 'succ : nat}

proc half (r : nat) (x : nat) =
  recv x ( 'zero() => send r 'zero()
         | 'succ('zero()) => send r 'zero()
         | 'succ('succ(y)) => h <- call half h y ;
                              send r 'succ(h) )
```

This might expand to

```
proc half (r : nat) (x : nat) =
recv x ( 'zero(x') =>
           recv x' (() => u : 1 <- send u () ;
                          send r 'zero(u))
       | 'succ(x') =>
           recv x' ( 'zero(x'') =>
                       recv x'' (() => u : 1 <- send u () ;
                       send r 'zero(u))
                   | 'succ(y) => h <- call half' h y ;
                                 send r 'succ(h) ) )
```

We now have to update the statics and dynamics for this enriched language in a way that is consistent with SAX. As we often do, we start with the statics. Message sequences seem more manageable than the more general form of pattern matching, so we start with them. There are two classes of rules, one for positive types that send to a client and one for negative types that sends to a provider. In the premise, we have to check that the message sequence fits the type of the channel, but the original channel is no longer needed.

$$\frac{\Delta \vdash \overline{M} : \lfloor A \rfloor}{\Delta \vdash \mathbf{send}\ x\ \overline{M} :: (x : A)}\ \mathsf{send}^+$$

Now we have rules for each of the positive types with the corresponding messages.

$$\frac{\Delta \vdash \overline{M} : \lfloor A_k \rfloor}{\Delta \vdash k(\overline{M}) : \lfloor \oplus\{\ell : A_\ell\}_{\ell \in L} \rfloor}\ \oplus R$$

$$\frac{\Delta \vdash \overline{M} : \lfloor B \rfloor}{\Delta, y : A \vdash (y, \overline{M}) : \lfloor A \otimes B \rfloor}\ \otimes R \qquad \frac{}{\cdot \vdash (\,) : \lfloor \mathbf{1} \rfloor}\ \mathbf{1}R$$

When we encounter an actual continuation channel rather than a message, we use an instance of the identity.

$$\frac{}{x' : A \vdash x' : \lfloor A \rfloor}\ \mathsf{id}_R$$

Do these rules look familiar? They should! Think about it before you read on.

These are *almost* the rules for *right focus* except that we can apply the identity to finish the focusing phase for any proposition A, not just for atoms and negative propositions. Similar, the premise antecedent $y : A$ in $\otimes R$ arises from the identity on A, rather than focusing on $\lfloor A \rfloor$ on the right.

These differences reflects differences between *proof construction*, where we would like to chain together inferences as much as possible to minimize nondeterminism, and *proof reduction* where we would like the freedom to write sequences as long or as short as we would like to. We therefore call this *partial focusing* which is also reflected in the notation $\lfloor A \rfloor$. An interesting property of partial focusing is that the right rules that had become axioms have turned back into right rules!

To complete this thought, message sequences of negative type correspond to *partial left focus*! We use a new notation here, writing δ for a singleton succedent $z : C$.

$$\frac{\Delta, \lfloor A \rfloor \vdash \overline{M} :: \delta}{\Delta, x : A \vdash \mathbf{send}\ x\ \overline{M} :: \delta}\ \mathsf{send}_L$$

$$\frac{\Delta, \lfloor A_k \rfloor \vdash \overline{M} :: \delta}{\Delta, \lfloor \&\{\ell : A_\ell\}_{\ell \in L} \rfloor \vdash k(\overline{M}) :: \delta}\ \&L \qquad \frac{\Delta, \lfloor B \rfloor \vdash \overline{M} :: \delta}{\Delta, y : A, \lfloor A \multimap B \rfloor \vdash (y, \overline{M}) :: \delta}\ \multimap L$$

$$\frac{}{\lfloor A \rfloor \vdash x' :: (x' : A)}\ \mathsf{id}_L$$

6 Pattern Matching

Along with sequences of messages, we also changed continuations so they can receive and discriminate whole message sequences. This looks more complicated than message sequences themselves since patterns can be nested and appear in different orders and to different depths. To allow this we define the operation of *projection* that filters out cases from a complex pattern match.

We might conjecture that since message sequences correspond to (partial) focusing that pattern matching will correspond to (partial) inversion. That's not far-fetched since the corresponding logical connectives are indeed invertible!

We start on the right.

$$\frac{\Delta\ ;\ A \vdash \overline{K} :: \delta}{\Delta, x : A \vdash \mathbf{recv}\ x\ \overline{K} :: \delta}\ \mathsf{recv}_L$$

The judgment form $\Delta\ ;\ A \vdash \overline{K} :: \delta$ is inspired by the notation for inversion on the left, $\Delta\ ;\ \Omega \vdash C$. In the case of message sequences, the ordered inversion context Ω will always be a singleton.

We construct the rules such that $\overline{K}$ in the judgment $\Delta \;;\; A \vdash \overline{K} :: \delta$ and later $\Delta \vdash \overline{K} : A$ cannot be empty. This rules out the case where the there is no branch for a (well-typed) message received.

We start with conjunction this time. When the antecedent is $A \otimes B$ then we need to receive a channel of type A and then B has to be matched against the remaining continuations.

$$\frac{\Delta, y : A \;;\; B \vdash \overline{K} \mathbin{@} (y, \_) :: \delta}{\Delta \;;\; A \otimes B \vdash \overline{K} :: \delta} \otimes L$$

The projection is only defined if all patterns are pairs, and rule can only be applied if the projection $\overline{K} \mathbin{@} (y, \_)$ is nonempty. Projection also instantiates the variable bound in the patterns with y so that all branches in $\overline{K} \mathbin{@} (y, \_)$ use the same variable. The fact that $y : A$ is added to the antecedents and not to the ordered context is a departure from the usual inversion, but important to enforce matching against message sequences (not trees).

$$\begin{array}{lcll} (z, \overline{M}) \Rightarrow P(z) \mid \overline{K} & @ & (y, \_) & = \quad \overline{M} \Rightarrow P(y) \mid \overline{K} \mathbin{@} (y, \_) \\ (z, \overline{M}) \Rightarrow P(z) & @ & (y, \_) & = \quad \overline{M} \Rightarrow P(y) \\ \overline{K} & @ & (y, \_) & \text{undefined otherwise} \end{array}$$

Here we have abbreviated $\overline{M} \Rightarrow P \mid \cdot$ as $\overline{M} \Rightarrow P$. Since we would like the language to remain deterministic, at the unit type there must only be a single branch and we revert back to the ordinary typing judgment for processes.

$$\frac{\Delta \vdash \overline{K} \mathbin{@} (\,) :: \delta}{\Delta \;;\; \mathbf{1} \vdash \overline{K} :: \delta} \mathbf{1} L$$

$$\begin{array}{lcll} ((\,) \Rightarrow P) & @ & (\,) & = \quad P \\ \overline{K} & @ & (\,) & \text{undefined otherwise} \end{array}$$

Finally we come to external choice. The patterns must all start with a label, so we project onto each label of the external choice.

$$\frac{\Delta \;;\; A_\ell \vdash \overline{K} \mathbin{@} \ell(\_) :: \delta \quad (\forall \ell \in L)}{\Delta \;;\; \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \overline{K} :: \delta} \oplus L$$

If $\overline{K} \mathbin{@} \ell(\_)$ is empty then this means the label ℓ is not accounted for among the patterns even though it should be. In this case we won't be able to complete the typing derivation because the other rules $\oplus L$, $\mathbf{1} L$, and $\mathsf{cont}/\mathsf{var}^+$ (see below) all require the continuation to have at least one branch. Furthermore, we enforce that all branches start with a label in L. This latter condition is not strictly necessary for progress and preservation but retains the connection to the logical inference rules.

$$\begin{array}{lcll} (\ell(\overline{M}) \Rightarrow P \mid \overline{K}) & @ & \ell(\_) & = \quad \overline{M} \Rightarrow P \mid (\overline{K} \mathbin{@} \ell(\_)) \\ (k(\overline{M}) \Rightarrow P \mid \overline{K}) & @ & \ell(\_) & = \quad \overline{K} \mathbin{@} \ell(\_) \qquad \text{for } k \neq \ell \text{ and } k \in L \\ (\cdot) & @ & \ell(\_) & = \quad \cdot \\ \overline{K} & @ & \ell(\_) & \text{undefined otherwise} \end{array}$$

The last case arises when the pattern consists of a single branch with a single variable. We just revert to the usual typing judgment.

$$\frac{\Delta, x' : A \vdash P(x') :: \delta}{\Delta \,;\, A \vdash (x' \Rightarrow P(x')) :: \delta}\ \mathsf{cont/var}^{+}$$

This rule also marks a difference to full inversion: A does not need to be a negative type.

We show the remaining rules for right inversion on negative types without further discussion since we have seen all the necessary ideas already.

$$\frac{\Delta \vdash \overline{K} : A}{\Delta \vdash \mathbf{recv}\ x\ \overline{K} :: (x : A)}\ \mathbf{recv}_R$$

$$\frac{\Delta, y : A \vdash \overline{K}\ @\ (y, \_) : B}{\Delta \vdash \overline{K} : A \multimap B} \multimap R \qquad \frac{\Delta \vdash \overline{K}\ @\ \ell(\_) : A_\ell \quad (\forall \ell \in L)}{\Delta \vdash \overline{K} : \&\{\ell : A_\ell\}_{\ell \in L}}\ \&R$$

$$\frac{\Delta \vdash P(x') :: (x' : A)}{\Delta \vdash (x' \Rightarrow P(x')) : A}\ \mathsf{cont/var}^{-}$$

7 Dynamics for Message Sequences

We could give a dynamics for message sequences and general pattern matching directly on the extended syntax. We pursue here a different approach where the dynamics is defined by translation into the SAX core language. This translation has to create fresh channels for the middle of message sequences, and has to break up complex patterns into a nested matches of simple patterns.

The translations are type-directed, so we translate $\mathbf{send}\ x\ \overline{M}$ with metalevel function $\mathbf{send}^*\ (x : A)\ \overline{M} = P$ where P uses only simple messages. Similarly, a $\mathbf{recv}\ x\ \overline{K}$ is translated by $\mathbf{recv}^*\ (x : A)\ \overline{K} = P$. We keep in mind the following properties (becoming theorems) where the conclusion is typed in the original SAX system.

1. If $\Delta \vdash \mathbf{send}\ x\ \overline{M} :: (x : A)$ then $\Delta \vdash (\mathbf{send}^*\ (x : A)\ \overline{M}) :: (x : A)$
2. If $\Delta, x : A \vdash \mathbf{send}\ x\ \overline{M} :: \delta$ then $\Delta \vdash (\mathbf{send}^*\ (x : A)\ \overline{M}) :: \delta$
3. If $\Delta, x : A \vdash \mathbf{recv}\ x\ \overline{K} :: \delta$ then $\Delta, x : A \vdash (\mathbf{recv}^*\ x\ \overline{K}) :: \delta$
4. If $\Delta \vdash \mathbf{recv}\ x\ \overline{K} :: (x : A)$ then $\Delta \vdash (\mathbf{recv}^*\ x\ \overline{K}) :: (x : A)$

We have to generalize these properties to talk about partial focusing, which we leave as an exercise

$$
\begin{array}{lcl}
\mathbf{send}^*\ (x : \oplus\{\ell : A_\ell\})\ k(\overline{M}) & = & x' \leftarrow \mathbf{send}^*\ (x' : A_k)\ \overline{M}\ ;\ \mathbf{send}\ x\ k(x') \\
\mathbf{send}^*\ (x : A \otimes B)\ (y, \overline{M}) & = & x' \leftarrow \mathbf{send}^*\ x'_B\ \overline{M}\ ;\ \mathbf{send}\ x\ (y, x') \\
\mathbf{send}^*\ (x : \mathbf{1})\ (\,) & = & \mathbf{send}\ x\ (\,) \\
\mathbf{send}^*\ (x : A)\ x' & = & \mathbf{fwd}\ x\ x' \\
\\
\mathbf{send}^*\ (x : \&\{\ell : A_\ell\})\ k(\overline{M}) & = & x' \leftarrow \mathbf{send}\ x\ k(x')\ ;\ \mathbf{send}^*\ (x' : A_k)\ \overline{M} \\
\mathbf{send}^*\ (x : A \multimap B)\ (y, \overline{M}) & = & x' \leftarrow \mathbf{send}\ x\ (y, x')\ ;\ \mathbf{send}^*\ (x' : B)\ \overline{M} \\
\mathbf{send}^*\ (x : A)\ x' & = & \mathbf{fwd}\ x'\ x
\end{array}
$$

$$
\begin{array}{lcl}
\mathbf{recv}^*\ (x : \oplus\{\ell : A_\ell\}_{\ell \in L})\ \overline{K} & = & \mathbf{recv}\ x\ (\ell(x') \Rightarrow \mathbf{recv}^*\ (x : A_\ell)\ (\overline{K}\ @\ \ell(\_)))_{\ell \in L} \\
\mathbf{recv}^*\ (x : A \otimes B)\ \overline{K} & = & \mathbf{recv}\ x\ ((y, x') \Rightarrow \mathbf{recv}^*\ (x : B)\ (\overline{K}\ @\ (y, \_))) \\
\mathbf{recv}^*\ (x : \mathbf{1})\ \overline{K} & = & \mathbf{recv}\ x\ ((\,) \Rightarrow \overline{K}\ @\ (\,)) \\
\mathbf{recv}^*\ (x : A)\ (x' \Rightarrow P(x')) & = & P(x) \\
\\
\mathbf{recv}^*\ (x : \&\{\ell : A_\ell\}_{\ell \in L})\ \overline{K} & = & \mathbf{recv}\ x\ (\ell(x') \Rightarrow \mathbf{recv}^*\ (x : A_\ell)\ (\overline{K}\ @\ \ell(\_)))_{\ell \in L} \\
\mathbf{recv}^*\ (x : A \multimap B)\ \overline{K} & = & \mathbf{recv}\ x\ ((y, x') \Rightarrow \mathbf{recv}^*\ (x : B)\ (\overline{K}\ @\ (y, \_))) \\
\mathbf{recv}^*\ (x : A)\ (x' \Rightarrow P(x')) & = & P(x)
\end{array}
$$

References

Frank Pfenning and Klaas Pruiksma. Relating message passing and shared memory, proof-theoretically. In S. Jongmans and A. Lopes, editors, *25th International Conference on Coordination Models and Languages (COORDINATION 2023)*, pages 3–27, Lisbon, Portugal, June 2023. Springer LNCS 13908. Notes to an invited talk.

Max Willsey, Rokhini Prabhu, and Frank Pfenning. Design and implementation of Concurrent C0. In *Fourth International Workshop on Linearity*, pages 73–82. EPTCS 238, June 2016.

Lecture Notes on Futures

15-836: Substructural Logics
Frank Pfenning

Lecture 16
November 2, 2023

1 Introduction

In many ways the border between message passing and shared memory concurrency is fluid. We can think of a message passing language as implemented using shared memory, or shared memory representing messages passed between threads. So far, we have taken the message passing view of communication, we will now take the shared memory view.

Shared memory comes in several forms. We strive to find the right level of abstraction to retain the close connection to logic and also illuminate the correspondence to message passing. It turns out that *futures* [Halstead, 1985] are the perfect fit. They were first developed for Lisp, a dynamically typed language, but are entirely compatible with static typing [Pruiksma and Pfenning, 2022, Somayyajula and Pfenning, 2022, 2023].

What are futures? Consider the construct

$$\textbf{let } x = \textbf{future } e_1 \textbf{ in } e_2(x)$$

in a functional language. The idea is the **future** e_1 immediately returns a *promise* p. Then we evaluate e_1 and $e_2(p)$ in parallel. If evaluation of $e_2(p)$ requires the value of p it blocks until the evaluation of e_1 has fulfilled the promise by providing a value and $e_2(p)$ can continue.

The analogy with message passing should be clear: a promise acts as a channel of communication between $e_1$and e_2. We think of the future as being a designated shared memory location where the value of the promise can eventually be found. This point of view has several advantages. For one, it is quite close to an implementation. For another, it quite naturally lends itself to a *sequential* implementation which is less apparent under message passing. Finally, it allows us to investigate, formally, the connection to message passing [Pfenning and Pruiksma, 2023].

While futures aren't intrinsically substructural (and certainly weren't conceived as such), it turns out that a substructural version has been proposed [Blelloch and Reid-Miller, 1999] and can have advantages in asymptotic complexity over nonlinear ones. Our development in this lecture starts with the linear version and then generalizes it by adding structural types.

This form of shared memory of *write-once* shared memory: once written, it can be read by multiple consumers but it can not be modified. Allowing this would require an imperative language with mutable shared memory. Such languages (or libraries in imperative host languages) certainly exist (including, for example, Halstead's original Multilisp) and programs in them are subject to reasoning via external means. For example, we may want to reason about programs in Rust using concurrent separation logic [Brookes, 2007, O'Hearn, 2007, Jung et al., 2018], a substructural logic in a different mold from the ones we have been discussing. In this case the programming language and logic are not related by a proofs-as-programs correspondence.

2 Reinterpreting SAX: Positive Types

The fundamental idea is that in a sequent each variable stands for a memory address. A process P reads from the addresses among the antecedents and writes to the address labeling the succedent.

$$\underbrace{x_1 : A_1, \ldots, x_n : A_n}_{\text{read}} \vdash P :: \underbrace{(x : A)}_{\text{write}}$$

If everything is linear the process P should definitely read from all the x_i and write to x. However, in the presence of recursion the mere type system does not guarantee that and we need some additional reasoning [Somayyajula and Pfenning, 2022].

Under message passing, the type A described the type of message exchanged. Here, it describes the contents of the memory cell. What was a continuation channel now becomes an address of further data. Cut now allocates a new shared cell, while the identity moves the contents of one cell to another. We first focus on posi-

tive types.

$$
\begin{array}{llcll}
\text{Values} & V & ::= & k(x) & (\oplus) \\
 & & \mid & (x_1, x_2) & (\otimes) \\
 & & \mid & (\,) & (\mathbf{1}) \\
\text{Continuations} & K & ::= & (\ell(x) \Rightarrow P_\ell(x))_{\ell \in L} & (\oplus) \\
 & & \mid & ((x_1, x_2) \Rightarrow P(x_1, x_2)) & (\otimes) \\
 & & \mid & ((\,) \Rightarrow P) & (\mathbf{1}) \\
\text{Processes} & P & ::= & x \leftarrow P(x)\ ;\ Q(x) & \mathsf{cut} \\
 & & \mid & \mathbf{move}\ x\ y & \mathsf{id} \\
 & & \mid & \mathbf{write}\ x\ V & \\
 & & \mid & \mathbf{read}\ x\ K & \\
 & & \mid & \mathbf{call}\ p\ x\ y_1 \ldots y_n &
\end{array}
$$

At runtime, we think of tagged value such as $k(a)$ as a pair consisting of a tag k and an address a, a value (a_1, a_2) as a pair of addresses a_1 and a_2, and $(\,)$ as a unit value. Continuations branch based on a value read from memory.

We have replaced **send** and **recv** with **read** and **write**. Also, instead of forwarding between channels we move the contents of one memory location to another.

Both statics and dynamics for the positive types are straightforward.

$$\frac{k \in L}{y : A_k \vdash \mathbf{write}\ x\ k(y) :: (x : \oplus\{\ell : A_\ell\}_{\ell \in L})}\ \oplus X$$

$$\frac{\Delta, y : A_\ell \vdash Q_\ell(y) \quad (\forall \ell \in L)}{\Delta, x : \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \mathbf{read}\ k\ (\ell(y) \Rightarrow Q_\ell(y))_{\ell \in L} :: (z : C)}\ \oplus L$$

$$\frac{}{x_1 : A, x_2 : B \vdash \mathbf{write}\ x\ (x_1, x_2) :: (x : A \otimes B)}\ \otimes X$$

$$\frac{\Delta, x_1 : A, x_2 : B \vdash Q(x_1, x_2) :: \delta}{\Delta, x_1 : A \otimes B \vdash \mathbf{recv}\ x\ ((x_1, x_2) \Rightarrow Q(x_1, x_2)) :: \delta}\ \otimes L$$

$$\frac{}{\cdot \vdash \mathbf{write}\ x\ (\,) :: (x : \mathbf{1})}\ \mathbf{1}X \qquad \frac{\Delta \vdash Q :: \delta}{\Delta, x : \mathbf{1} \vdash \mathbf{read}\ x\ ((\,) \Rightarrow Q) :: \delta}\ \mathbf{1}L$$

Cut and identity do not change from the sequent calculus.

$$\frac{}{y : A \vdash \mathbf{move}\ x\ y :: (x : A)}\ \mathsf{id} \qquad \frac{\Delta \vdash P(x) :: (x : A) \quad \Delta', x : A \vdash Q(x) :: \delta}{\Delta, \Delta' \vdash x_A \leftarrow P(x)\ ;\ Q(x) :: \delta}\ \mathsf{cut}$$

The dynamics relies on the $V \triangleright K$ operation carried over from the message passing setting. However, we differentiate memory cells at address a containing value V,

written $\mathsf{cell}(a, V)$, from processes. This will give us properties such as: a configuration is *final* if it contains only memory cells and no processes.

$$
\begin{array}{lcll}
\mathsf{proc}(x \leftarrow P(x)\,;\,Q(x)) & \longrightarrow & \mathsf{proc}(P(a)), \mathsf{proc}(Q(a)) & (a \text{ fresh}) \\
\mathsf{cell}(b, V), \mathsf{proc}(\mathbf{move}\ a\ b) & \longrightarrow & \mathsf{cell}(a, V) & \\
\mathsf{proc}(\mathbf{write}\ a\ V) & \longrightarrow & \mathsf{cell}(a, V) & \\
\mathsf{cell}(a, V), \mathsf{proc}(\mathbf{read}\ a\ K) & \longrightarrow & \mathsf{proc}(V \triangleright K) & \\
\mathsf{proc}(\mathbf{call}\ p\ a\ b_1 \ldots b_n) & \longrightarrow & \mathsf{proc}(P(a, b_1, \ldots, b_n)) & \\
 & & \text{for } p\ x\ y_1 \ldots y_n = P(x, y_1, \ldots, y_n) \in \Sigma &
\end{array}
$$

$$
\begin{array}{lcll}
k(a) \triangleright (\ell(x) \Rightarrow P_\ell(x))_{\ell \in L} & = & P_k(a) & (k \in L) \\
(a_1, a_2) \triangleright ((x_1, x_2) \Rightarrow P(x_1, x_2)) & = & P(a_1, a_2) & \\
(\,) \triangleright ((\,) \Rightarrow P) & = & P &
\end{array}
$$

Here is a simple program we can write already, reversing a list.

```
type bin = +{'b0 : bin, 'b1 : bin, 'e : 1}
type list = +{'cons : bin * list, 'nil : 1}

proc rev (R : list) (L : list) (K : list) =
   read L ( 'cons(p) => read p ((x,L') =>
               p' : bin * list <- write p' (x,K) ;
               K' : list <- write K' 'cons(p') ;
               call R L' K')
          | 'nil(u) => read u (() =>
               move R K) )

proc reverse (R : list) (L : list) =
   u : 1 <- write u () ;
   K : list <- write K 'nil() ;
   call rev R L K
```

Using the equivalent of message sequences, this could be more compact—something we'll get back to in the next lecture.

3 Reinterpreting SAX: Negative Types

So far, things worked out as one might expect: on positive types, receives become reads and sends become writes. Negative types present a surprise because *every action on the succedent is a write!* This means that cells no longer just contain small values V, but they also have to contain continuations. We will shortly write this out. But first the rules: right rules write, left rules (even in the form of axioms)

read.

$$\frac{\Delta \vdash P_\ell(y) :: (y : A_\ell) \quad (\forall \ell \in L)}{\Delta \vdash \mathbf{write}\ x\ (\ell(y) \Rightarrow P_\ell(y)) :: (x : \&\{\ell : A_\ell\}_{\ell \in L})}\ \&R$$

$$\frac{k \in L}{x : \&\{\ell : A_\ell\}_{\ell \in L} \vdash \mathsf{read}\ x\ k(y) :: (y : A_k)}\ \&X$$

$$\frac{\Delta, x_1 : A \vdash P(x_1, x_2) :: (x_2 : B)}{\Delta \vdash \mathbf{write}\ x\ ((x_1, x_2) \Rightarrow P(x_1, x_2)) :: (x : A \multimap B)}\ \multimap R$$

$$\frac{}{x_1 : A, x : A \multimap B \vdash \mathbf{read}\ x\ (x_1, x_2) :: (x_2 : B)}\ \multimap X$$

Let's take a closer look at the meaning of linear functions. **write** a $((x_1, x_2) \Rightarrow P(x_1, x_2))$ will write the continuation $(x_1, x_2) \Rightarrow P(x_1, x_2)$ to the cell at address a.

Conversely, **read** a (a_1, a_2) will read the continuation and pass it a_1 and a_2, where $a_1 : A$ is the "actual argument" of the function and $a_2 : B$ is the destination for the result.

Our syntax is now:

Values	V	$::=$	$k(x)$	$(\oplus, \&)$
		$\mid$	(x_1, x_2)	$(\otimes, \multimap)$
		$\mid$	$()$	$(\mathbf{1})$
Continuations	K	$::=$	$(\ell(x) \Rightarrow P_\ell(x))_{\ell \in L}$	$(\oplus, \&)$
		$\mid$	$((x_1, x_2) \Rightarrow P(x_1, x_2))$	$(\otimes, \multimap)$
		$\mid$	$(() \Rightarrow P)$	$(\mathbf{1})$
Storable	S	$::=$	$V \mid K$	
Processes	P	$::=$	$x \leftarrow P(x)\ ;\ Q(x)$	cut
		$\mid$	**move** $x\ y$	id
		$\mid$	**write** $x\ S$	
		$\mid$	**read** $x\ S$	
		$\mid$	**call** $p\ x\ y_1 \ldots y_n$	

The dynamics also changes subtly from purely positive types. We add the following two, while the remaining ones remain the same.

$$\begin{array}{lcl} \mathsf{proc}(\mathbf{write}\ a\ K) & \longrightarrow & \mathsf{cell}(a, K) \\ \mathsf{cell}(a, K), \mathsf{proc}(\mathbf{read}\ a\ V) & \longrightarrow & \mathsf{proc}(V \rhd K) \end{array}$$

We use map as iteration as an example. First, the message passing version.

```
type bin = +{'b0 : bin, 'b1 : bin, 'e : 1}
type list = +{'cons : bin * list, 'nil : 1}

type iter = &{'next : bin -o bin * iter, 'done : 1}
```

```
proc map (R : list) (i : iterm) (L : list) =
  recv L ( 'cons(p) => recv p ((x,L') =>
              f : bin -o bin * iter <- send i 'next(f) ;
              p : bin * iter <- send f (x, p) ;
              recv p ((y, i') => R' <- call map R' i' L' ;
                                 q : bin * list <- send q (y, R') ;
                                 send R 'cons(q)))
         | 'nil(u) => recv u (() =>
              v : 1 <- send i 'done(v)
              send R 'nil(v)) )
```

To convert this to a program using futures, positive send/receive become read-/write, respectively, while the this correspondence is switched for negative types.

```
    read L ( 'cons(p) => read p ((x,L') =>
              f : bin -o bin * iter <- read i 'next(f) ;
              p : bin * iter <- read f (x, p) ;
              read p ((y, i') => R' <- call map R' i' L' ;
                                 q : bin * list <- write q (y, R') ;
                                 write R 'cons(q)))
         | 'nil(u) => read u (() =>
              v : 1 <- read i 'done(v)
              write R 'nil(v)) )
```

4 Mixed Linear/Structural Futures

We have our recipe: We combine linear and structural types by adding appropriate shifts. The upshift is intrinsically negative, while the downshift is intrinsically positive. We have already assigned a syntax to the processes for these shifts that

we reuse.

Values	V	$::=$	$k(x)$	$(\oplus, \&)$
		$\mid$	(x_1, x_2)	$(\otimes, \multimap)$
		$\mid$	$()$	$(\mathbf{1})$
		$\mid$	$\langle x \rangle$	$(\downarrow, \uparrow)$
Continuations	K	$::=$	$(\ell(x) \Rightarrow P_\ell(x))_{\ell \in L}$	$(\oplus, \&)$
		$\mid$	$((x_1, x_2) \Rightarrow P(x_1, x_2))$	$(\otimes, \multimap)$
		$\mid$	$(() \Rightarrow P)$	$(\mathbf{1})$
		$\mid$	$(\langle x \rangle \Rightarrow P(x))$	$(\downarrow, \uparrow)$
Storable	S	$::=$	$V \mid K$	
Processes	P	$::=$	$x \leftarrow P(x)\ ;\ Q(x)$	cut
		$\mid$	$\mathbf{move}\ x\ y$	id
		$\mid$	$\mathbf{write}\ x\ S$	
		$\mid$	$\mathbf{read}\ x\ S$	
		$\mid$	$\mathbf{call}\ p\ x\ y_1 \ldots y_n$	

In the typing rules we just have to replace send and receive by write and read, as appropriate.

$$\frac{}{\Delta_\mathsf{S}, y_\mathsf{S} : A_\mathsf{S} \vdash \mathbf{write}\ x_\mathsf{L}\ \langle y_\mathsf{S} \rangle :: (x_\mathsf{L} :: \downarrow A_\mathsf{S})}\ \downarrow R$$

$$\frac{\Delta, y_\mathsf{S} : A_\mathsf{S} \vdash Q(y_\mathsf{S}) :: \delta}{\Delta, x_\mathsf{L} : \downarrow A_\mathsf{S} \vdash \mathbf{read}\ x_\mathsf{L}\ (\langle y_\mathsf{S} \rangle \Rightarrow Q(y_\mathsf{S})) :: \delta}\ \downarrow L$$

$$\frac{\Delta \vdash P(y_\mathsf{L}) :: (y_\mathsf{L} : A_\mathsf{L})}{\Delta \vdash \mathbf{write}\ x_\mathsf{S}\ (\langle y_\mathsf{L} \rangle \Rightarrow P(y_\mathsf{L})) :: (x_\mathsf{S} : \uparrow A_\mathsf{L})}\ \uparrow R$$

$$\frac{}{\Delta_\mathsf{S}, x_\mathsf{S} : \uparrow A_\mathsf{L} \vdash \mathbf{read}\ x_\mathsf{S}\ \langle y_\mathsf{L} \rangle :: (y_\mathsf{L} : A_\mathsf{L})}\ \uparrow L$$

In the dynamics, the changes are a little less straightforward. For addresses of structural type we need to create *persistent cells* in the dynamics. We write $!\mathsf{cell}(a_\mathsf{S}, S)$ for a persistent cell. This means when it is read it remains in configuration rather than being consumed. The rules before remain what they are, assuming all the addresses are linear. In addition we have:

$$\begin{array}{lcl}
\mathsf{proc}(\mathbf{write}\ a_\mathsf{S}\ S) & \longrightarrow & !\mathsf{cell}(a_\mathsf{S}, S) \\
!\mathsf{cell}(a_\mathsf{S}, S), \mathsf{proc}(\mathbf{read}\ a_\mathsf{S}\ S') & \longrightarrow & \mathsf{proc}(S \bowtie S') \\
!\mathsf{cell}(b_\mathsf{S}, S), \mathsf{proc}(\mathbf{move}\ a_\mathsf{S}\ b_\mathsf{S}) & \longrightarrow & !\mathsf{cell}(a_\mathsf{S}, S)
\end{array}$$

Here $S \bowtie S'$ is defined by $K \bowtie V = V \bowtie K = V \rhd K$, accounting for both positive and negative types.

As an example, consider a map over a linear list with shared binary numbers. We write $A[m]$ for a type of mode m and $(S)A_s$ for $\downarrow A_s$, signifying that the scope is shared. The code uses some compound values, analogous to message sequences. We will return to them in the next lecture.

```
type bin[m] = +{'b0 : bin[m], 'b1 : bin[m], 'e : 1}
type bin_s = bin[S]
type list = +{'cons : (S)bin_s * list, 'nil : 1} % linear

proc map (R : list) (F : bin_s -> bin_s) (L : list) =
  read L ( 'cons(<x>,L') => y : bin_s <- read F (x, y) ;
                            R' <- call map R' F L' ;
                            write R 'cons(<y>,R')
         | 'nil() => write R 'nil() )
```

It is of course possible to give other modes to map.

References

G. E. Blelloch and M. Reid-Miller. Pipeling with futures. *Theory of Computing Systems*, 32:213–239, 1999.

Stephen Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 365(1–3):227–270, 2007.

Robert H. Halstead. Multilisp: A language for parallel symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–539, October 1985.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation of higher-order concurrent separation logic. *Journal of Functional Programming*, 29:e20, November 2018.

Peter O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.

Frank Pfenning and Klaas Pruiksma. Relating message passing and shared memory, proof-theoretically. In S. Jongmans and A. Lopes, editors, *25th International Conference on Coordination Models and Languages (COORDINATION 2023)*, pages 3–27, Lisbon, Portugal, June 2023. Springer LNCS 13908. Notes to an invited talk.

Klaas Pruiksma and Frank Pfenning. Back to futures. *Journal of Functional Programming*, 32:e6, 2022.

Siva Somayyajula and Frank Pfenning. Type-based termination for futures. In *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*, pages 12:1–12:21, Haifa, Israel, August 2022. LIPIcs 228.

Siva Somayyajula and Frank Pfenning. Dependent type refinements for futures. In M. Kerjean and P. Levy, editors, *39th International Conference on Mathematical Foundations of Programming Semantics (MFPS 2023)*, Bloomington, Indiana, USA, June 2023. Preliminary version.

Lecture Notes on Data Layout

15-836: Substructural Logics
Frank Pfenning

Lecture 17
November 9, 2023

1 Introduction

Data layout is a critical component in the efficient compilation of functional languages (see, for example, [Morrisett, 1995, Weeks, 2006, Vollmer et al., 2017, 2019]). Yet, in implementations of functional languages data layout decisions are left to the compiler rather than being available to the programmer. In today's lecture we design a type system in which certain high-level data layout decisions are explicit in the types, while lower-level details are still left to a compiler.

The surprising property of the type system is that it corresponds directly to a fragment of adjoint logic in its semi-axiomatic formulation which we call SNAX. In other words, SNAX provides a logical explanation for issues of data layout! Petersen et al. [2003] was an early attempt at characterizing data layout using an *ordered* type system. While this worked as far as it went, it did not generalize further. The root cause seems to be that the proofs-as-programs interpretation for ordered logic does not capture *adjacency* because the general rule of cut must apply to a proposition in the middle of ordered antecedents.

The line of research on SAX provided a surprising twist. SAX itself does not satisfy traditional cut elimination, as explained in a prior lecture. Certain cuts may be allowed if the cut formula arises from a use of a new axiom and is therefore a subformula of the goal sequent. We call these cuts *snips* and prove a new version of cut elimination [DeYoung et al., 2020] in which snips are allowed to remain. But we didn't tackle the question what the *computational* meaning of snips might be. It turns out that while a *cut* allocates a new memory cell, a *snip* merely computes an address relative to an existing address.

The fundamental connection between semi-axiomatic proofs and data layout is developed for nonlinear futures by DeYoung and Pfenning [2022]. Since this is largely consistent with the approach and notation of this course, we will not repeat

repeat it in these notes but provide a link to the extended version of the paper paper.[1]

These notes then will cover only the connection between *partial focusing* and data layout in the SNAX source language under the shared memory interpretation, which was discovered (during this course) in analogy to message sequences.

2 Data Layout: Compound Values

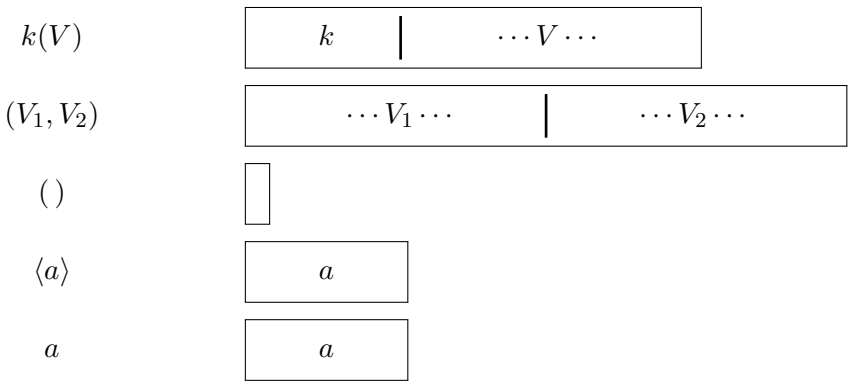
Message sequences were defined to model asynchronous communication along buffered channels. We just consider the positive types, because we won't be specific about how negative types are laid out (they are not directly observable, after all).

$$
\begin{array}{llcll}
\text{Messages Sequences} & \overline{M} & ::= & k(\overline{M}) & (\oplus) \\
& & \mid & (y, \overline{M}) & (\otimes) \\
& & \mid & (\,) & (\mathbf{1}) \\
& & \mid & \langle x' \rangle & (\downarrow) \\
& & \mid & x' & \text{cont. channel}
\end{array}
$$

When we think about memory layout, we do not need the first component of a pair to be an address—it could just be another value. The continuation channel x' is replaced by an address x, but in the typing rules to come later we will restrict such address to be of negative type. The reason is that we would like to statically allocate the space for a value. We just write V and K instead of $\overline{V}$ and $\overline{K}$ since the restricted case is just a special case.

$$
\begin{array}{llcll}
\text{Values} & V & ::= & k(V) & (\oplus) \\
& & \mid & (V_1, V_2) & (\otimes) \\
& & \mid & (\,) & (\mathbf{1}) \\
& & \mid & \langle x \rangle & (\downarrow) \\
& & \mid & x & (\multimap, \&, \uparrow)
\end{array}
$$

We picture the layout as follows:



[1] https://arxiv.org/abs/2212.06321v3.pdf

We imagine that the unit doesn't actually take any space, but we still display it as a narrow box.

Let's look at two recursive types:

$$\mathsf{nat} = \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \mathsf{nat}\}$$
$$\mathsf{list} = \oplus\{\mathsf{nil} : \mathbf{1}, \mathsf{cons} : \mathsf{nat} \otimes \mathsf{list}\}$$

Because these types are recursive and purely positive, their layout would be unbounded in size. This is the same problem as posed by (possibly mutually) recursive structs in C. In C, as here, the solution is to require an indirection via a pointer/address, which has a fixed size representation.

The indirection can be either through a downshift $\downarrow A$ or through a negative type A^-. For natural numbers, there are two obvious options:

$$\mathsf{nat} = \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \downarrow\mathsf{nat}\} \qquad \% \text{ eager}$$
$$\mathsf{nat} = \oplus\{\mathsf{zero} : \mathbf{1}, \mathsf{succ} : \uparrow\mathsf{nat}\} \qquad \% \text{ lazy}$$

The first would be the ordinary (eager) natural numbers, observable in their entirety. The second would be the *lazy natural numbers* because the successor a number would be $\mathsf{succ}\ \langle a \rangle$ where at the address is a continuation that can compute the tail.

But something doesn't seem right, because shifts in mixed linear/nonlinear logic go between structural and linear types. We are saved by the generality of adjoint logic, where $\downarrow^{\ell}_{m} A$ only requires that $\ell \geq m$. If we are working just with linear natural numbers, the downshift would be $\downarrow^{\mathsf{L}}_{\mathsf{L}}\mathsf{nat}$. For structural natural numbers it would probably $\downarrow^{\mathsf{S}}_{\mathsf{S}}\mathsf{nat}$. Since for the moment we are just working in purely linear logic, we just write $\downarrow A$ for $\downarrow^{\mathsf{L}}_{\mathsf{L}} A$.

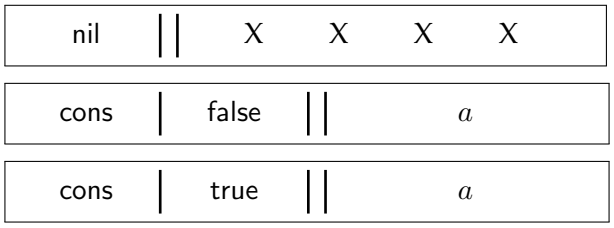
The ordinary eager lists might have pointers to natural numbers.

$$\mathsf{list} = \oplus\{\mathsf{nil} : \mathbf{1}, \mathsf{cons} : \downarrow\mathsf{nat} \otimes \downarrow\mathsf{list}\}$$

The representation might be more compact for lists of booleans by "inlining" them instead of having a pointer to a Boolean.

$$\mathsf{bool} = \oplus\{\mathsf{false} : \mathbf{1}, \mathsf{true} : \mathbf{1}\}$$
$$\mathsf{listbool} = \oplus\{\mathsf{nil} : \mathbf{1}, \mathsf{cons} : \mathsf{bool} \otimes \downarrow\mathsf{listbool}\}$$

When a pointer to the element is embedded in a list it is called a *boxed representation*; otherwise it is said to be *unboxed*. Data of the unboxed type listbool might be layed out as on of the following, where a is the address for the tail of the list.



The extra unused space for nil is there because all values of type listbool should be laid out with the same width.

3 Partial Focusing Revisited

As can be seen from the development above, values still arise from partial focusing but with slightly different criteria for partiality. We begin with the rules for writing with positive types.

$$\frac{\Delta \vdash V : \lceil A \rceil}{\Delta \vdash \mathbf{write}\ x\ V :: (x : A)}\ \mathsf{write}$$

Now we have rules for each of the positive types with the corresponding values.

$$\frac{\Delta \vdash V : \lceil A_k \rceil}{\Delta \vdash k(V) : \lceil \oplus\{\ell : A_\ell\}_{\ell \in L} \rceil}\ \oplus R$$

$$\frac{\Delta_1 \vdash V_1 : \lceil A \rceil \quad \Delta \vdash V_2 : \lceil B \rceil}{\Delta_1, \Delta_2 \vdash (V_1, V_2) : \lceil A \otimes B \rceil}\ \otimes R \qquad \frac{}{\cdot \vdash (\,) : \lceil \mathbf{1} \rceil}\ \mathbf{1}R$$

When we encounter a downshift or a negative type we end the partial focusing phase, either with the corresponding axiom or an identity.

$$\frac{}{x : A \vdash \langle x \rangle : \lceil \downarrow A \rceil}\ \mathsf{down}X \qquad \frac{}{x : A^- \vdash x : \lceil A^- \rceil}\ \mathsf{id}^-$$

Pattern matching works symmetrically. The pattern has to be deep enough to cover all well-typed values of a given type. Inversion now has to continue on both sides of a pair, so we need to generalize to allow the patterns to be nested.

$$\begin{array}{lrcl} \text{Pattern Sequence} & \overline{V} & ::= & V \cdot \overline{V} \mid (\cdot) \\ \text{Continuations} & K & ::= & (\overline{V} \Rightarrow P \mid K) \mid \cdot \end{array}$$

The sequence of nested patterns match the ordered context in $\Delta\ ;\ \Omega \vdash K :: \delta$. The judgment is started with Ω being a singleton.

$$\frac{\Delta\ ;\ \lceil A \rceil \vdash K :: \delta}{\Delta, x : A \vdash \mathbf{read}\ x\ K :: \delta}\ \mathsf{read}$$

$$\frac{\Delta\ ;\ A\ B\ \Omega \vdash K\ @\ (\_,\_) :: \delta}{\Delta\ ;\ (A \otimes B)\ \Omega \vdash K :: \delta}\ \otimes L \qquad \frac{\Delta\ ;\ \Omega \vdash K\ @\ (\,) :: \delta}{\Delta\ ;\ \mathbf{1}\ \Omega \vdash K :: \delta}\ \mathbf{1}L$$

$$\frac{\Delta\ ;\ A_\ell\ \Omega \vdash \overline{K}\ @\ \ell(\_) :: \delta \quad (\forall \ell \in L)}{\Delta\ ;\ \oplus\{\ell : A_\ell\}_{\ell \in L}\ \Omega \vdash K :: \delta}\ \oplus L$$

$$\frac{\Delta, x : A\ ;\ \Omega \vdash \overline{K}\ @\ \langle x \rangle :: \delta}{\Delta\ ;\ (\downarrow A)\ \Omega \vdash K :: \delta}\ \downarrow L \qquad \frac{\Delta, x : A^-\ ;\ \Omega \vdash \overline{K}\ @\ x :: \delta}{\Delta\ ;\ A^-\ \Omega \vdash K :: \delta}$$

$$\frac{\Delta \vdash P :: \delta}{\Delta\ ;\ \cdot \vdash (\cdot) \Rightarrow P :: \delta}$$

In the definition below we don't explicate failure conditions (for example, if there no branches for a given tag, or if there is a mismatch between the projection p and the pattern).

$$\begin{array}{lcll}
((V_1, V_2) \cdot \overline{V} \Rightarrow P \mid K) & @ & (_,_) & = & (V_1 \cdot V_2 \cdot \overline{V} \Rightarrow P) \mid (K\ @\ (_,_)) \\
((\,) \cdot \overline{V} \Rightarrow P \mid K) & @ & (\,) & = & (\overline{V} \Rightarrow P) \mid (K\ @\ (\,)) \\
(\ell(V) \cdot \overline{V} \Rightarrow P \mid K) & @ & \ell(_) & = & (V \cdot \overline{V} \Rightarrow P) \mid (K\ @\ \ell(_)) \\
(k(V) \cdot \overline{V} \Rightarrow P \mid K) & @ & \ell(_) & = & K\ @\ \ell(_) \qquad \text{for } k \neq \ell \text{ and } k \in L \\
(\langle x \rangle \cdot \overline{V} \Rightarrow P(x) \mid K) & @ & \langle y \rangle & = & (\overline{V} \Rightarrow P(y)) \mid (K\ @\ \langle y \rangle) \\
(x \cdot \overline{V} \Rightarrow P(x) \mid K) & @ & y & = & (\overline{V} \Rightarrow P(y)) \mid (K\ @\ y) \\
(\cdot) & @ & p & = & (\cdot)
\end{array}$$

4 Example: Append with Three Types

We give three different examples, one for appending two lists of pointers to natural numbers, and one for appending lists of (unboxed) booleans.

In the first example we pass memory contents directly instead of pointers.

```
type nat = +{'zero : 1, 'succ : <down> nat}
type list = +{'nil : 1, 'cons : <down> nat * <down> list}

proc append (R : list) (L : list) (K : list) =
  read L ( 'nil() => move R K
         | 'cons(<x>, <L'>) => R' : list <- call append R' L' K
                               write R 'cons(<x>, <R'>) )
```

In the next version, we pass pointers instead of the layout structures. Because in this version we have to match all the way until we encounter an address, there is a slight awkwardness in the recursive calls: we might prefer not to decompose and recompose the tail of the list.

```
type nat = +{'zero : 1, 'succ : nat_ptr}
type nat_ptr = <down> nat

type list = +{'nil : 1, 'cons : <down> nat * list_ptr}
type list_ptr = <down> list

proc append (R : list_ptr) (L : list_ptr) (K : list_ptr) =
  read L ( <'nil()> => move R K
         | <'cons(<x>,<L'>)> => R' : list_ptr <- call append R' <L'> K
                                write R 'cons(<x>, R') )
```

In the unboxed example we see that partial matching could be quite helpful, because without that we need to match the entirety of the boolean instead of being able to leave it as a variable.

```
type bool = +{'false : 1, 'true : 1}
type boollist = +{'nil : 1, 'cons : bool * <down>boollist}

proc append (R : boollist) (L : boollist) (K : boollist) =
  read L ( 'nil() => move R K
         | 'cons('false(),<L'>) => R' : list <- call append R' L' K
                                   write R 'cons('false(), <R'>)
         | 'cons('true(),<L'>) => R' : list <- call append R' L' K
                                  write R 'cons('true(), <R'>) )
```

It would be easy to accommodate partial matches by removing the restriction on variables in values to be of negative type.

References

Henry DeYoung and Frank Pfenning. Data layout from a type-theoretic perspective. In *38th Conference on the Mathematical Foundations of Programming Semantics (MFPS 2022)*. Electronic Notes in Theoretical Informatics and Computer Science 1, 2022. URL https://arxiv.org/abs/2212.06321v6. Invited paper. Extended version available at https://arxiv.org/abs/2212.06321v3.pdf.

Henry DeYoung, Frank Pfenning, and Klaas Pruiksma. Semi-axiomatic sequent calculus. In Z. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, pages 29:1–29:22, Paris, France, June 2020. LIPIcs 167.

Greg Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, December 1995. Available as Technical Report CMU-CS-95-226.

Leaf Petersen, Robert Harper, Karl Crary, and Frank Pfenning. A type theory for memory allocation and data layout. In G. Morrisett, editor, *Conference Record of the 30th Annual Symposium on Principles of Programming Languages (POPL'03)*,

pages 172–184, New Orleans, Louisiana, January 2003. ACM Press. Extended version available as Technical Report CMU-CS-02-171, December 2002.

Michael Vollmer, Sarah Spall, Buddhika Chamith, Laith Sakka, Chaitanya Koparkar, Milind Kulkarni, Sam Tobin-Hochstadt, and Ryan R. Newton. Compiling tree transformas to operate on packed representations. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, pages 26:1–26:29, Barcelona, Spain, June 2017. LIPIcs 745.

Michael Vollmer, Chaitanya Koparkar, Mike Rainey, Laith Sakka, and Milind Kulkarni. LoCal: A language for programs operating in serialized data. In Kathryn McKinley and Kathleen Fisher, editors, *40th Conference on Programming Language Design and Implementation (PLDI 2019)*, pages 48–62, Phoenix, Arizona, June 2019. ACM.

Stephen Weeks. Whole-program compilation in MLton. In Andrew Kennedy and Franọis Pottier, editors, *Proceedings of the Workshop on ML*, Portland, Oregon, September 2006. ACM. Slides available at http://www.mlton.org/References.attachments/060916-mlton.pdf.

Lecture Notes on The Inverse Method

15-836: Substructural Logics
Frank Pfenning

Lecture 18
November 15, 2023

1 Introduction

In this lecture we return to an early theme, namely forward inference. We also switch gears from the proofs-as-programs interpretation of substructural logic in terms of message passing and shared memory to general theorem proving.

Why would we want to prove theorems in substructural logics? First, we have already seen that forward inference (which is a particular form of theorem proving) can be used to model various algorithmic problems, like parsing, subtyping, planning, or graph algorithms. Second, there are a logics for reasoning about programs, specifically separation logic [Reynolds, 2002] and concurrent separation logic [O'Hearn, 2007, Brookes, 2007], both of which substructural. Proving correctness of imperative programs in such logics ultimately comes down to theorem proving in substructural logic. Third, the problem of program synthesis that has recently garnered much attention can be simplified if we know, for example, that the programs we want to synthesize have substructural types because it drastically reduces the search space [Hughes and Orchard, 2020, Melo e Sousa, 2021]. Fourth, a structured form of proof search is the basis for substructural logic programming [Hodas and Miller, 1994, López et al., 2005] that allows yet another class of algorithms to be expressed at a high level of abstraction.

In a way there is an "obvious" method to do theorem proving: we use the rules of the cut-free sequent calculus for bottom-up proof construction. Once theorems become even somewhat complex, this is no longer feasible because there are too many choices and therefore too much backtracking. We can use inversion to reduce the number of choices, but there remains much nondeterminism. Chaining together rules with focusing [Andreoli, 1992, 2001] makes this even better, but proof search continues to suffer from the difficulty of learning from failure on some branches while searching others. As far as I am aware, clause learning for SAT has

not yet been understood at a sufficiently fundamental level to effectively apply to nonclassical and substructural logics.

An alternative approach is to use Maslov's *inverse method* [Maslov, 1964]. It is called "inverse" because it proceeds from identity sequents towards the goal instead of from the goal towards identity sequents. At first this might seem a crazy idea because the universe of theorems we generate is infinite and poorly structured, but as we will see it works! The inverse method is quite beautiful because it applies essentially to any logic that admits a cut-free sequent calculus, with particular logic-specific considerations in each case [Voronkov, 1992, Degtyarev and Voronkov, 2001]. Other techniques such as resolution are tied specifically to classical logic, so they don't seem as useful for substructural logics. Applications of the inverse method to substructural logic have also been devised [Chaudhuri and Pfenning, 2005a,b, Chaudhuri, 2006] and generalize the material in these notes further.

2 The Basic Idea

Let's look at the example

$$A \multimap (B \mathbin{\&} C) \vdash (A \multimap B) \mathbin{\&} (A \multimap C)$$

which we should be able to prove. In today's lecture, we use A, B, C, etc. to stand for atomic proposition rather than P, Q, R. It seems clear (more later) that the possible identities at the leaves of a proof tree for this sequent should be

$$\frac{}{A \vdash A}\ \mathsf{id}_A \qquad \frac{}{B \vdash B}\ \mathsf{id}_B \qquad \frac{}{C \vdash C}\ \mathsf{id}_C$$

The space of possible forward inferences here seems *huge*! For example, we might deduce

$$\frac{\dfrac{}{A \vdash A}\ \mathsf{id}_A}{\vdash A \multimap A}\ \multimap R$$

It is easy to see that this will not get us anywhere, keep in mind our overall goals. Why is that? Before you read on, think about this and see if you can extend a tentative answer to a more general idea how to make forward inference plausible.

The reason this inference is useless is because $A \multimap A$ is not a subformula that occurs in our goal sequent. In the cut-free sequent calculus, though, all propositions that occur in a proof are subformulas of the final goal sequent.

The idea then is to *specialize* the inference rules so they can be applied in the forward direction but to infer subformulas of the goal sequent! Before we do this, let's examine the subformula property more precisely. By looking at the goal sequent, we can not only predict which subformulas might occur in a proof, but also on which side of a sequent they will be. Except for implication, all the rules keep formulas on the same side of the sequent. And the antecedent of an implication is always on the opposite side of the sequent from the implication itself.

Let's apply this idea, naming the subformulas according to the side of the sequent they may appear on, using L_i for left and R_j for right subformulas.

$$\underbrace{A^R \multimap \underbrace{(B^L \mathbin{\&} C^L)}_{=L_1}}_{=L_0} \vdash \underbrace{\underbrace{(A^L \multimap B^R)}_{=R_1} \mathbin{\&} \underbrace{(A^L \multimap C^R)}_{=R_2}}_{=R_0}$$

First, the possible identities that might be used. We see that all three atoms, A, B, and C may appear on the left as well as on the right in a sequent, so all three identities are possible.

$$\frac{}{A^L \vdash A^R}\ \mathsf{id}_A \qquad \frac{}{B^L \vdash B^R}\ \mathsf{id}_B \qquad \frac{}{C^L \vdash C^R}\ \mathsf{id}_C$$

Next, consider $R_0 = R_1 \& R_2$. Since this is a right formula, there is only one possible specialized rule to infer R_0.

$$\frac{\Delta \vdash R_1 \quad \Delta \vdash R_2}{\Delta \vdash R_0}\ \&R_0$$

For $R_1 = A^L \multimap B^R$ and $R_2 = A^L \vdash C^R$ we also get just a single rule each, that is, two instances of $\multimap R$.

$$\frac{\Delta, A^L \vdash B^R}{\Delta \vdash R_1}\ \multimap R_1 \qquad \frac{\Delta, A^L \vdash C^R}{\Delta \vdash R_2}\ \multimap R_2$$

For L_0 there is a single instance of the $\multimap L$ rule.

$$\frac{\Delta_1 \vdash A^R \quad \Delta_2, L_1 \vdash \delta}{\Delta_1, \Delta_2, L_0 \vdash \delta}\ \multimap L_0$$

The last remaining subformula is $L_1 = B^L \mathbin{\&} C^L$ has two specialized left rules.

$$\frac{\Delta, B^L \vdash \delta}{\Delta, L_1 \vdash \delta}\ \&L_1^1 \qquad \frac{\Delta, C^L \vdash \delta}{\Delta, L_1 \vdash \delta}\ \&L_1^2$$

$$\frac{}{A \vdash A}\ \mathsf{id}_A \qquad \frac{}{B \vdash B}\ \mathsf{id}_B \qquad \frac{}{C \vdash C}\ \mathsf{id}_C$$

$$\frac{\Delta \vdash R_1 \quad \Delta \vdash R_2}{\Delta \vdash R_0}\ \&R_0 \qquad \frac{\Delta, A \vdash B}{\Delta \vdash R_1}\ \multimap R_1 \qquad \frac{\Delta, A \vdash C}{\Delta \vdash R_2}\ \multimap R_2$$

$$\frac{\Delta_1 \vdash A \quad \Delta_2, L_1 \vdash \delta}{\Delta_1, \Delta_2, L_0 \vdash \delta}\ \multimap L_0 \qquad \frac{\Delta, B \vdash \delta}{\Delta, L_1 \vdash \delta}\ \&L_1^1 \qquad \frac{\Delta, C \vdash \delta}{\Delta, L_1 \vdash \delta}\ \&L_1^2$$

Figure 1: Specialized rules for $A \multimap (B \,\&\, C) \vdash (A \multimap B) \,\&\, (A \multimap C)$, goal sequent $L_0 \vdash R_0$

The rules are summarized in Figure 1. We have dropped the superscripts on the atoms since they are determined by their position. An interesting observation is that there are no longer any logical connectives! So during inference we do not consider any of the usual sequent calculus rules, just these specialized ones. Because of the (side-aware) subformula property, there is a proof of our goal sequent if and only if we can infer $L_0 \vdash R_0$ with these rules.

We proceed in a breadth first fashion, always applying all possible rules considering the "facts" already in our database, where the facts are sequents that can be derived. Note that even though our logic is linear, this inference is a structural inference so we may hope that it saturates. We start with the first round, in which only two rules can be applied.

(1)	$A \vdash A$	(id_A)
(2)	$B \vdash B$	(id_B)
(3)	$C \vdash C$	(id_C)
(4)	$L_1 \vdash B$	$(\&L_1^1\ 1)$
(5)	$L_1 \vdash C$	$(\&L_1^2\ 2)$

Since $L_1 = B \,\&\, C$, we see that sequents (4) and (5) make sense after we expand the definitions. Besides inferences that only give us sequents we already know, the only new ones are two applications of $\multimap L_0$.

(1)	$A \vdash A$	(id_A)
(2)	$B \vdash B$	(id_B)
(3)	$C \vdash C$	(id_C)
(4)	$L_1 \vdash B$	$(\&L_1^1\ 1)$
(5)	$L_1 \vdash C$	$(\&L_1^2\ 2)$
(6)	$A, L_0 \vdash B$	$(\multimap L_0\ 1\ 4)$
(7)	$A, L_0 \vdash C$	$(\multimap L_0\ 1\ 5)$

Now we can apply $\multimap R_1$ and $\multimap R_2$, followed by $\&R_0$.

(1)	$A \vdash A$	(id_A)
(2)	$B \vdash B$	(id_B)
(3)	$C \vdash C$	(id_C)
(4)	$L_1 \vdash B$	$(\& L_1^1\ 1)$
(5)	$L_1 \vdash C$	$(\& L_1^2\ 2)$
(6)	$A, L_0 \vdash B$	$(\multimap L_0\ 1\ 4)$
(7)	$A, L_0 \vdash C$	$(\multimap L_0\ 1\ 5)$
(8)	$L_0 \vdash R_1$	$(\multimap R_1\ 6)$
(9)	$L_0 \vdash R_2$	$(\multimap R_2\ 7)$
(10)	$L_0 \vdash R_0$	$(\& R_0\ 8\ 9)$

We have to be careful about the final inference because both premises must have the same antecedent Δ. Fortunately, that is the case with $\Delta = L_0$.

The sequent (10) is also our goal sequent, but we also have reached a point of saturation: any further inferences would only yield sequents we already have.

Next we do an example that is not provable:

$$A \multimap (B \otimes C) \vdash (A \multimap B) \otimes (A \multimap C)$$

As before, we label subformulas.

$$\underbrace{A^R \multimap \underbrace{(B^L \otimes C^L)}_{=L_1}}_{=L_0} \vdash \underbrace{\underbrace{(A^L \multimap B^R)}_{=R_1} \otimes \underbrace{(A^L \multimap C^R)}_{=R_2}}_{=R_0}$$

Instances of the identity as the same as before.

$$\frac{}{A^L \vdash A^R}\ \mathsf{id}_A \qquad \frac{}{B^L \vdash B^R}\ \mathsf{id}_B \qquad \frac{}{C^L \vdash C^R}\ \mathsf{id}_C$$

For the left propositions we generate:

$$\frac{\Delta_1 \vdash A \quad \Delta_2, L_1 \vdash \delta}{\Delta_1, \Delta_2, L_0 \vdash \delta}\ \multimap L_0 \qquad \frac{\Delta, A, B \vdash \delta}{\Delta, L_1 \vdash \delta}\ \otimes L_1$$

And for the right propositions:

$$\frac{\Delta_1 \vdash R_1 \quad \Delta_2 \vdash R_2}{\Delta_1, \Delta_2 \vdash R_0}\ \otimes R_0 \qquad \frac{\Delta, A \vdash B}{\Delta \vdash R_1}\ \multimap R_1 \quad \frac{\Delta, A \vdash C}{\Delta \vdash R_2}\ \multimap R_2$$

Now we throw away the general rules and start with

(1)	$A \vdash A$	(id_A)
(2)	$B \vdash B$	(id_B)
(3)	$C \vdash C$	(id_C)

At this point we realize that *no rule is applicable*! Therefore we know the goal sequent is not provable.

The same style of rule generations applies to most of the connectives of linear logic ($A \multimap B$, $A \mathbin{\&} B$, $A \otimes B$, $\mathbf{1}$, $A \oplus B$) but has to be modified when we consider the exponential $!A = \downarrow\uparrow A$ or the additive units $\mathbf{0}$ and $\top$. We'll return to them in Section 5.

3 The Inverse Method with Focusing

We can exploit focusing to create fewer rules and take bigger steps. The only sequents that will explicitly appear in a focused inverse method proof are *stable sequents*, which are those where no invertible rule can be applied and proof must proceed by focusing either on the right or left.

Stable antecedents are either negative propositions A^- or suspended positive atoms $\langle P^+ \rangle$. Stable succedents are either positive propositions A^+ or suspended negative atoms $\langle P^- \rangle$. We purposely omit $\mathbf{0}$ and $\top$ for now.

$$\begin{array}{lcll} \text{Negative Propositions} & A^- & ::= & A \multimap B \mid A \mathbin{\&} B \\ \text{Stable Antecedents} & \Delta & ::= & \cdot \mid \Delta, \langle P^+ \rangle \mid \Delta, A^- \\ \text{Positive Propositions} & A^+ & ::= & A \otimes B \mid \mathbf{1} \mid A \oplus B \\ \text{Stable Succedents} & \delta & ::= & \langle P^- \rangle \mid A^+ \end{array}$$

We just use letters Δ and δ for the stable antecedents and succedents, since only those are of interest in this section.

This time, we do not introduce intermediate names ahead of time, but will do so during the rule generation process. Our goal is

$$A \multimap (B \mathbin{\&} C) \vdash (A \multimap B) \mathbin{\&} (A \multimap C)$$

This is not stable, so we have to apply inversion until we have reached one more more stable sequent. For this purpose we have to decide which atoms should be positive and which negative. For simplicity, we make them all positive. You may want to review the rules for focusing from Lecture 12. We reach two stable sequents:

$$A^+ \multimap (B^+ \mathbin{\&} C^+), \langle A^+ \rangle \vdash C^+$$
$$A^+ \multimap (B^+ \mathbin{\&} C^+), \langle B^+ \rangle \vdash C^+$$

We have to prove both of these to verify our goal sequent. Let's take the first one (the second one will be symmetric). We can only focus on $A^+ \multimap (B^+ \mathbin{\&} C^+)$ on the left or on C^+ on the right, since we cannot focus on suspended atom. Let's try the first one, defining $L_0 = A^+ \multimap (B^+ \mathbin{\&} C^+)$. We don't know under which circumstances we might focus on this proposition, but if we do the proof will start

as follows (omitting other antecedents and succedents for now):

$$\dfrac{\dfrac{\vdots \qquad \vdots}{\vdash [A^+] \quad [B^+ \,\&\, C^+] \vdash}}{\dfrac{[A^+ \multimap (B^+ \,\&\, C^+)] \vdash}{L_0 \vdash}}$$

There is only one way to proceed with the first open subgoal: right focus on A^+ only succeeds if $\langle A^+ \rangle$ is among the antecedents. That is, for focusing to succeed, such an antecedent must have been in the conclusion.

$$\dfrac{\dfrac{\dfrac{}{\langle A^+ \rangle \vdash [A^+]}\,\mathsf{id}^+ \qquad \dfrac{\vdots}{[B^+ \,\&\, C^+] \vdash}}{\langle A^+ \rangle, [A^+ \multimap (B^+ \,\&\, C^+)] \vdash}}{\langle A^+ \rangle, L_0 \vdash}$$

In the remaining open subproof we could proceed with focus on B^+ or focus on C^+. As a next step in either of these we lose focus and then have to apply inversion. This inversion will immediately suspect B^+ and C^+, respectively. We show the first version:

$$\dfrac{\dfrac{\dfrac{}{\langle A^+ \rangle \vdash [A^+]}\,\mathsf{id}^+ \qquad \dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\langle B^+ \rangle \vdash}}{\langle B^+ \rangle \,;\, \cdot \vdash}}{\cdot \,;\, B^+ \vdash}}{[B^+] \vdash}}{[B^+ \,\&\, C^+] \vdash}\,\&L_1}{\langle A^+ \rangle, [A^+ \multimap (B^+ \,\&\, C^+)] \vdash}}{\langle A^+ \rangle, L_0 \vdash}$$

The sequent at the top of this rather bureaucratic chain of reasoning is stable. We

can fill in some additional (stable) antecedents and succedents.

$$\dfrac{\dfrac{}{\langle A^+\rangle \vdash [A^+]}\ \mathsf{id}^+ \quad \dfrac{\dfrac{\dfrac{\vdots}{\Delta, \langle B^+\rangle \vdash \delta}}{\Delta, \langle B^+\rangle\ ;\ \cdot \vdash \delta}}{\Delta\ ;\ B^+ \vdash \delta}}{\Delta[B^+] \vdash \delta}}{\Delta, [B^+ \mathbin{\&} C^+] \vdash \delta}\ \&L_1}{\Delta, \langle A^+\rangle, [A^+ \multimap (B^+ \mathbin{\&} C^+)] \vdash \delta}}{\Delta, \langle A^+\rangle, L_0 \vdash \delta}$$

From this, and its symmetric variant with C^+ instad of B^+ we extract two big-step rules between stable sequents.

$$\dfrac{\Delta, \langle B^+\rangle \vdash \delta}{\Delta, \langle A^+\rangle, L_0 \vdash \delta}\ L_0^1 \qquad \dfrac{\Delta, \langle C^+\rangle \vdash \delta}{\Delta, \langle A^+\rangle, L_0 \vdash \delta}\ L_0^2$$

Interestingly, this does not expose any new subformulas we have to focus on since suspended atoms cannot be the subject of focusing. We still have the succedent C^+ in our original goal sequent.

$$\dfrac{\dfrac{\vdots}{\vdash [C^+]}}{\vdash C^+}$$

This can only succeed if C^+ is a suspended antecedent, so we obtain the rule

$$\dfrac{}{\langle C^+\rangle \vdash C^+}\ \mathsf{id}_C$$

Due to focusing we only obtain 3 rules compared to 8 before. We can only perform one step:

$$\dfrac{\begin{array}{llll}(1) & \langle C^+\rangle \vdash C^+ & (\mathsf{id}_C)\end{array}}{\begin{array}{llll}(2) & \langle A^+\rangle, L_0 \vdash C^+ & (L_0^2\ 1)\end{array}}$$

The sequent (2) is already our goal sequent, so we are done in 2 steps (and two more for symmetric conjunct that arose from the initial inversion). Compare this to the 10 sequents we derived before hitting our goal without the benefit of focusing!

As an example of something that cannot be proven we consider the type of the S combinator. This is true only if the logic admits contraction.

$$\vdash (A \multimap (B \multimap C)) \multimap ((A \multimap B) \multimap (A \multimap C)$$

This is not a stable sequent, so deciding once again that all atoms should be positive we get

$$A^+ \multimap (B^+ \multimap C^+), A^+ \multimap B^+, \langle A^+ \rangle \vdash C^+$$

There are three propositions we could focus on, two on the left and one on the right.

$$\dfrac{\dfrac{\vdots}{\langle C^+ \rangle \vdash} \quad \dfrac{\dfrac{\vdots}{\vdash [B^+]} \quad \dfrac{\dfrac{\vdots}{C^+ \vdash}}{[C^+] \vdash}}{[B^+ \multimap C^+] \vdash}}{[A^+ \multimap (B^+ \multimap C^+)] \vdash} \; L_0$$

$$\dfrac{\dfrac{\vdots}{\vdash [A^+]} \quad \dfrac{\dfrac{\vdots}{\vdash [B^+]} \quad \dfrac{\dfrac{\vdots}{\langle C^+ \rangle \vdash}}{C^+ \vdash}}{[B^+ \multimap C^+] \vdash}}{[A^+ \multimap (B^+ \multimap C^+)] \vdash} \; L_0$$

The first two open subgoal can only be closed with identities, as in the last example. After filling in missing antecedents and succedents, we obtain:

$$\dfrac{\Delta, \langle C^+ \rangle \vdash \delta}{\Delta, \langle A^+ \rangle, \langle B^+ \rangle, L_0 \vdash \delta} \; L_0$$

Focusing on the left on $A^+ \multimap B^+$ and on the right on C^+ similarly give us the following two rules:

$$\dfrac{\Delta, \langle B^+ \rangle \vdash \delta}{\Delta, \langle A^+ \rangle, L_1 \vdash \delta} \; L_1 \qquad\qquad \dfrac{}{\langle C^+ \rangle \vdash C^+} \; \mathsf{id}_C^+$$

Our goal sequent is

$$L_0, L_1, \langle A^+ \rangle \vdash C^+$$

Initially, we have only one sequent, and only L_0 applies.

$$\begin{array}{llll} (1) & \langle C^+ \rangle \vdash C^+ & (\mathsf{id}_C) \\ \hline (2) & \langle A^+ \rangle, \langle B^+ \rangle, L_0 \vdash C^+ & (L_0\ 1) \end{array}$$

Now we can apply L_1, where $\Delta = \langle A^+ \rangle, L_0$ and $\delta = C^+$. We get:

$$\begin{array}{llll} (1) & \langle C^+ \rangle \vdash C^+ & (\mathsf{id}_C) \\ \hline (2) & \langle A^+ \rangle, \langle B^+ \rangle, L_0 \vdash C^+ & (L_0\ 1) \\ \hline (3) & \langle A^+ \rangle, L_0, \langle A^+ \rangle, L_1 \vdash C^+ & (L_1\ 2) \end{array}$$

At this point we have reached saturation and *almost* proved our goal sequent. The only problem is that we have two copies of $\langle A^+ \rangle$. If we go back and look at the

original goal we see that this makes sense: if we had two copies of $\langle A^+ \rangle$ it would indeed be provable linearly.

This points out another important observation: if we think about the linear sequent (without the exponential or shifts), as we go up we never duplicate any propositions. The goal sequent has only one left occurrence of A^+, so a sequent with two left occurrences of A^+ as the one labeled (3) could not occur. So we should reject it and (in this example), we reach saturation essentially one step earlier.

The insight here is to obtain an inverse method for a given logic we follow these steps (first without focusing):

1. Obtain the usual backwards sequent calculus without cut, and identity limited to atoms (and prove the admissibility of cut and general identity).
2. Label the left- and right-subformulas of the goal sequent.
3. Derive specialized inference rules for each label, and then discard the general rules.
4. Consider any logic-specific additions or modifications of the specialized rules.
5. Saturate the space of sequents derivable with the specialized rules. Even if the logic is undecidable, we can explore the search space by forward reasoning although it may not saturate.
6. If we find a proof of the goal sequent, we succeed.
7. If we saturate without generating the goal sequent, we fail

This is modified slightly for focusing, because we need to generate (and prove!) a focused version of our logic first. Then we generate "big-step" rules that go from stable sequent to stable sequent. The (non-atomic) propositions in the new stable sequent are then named and according to their sidedness focused on to derive more rules.

4 Strict, Affine, and Structural Logic

Assume we have a logic with contraction, such as strict logic. Then we just add the rule of contraction

$$\frac{\Delta, A, A \vdash C}{\Delta, A \vdash C}\ \mathsf{contract}$$

In this system, because we apply the rules from the premises to the conclusion this actually *is* contraction—usually we use it to achieve duplication of a proposition. In the absence of quantifiers (as in this lecture), we could also just treat antecedents as set and write $\Delta_1 \cup \Delta_2$ instead of Δ_1, Δ_2 whenever they are combined. We would

then never have more than one copy of a contractible proposition in a stable sequent.

If we have weakening, matters are a bit more complicated. For example, we should not have rules such as

$$\frac{}{\Delta, \langle A^+ \rangle \vdash A^+}\ \mathsf{id}^+_A$$

That's because even if we have a finite set of labels, there are still many possibilities for Δ. We don't want to enumerate them. We can think of it this way: in backward reasoning, we postpone weakening all the way to the leaves of the proof tree (id or $\mathbf{1}R$). In forward reasoning, we also postpone weakening, but downwards, towards the root of the proof tree.

So where exactly do we finally need to apply weakening? One situation is where we have derived something that can be weakened to our goal sequent. We capture this with a subsumption relation: $(\Delta \vdash A) \leq (\Delta' \vdash A')$ if $\Delta \subseteq \Delta'$ and $A = A'$. Whenever we apply an inference, we can check three properties:

Forward Subsumption: If inference yields $\Delta' \vdash A'$ and there is a sequent $\Delta \vdash A$ in our database such that $\Delta \vdash A \leq \Delta' \vdash A'$ then we do not add the new sequent. We already know something stronger.

Backward Subsumption: If the inference yields $\Delta \vdash A$ and there is a sequent $\Delta' \vdash A'$ in our database such that $\Delta \vdash A \leq \Delta' \vdash A'$ then we replace the old sequent by the newer (stronger) one.

If inference yields $\Delta \vdash A$ and $(\Delta \vdash A) \leq G$ where G is the goal sequent, we succeed.

This is an example of the general principle of *subsumption* in forward inference. Towards saturation, we don't check facts in the database for *equality*, but a more general *subsumption* criterion for redundancy. What that might be may change from inference system to inference system.

Let's try this with the quintessential property that is true in *affine logic* but not in linear logic (writing $A - B$ for affine implication):

$$\vdash A - (B - A)$$

In the small-step system, we introduce two names

$$\begin{array}{lcl} R_0 & = & A^L - R_1 \\ R_1 & = & B^L - A^R \end{array}$$

We generate the rules below. There is no identity for B because it occurs only as B^L and not B^R.

$$\frac{}{A \vdash A}\ \mathsf{id}_A \qquad \frac{\Delta, A \vdash R_1}{\Delta \vdash R_0}\ -R_0 \qquad \frac{\Delta, B \vdash A}{\Delta \vdash R_1}\ -R_1$$

From $A \vdash A$ we cannot apply a single rule! That's problematics because our proposition actually holds in affine logic. In this example it happens that $A \vdash A$ can be weakened to $\Delta, B \vdash A$ with $\Delta = A$.

We can rectify this by allowing weakening when matching against the premises of rules: B doesn't have to be in the sequent. Allowing this we would derive $A \vdash R_1$ and then $\cdot \vdash R_0$, which is our goal sequent. We could also generate another rule that accounts for B being absent.

$$\frac{\Delta \vdash A}{\Delta \vdash R_1}\ -R'_1$$

This would get awkward however when we move to focusing since there might be too many variants of the rules.

Returning to our example, if we apply inversion we obtain the stable goal sequent

$$\langle A^+ \rangle, \langle B^+ \rangle \vdash A^+$$

We can only focus on A^+ on the right, which gives us

$$\frac{}{\langle A^+ \rangle \vdash A^+}\ \mathsf{id}^+_A$$

and $(\langle A^+ \rangle \vdash A^+) \leq (\langle A^+ \rangle, \langle B^+ \rangle \vdash A^+)$.

Also, if we generate a right rule for external choice $A \mathbin{\&} B$ we can no longer require the two branches to have the same antecedents. We define $\Delta_1 \ \mathsf{max}\ \Delta_2$ for multisets to take the maximum of the multiplicity of each element in the two multisets. Then we have the forward rule

$$\frac{\Delta_1 \vdash A \quad \Delta_2 \vdash B}{\Delta_1 \ \mathsf{max}\ \Delta_2 \vdash A \mathbin{\&} B}\ \mathbin{\&}R$$

When mixing logics, for example, in the adjoint framework, we have to combine the various considerations.

5 $\top$ and $\mathbf{0}$ Revisited

We didn't cover this in lecture, but how would we handle

$$\frac{}{\Delta, \mathbf{0} \vdash \delta}\ \mathbf{0}L$$

in the forward direction? We do not want to enumerate possible antecedents or succedents, we want to leave them open. Then we would have something like

$$\frac{}{\mathbf{0} \vdash^W \cdot}\ \mathbf{0}L \qquad \frac{}{\cdot \vdash^W \top}\ \top R$$

where the W on the sequent indicates that the sequent can be weakenend (in antecedent or succedent). Then we precise sequents and weakenable sequents and we have to carefully define rule application in the mixed case and investigate how this attribute of sequents propagates.

Alternatively, we might be able to introduce *metavariables* D and d to stand for an arbitrary Δ and δ respectively and write these as

$$\frac{}{D, \mathbf{0} \vdash d}\ \mathbf{0}L \qquad \frac{}{D \vdash \top}\ \top R$$

and instantiate them as part of the rule application process.

References

Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):197–347, 1992.

Jean-Marc Andreoli. Focussing and proof construction. *Annals of Pure and Applied Logic*, 107(1–3):131–163, 2001.

Stephen Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 365(1–3):227–270, 2007.

Kaustuv Chaudhuri. *The Focused Inverse Method for Linear Logic*. PhD thesis, Carnegie Mellon University, December 2006. Available as technical report CMU-CS-06-162.

Kaustuv Chaudhuri and Frank Pfenning. A focusing inverse method prover for first-order linear logic. In R.Nieuwenhuis, editor, *Proceedings of the 20th International Conference on Automated Deduction (CADE-20)*, pages 69–83, Tallinn, Estonia, July 2005a. Springer Verlag LNCS 3632.

Kaustuv Chaudhuri and Frank Pfenning. Focusing the inverse method for linear logic. In L.Ong, editor, *Proceedings of the 14th Annual Conference on Computer Science Logic (CSL'05)*, pages 200–215, Oxford, England, August 2005b. Springer Verlag LNCS 3634.

Anatoli Degtyarev and Andrei Voronkov. The inverse method. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 4, pages 181–272. Elsevier Science and MIT Press, 2001.

Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994. A preliminary version appeared in the Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science, pages 32–42, Amsterdam, The Netherlands, July 1991.

Jack Hughes and Dominic Orchard. Resourceful program synthesis from graded linear types. In Maribel Fernández, editor, *30th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2020)*, pages 151–170, Bologna, Italy, September 2020. LNCS 12561.

Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In A.Felty, editor, *Proceedings of the 7th International Symposium on Principles and Practice of Declarative Programming (PPDP'05)*, pages 35–46, Lisbon, Portugal, July 2005. ACM Press.

Sergei Maslov. The inverse method of establishing deducibility in the classical predicate calculus. *Soviet Mathematical Doklady*, 5:1420–1424, 1964.

Maria Inês Melo e Sousa. *Synthesis of Programs from Linear Types*. M.Sc. thesis, University of Porto, 2021.

Peter O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.

John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Symposium on Logic in Computer Science*, pages 55–74, Copenhagen, Denmark, July 2002. IEEE Computer Society.

Andrei Voronkov. Theorem proving in non-standard logics based on the inverse method. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE 1992)*, pages 648–662, Saratoga Springs, New York, 1992. Springer LNAI 607.

Lecture Notes on Resource Semantics

15-836: Substructural Logics
Frank Pfenning

Lecture 19
November 16, 2023

1 Introduction

When researchers use the term "*programming language semantics*" they usually either mean a *dynamic semantics* (also referred to as an *operational semantics*) or a *denotational semantics*, that is, an interpretation into a mathematical domain. Meaning is given to programs either by how they execute or by what they denote in some abstract domain. These are both respectable and valuable notions to study. As an intuitionist, though, I find it difficult myself to follow the domain-theoretic approach since it is mostly carried out in classical mathematics.

In logic, a similar divide exists. Semantics (usually credited to Tarski [1956]) interprets the syntax of logic in classical mathematics. In contrast, proof theory studies the notions of truth and consequence via the structure of proofs. Much (but certainly not all) of proof theory is carried out using constructive means, as exemplified by Gentzen's [1935] proof of cut elimination.

One thing I can do, as an intuitionist, is to interpret one constructive language in another and thereby gain a deeper understanding. For example, in Assignment 5 you are asked to interpret affine logic in linear logic. Previously, we have shown (following Girard [1987]) how to interpret (structural) intuitionistic logic in linear logic.

Can we go the other way? Is there, for example, an interpretation of linear logic in (structural) intuitionistic logic? A first answer to the question is "*No!*". For example, we know via a variety of different proofs that structural intuitionistic logic is decidable. Take, for example, the inverse method from the previous lecture. Because of the presence of contraction, the space of possible sequents we can derive is finite and saturation is assured. On the other hand, linear logic that includes the exponential ($!A$) or the adjoint modalities $\uparrow\downarrow A$) is *undecidable* since it can interpret Minsky machines [Lincoln et al., 1992]. This may be surprising, since linear logic *without* the exponential is decidable. As an example proof, in backward search for

a cut-free sequent proof, each premise of each rule is smaller than the conclusion in a simple multiset ordering based on counting the number of connectives.

Since we cannot interpret an undecidable problem in a decidable one, what is left? We can look for an interpretation into an intuitionistic *predicate calculus*, which is certainly also undecidable. This is indeed illuminating since it clarifies a *resource interpretation* of substructural logics that is also applicable to others such as ordered logic.

We start with a preliminary study by giving a *structural* set of rules in which the use of antecedents as resources is explicit. We then follow it by an explicit translation, following Reed and Pfenning [2010]. Related material is also in Reed's Ph.D. Thesis [Reed, 2009].

2 A Sequent Calculus with Explicit Resources

In this formulation of substructural logics we think of each antecedent as a resource and the succedent as a goal to be achieved using an explicit stated collection of resources. We write

$$A_1[\alpha_1], \ldots, A_n[\alpha_n] \vdash C[p]$$

where p is a combination of the resource variables $\alpha_1, \ldots, \alpha_n$.

For the moment, we focus on linear logic. We write $p * q$ for the combination of the resources denoted by p and q, and ϵ for the absence of resources.

$P[\alpha], Q[\beta] \vdash P \otimes Q[\alpha * \beta]$	holds
$P[\alpha], Q[\beta] \vdash P \otimes Q[\beta * \alpha]$	holds
$P[\alpha], Q[\beta] \vdash P \otimes Q[\alpha]$	does not hold
$P[\alpha], Q[\beta], R[\gamma] \vdash P \otimes Q[\alpha * \beta]$	holds
$P[\alpha], Q[\beta], R[\alpha] \vdash P \otimes Q[\alpha * \beta]$	holds?

We can argue if the last one should be allowed, because α does not stand for a unique resource but for two different ones. In our first system we make a presupposition that resource labelings are unique and the last sequent would not be well-formed. In Section 4 we will generalize our point of view. We will stick to the "no repetition in p" mantra because we are not interested in this lecture to capture that a specific resource may be used a fixed number of times.

Since we are modeling linear logic, resource combination $p * q$ should satisfy the following laws:

Associativity	$(p * q) * r = p * (q * r)$
Unit	$\epsilon * p = p = p * \epsilon$
Commutativity	$p * q = q * p$

We can recognize this as a *commutative monoid* which, not surprisingly, is also the structure of the antecedents in linear logic. Because we have no other equations,

we can also say it is the *free commutative monoid* over the variables denoted by α, β, etc.

Before we give the rules, it is easy to conjecture how we might, for example, capture *ordered logic*: we drop commutativity of the resource combination operator. For affine logic, we would need a partial order to capture that some resources may not need to be used.

Starting on formal rules, the identity is easy with weakening implicit.

$$\frac{}{\Gamma, A[\alpha] \vdash A[\alpha]}\ \mathsf{id}$$

The cut rule is a bit tricky because of the asymmetry between antecedents and succedents. But we are used to that by now. We start by writing the rule *without the resources* on the way to deriving what they should be.

$$\frac{\Gamma \vdash A[\quad] \quad \Gamma, A[\quad] \vdash C[\quad]}{\Gamma \vdash C[\quad]}\ \mathsf{cut}$$

We know that the antecedent $A[\quad]$ should be labeled with a fresh resource variable, and that the proof of C in the second premise should be able to use it.

$$\frac{\Gamma \vdash A[\quad] \quad \Gamma, A[\alpha] \vdash C[\quad * \alpha]}{\Gamma \vdash C[\quad]}\ \mathsf{cut}$$

Next we know that the proof of A in the first premise will use some resources q. These will be required for C as well.

$$\frac{\Gamma \vdash A[q] \quad \Gamma, A[\alpha] \vdash C[\quad * \alpha]}{\Gamma \vdash C[\quad * q]}\ \mathsf{cut}$$

Finally, we see that there may be some resources p besides q that are required in the second premise.

$$\frac{\Gamma \vdash A[q] \quad \Gamma, A[\alpha] \vdash C[p * \alpha]}{\Gamma \vdash C[p * q]}\ \mathsf{cut}^{\alpha}$$

Of course, they can't be used in the first premise. We also annotate the rule with α to remind ourselves that α in the premise must be fresh, that is, not already occur in Γ or $p * q$.

Based on similar considerations we get the following rules for conjunction.

$$\frac{\Gamma \vdash A[p] \quad \Gamma \vdash B[q]}{\Gamma \vdash A \otimes B[p * q]}\ \otimes R \qquad \frac{\Gamma, A \otimes B[\gamma], A[\alpha], B[\beta] \vdash C[p * \alpha * \beta]}{\Gamma, A \otimes B[\gamma] \vdash C[p * \gamma]}\ \otimes L^{\alpha,\beta}$$

Because our logic is structural we retain a copy of $A \otimes B[\gamma]$. However, if there is no repetition in the resources of the succedent, we can no longer use it in the premise—it is carried along but no longer usable.

Besides a theorem to come (we can go back and forth between linear logic and resource logic), we also check identity expansion and cut reduction.

$$\longrightarrow_E \quad \dfrac{\dfrac{\dfrac{}{\Gamma, A \otimes B[\gamma], A[\alpha] \vdash A[\alpha]}\,\mathsf{id}_A \quad \dfrac{}{\Gamma, A \otimes B[\gamma], B[\beta] \vdash B[\beta]}\,\mathsf{id}_B}{\Gamma, A \otimes B[\gamma], A[\alpha], B[\beta] \vdash A \otimes B[\alpha * \beta]} \otimes R}{\Gamma, A \otimes B[\gamma] \vdash A \otimes B[\gamma]} \otimes L^{\alpha,\beta} \qquad \dfrac{}{\Gamma, A \otimes B[\gamma] \vdash A \otimes B[\gamma]}\,\mathsf{id}_{A \otimes B}$$

and

$$\dfrac{\dfrac{\mathcal{D}_1 \quad \mathcal{D}_2}{\Gamma \vdash A[p] \quad \Gamma \vdash B[q]} \otimes R \quad \dfrac{\mathcal{E}'}{\Gamma, A \otimes B[\gamma], A[\alpha], B[\beta] \vdash C[\alpha * \beta * r]}}{\Gamma, A \otimes B[\gamma] \vdash C[\gamma * r]} \otimes L^{\alpha,\beta}}{\Gamma \vdash C[p * q * r]}\,\mathsf{cut}^{\gamma}_{A \otimes B}$$

Now γ is fresh in the cut, so it does not appear in $\alpha * \beta * r$. So we can apply *strengthening* to eliminate it from $\mathcal{E}'$. Then we can have two appeals to cut at smaller propositions.

$$\dfrac{\dfrac{\mathcal{D}_1 \quad \mathcal{D}_2}{\Gamma \vdash A[p] \quad \Gamma \vdash B[q]} \otimes R \quad \dfrac{\mathcal{E}'}{\Gamma, A \otimes B[\gamma], A[\alpha], B[\beta] \vdash C[\alpha * \beta * r]}}{\Gamma, A \otimes B[\gamma] \vdash C[\gamma * r]} \otimes L^{\alpha,\beta}}{\Gamma \vdash C[p * q * r]}\,\mathsf{cut}^{\gamma}_{A \otimes B}$$

$$\longrightarrow_R \quad \dfrac{\dfrac{\mathcal{D}_2}{\Gamma \vdash B[q]} \quad \dfrac{\dfrac{\dfrac{\mathcal{D}_1}{\Gamma \vdash A[p]}\,\mathsf{weaken}}{\Gamma, B[\beta] \vdash A[p]} \quad \dfrac{\mathcal{E}'}{\Gamma, A[\alpha], B[\beta] \vdash C[\alpha * \beta * r]}\,\mathsf{strengthen}}{\Gamma, B[\beta] \vdash C[p * \beta * r]}\,\mathsf{cut}^{\alpha}_{A}}{\Gamma \vdash C[p * q * r]}\,\mathsf{cut}^{\beta}_{B}$$

There is an implicit use of weakening to equalize the antecedents in the two premises in the cut on A.

Lemma 1 (Strengthening and Weakening)

(i) If $\Gamma, A[\alpha] \vdash C[p]$ *where* α *is not in* p*, then* $\Gamma \vdash C[p]$ *with the same derivation.*

(ii) If $\Gamma \vdash C[p]$ *then* $\Gamma, A[\alpha] \vdash C[p]$ *when* α *is not in* p*, with the same derivation.*

Because they follow largely similar considerations, we just show the rules for implication and external choice. Note how in external choice the same resources are required for both branches. The left rule for disjunction will have a similar property.

$$\frac{\Gamma, A[\alpha] \vdash B[p * \alpha]}{\Gamma \vdash A \multimap B[p]} \multimap R^{\alpha} \qquad \frac{\Gamma, A \multimap B[\gamma] \vdash A[p] \quad \Gamma, A \multimap B[\gamma], B[\beta] \vdash C[q * \beta]}{\Gamma, A \multimap B[\gamma] \vdash C[p * q * \gamma]} \multimap L^{\beta}$$

$$\frac{\Gamma \vdash A[p] \quad \Gamma \vdash B[p]}{\Gamma \vdash A \mathbin{\&} B[p]} \mathbin{\&} R$$

$$\frac{\Gamma, A \mathbin{\&} B[\gamma], A[\alpha] \vdash [p * \alpha]}{\Gamma, A \mathbin{\&} B[\gamma] \vdash C[p * \gamma]} \mathbin{\&} L_1^{\alpha} \qquad \frac{\Gamma, A \mathbin{\&} B[\gamma], B[\beta] \vdash [p * \beta]}{\Gamma, A \mathbin{\&} B[\gamma] \vdash C[p * \gamma]} \mathbin{\&} L_2^{\beta}$$

The system has been carefully designed to preserve the structure of proofs as much as possible. This makes the proof of adequacy relatively easy.

The first direction states that if $A_1, \ldots, A_n \vdash A$ then $A_1[\alpha_1], \ldots, A_n[\alpha_n] \vdash A[\alpha_1 * \cdots * \alpha_n]$ where for distinct variable α_i. This is proved by induction over the given derivation, taking advantage of weakening to add on the antecedents that remain in the premises of the target calculus.

For the other direction, we go from $\Gamma \vdash A[p]$ to $\Gamma|_p \vdash A$, where $\Gamma|_p$ is the restriction of Γ to the resource variables in p. For this direction, we assume that p does not have any repeated resource variables—if it did, we would not be able to model the structural proof with a linear one.

3 Adding Validity

Perhaps surprisingly, we already have all the expressive power we need in the target calculus to model at least $!A = {\downarrow}{\uparrow}A$. The key idea is that a structural proposition A_{S} corresponds to a new kind of antecedent $A[\epsilon]$. This expresses that A can be proved without the use of any resources, which is precisely our definition of validity in linear logic!

We restrict ourselves to the single structural proposition $\uparrow A_{\mathsf{L}}$ but we believe it should easily extend to allow further structural propositions (after all, our target calculus is structural).

We look at the rules for the shifts of the mixed linear/nonlinear logic and derive the corresponding resource-aware rules.

We start with the right rule for $\downarrow A_{\mathsf{S}}$, which is not invertible since $\downarrow$ is a positive connective.

$$\frac{\Delta_{\mathsf{S}} \vdash A_{\mathsf{S}}}{\Delta_{\mathsf{S}} \vdash {\downarrow}A_{\mathsf{S}}} \downarrow R \qquad \frac{\Gamma \vdash A[\epsilon]}{\Gamma \vdash {\downarrow}A[\epsilon]} \downarrow R$$

The restriction that Δ consists only of structural propositions is represented here by the fact that $\downarrow A$ must be true without any resource (ϵ). If there were usable resources in Γ, the would show up in the resources for the succedent.

In the left rule (which is invertible), α must be an available resource, but that the only requirement. In the mixed linear/nonlinear system this means the succedent must be linear.

$$\frac{\Delta, A_{\mathsf{S}} \vdash C_{\mathsf{L}}}{\Delta, \downarrow A_{\mathsf{S}} \vdash C_{\mathsf{L}}}\ \downarrow L \qquad \frac{\Gamma, \downarrow A[\alpha], A[\epsilon] \vdash C[p]}{\Gamma, \downarrow A[\alpha] \vdash C[p * \alpha]}\ \downarrow L$$

Next, we come to $\uparrow R$. In mixed linear/nonlinear logic, the presupposition of independence ensures that the antecedents in the conclusion are all structural, indicated by writing Δ_{S}. In the corresponding rule in resource logic there may be antecedents $B[\beta]$, but they cannot be used because the the succedent has the empty set of resources ϵ.

$$\frac{\Delta_{\mathsf{S}} \vdash A_{\mathsf{L}}}{\Delta_{\mathsf{S}} \vdash \uparrow A_{\mathsf{L}}}\ \uparrow R \qquad \frac{\Gamma \vdash A[\epsilon]}{\Gamma \vdash \uparrow A[\epsilon]}\ \uparrow R$$

The left rule codifies the idea that if $\uparrow A[\epsilon]$ we can obtain a copy of the resource A without using any resources. After all, it doesn't cost any resources to do so!

$$\frac{\Delta, \uparrow A_{\mathsf{L}}, A_{\mathsf{L}} \vdash C_{\mathsf{L}}}{\Delta, \uparrow A_{\mathsf{L}} \vdash C_{\mathsf{L}}}\ \uparrow L \qquad \frac{\Gamma, \uparrow A[\epsilon], A[\alpha] \vdash C[p * \alpha]}{\Gamma, \uparrow A[\epsilon] \vdash C[p]}\ \uparrow L^{\alpha}$$

At this point in lecture we were concerned about $\uparrow R$ and $\uparrow L$. Because $\uparrow$ is negative, the right rule should be invertible and the left rule should not. We therefore applied our identity expansion test. First, the correct proof that starts with the right rule.

$$\dfrac{\dfrac{\dfrac{\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots}{\Gamma, \uparrow A[\epsilon], A[\alpha] \vdash A[\alpha]}\ \mathsf{id}_A}{\Gamma, \uparrow A[\epsilon] \vdash A[\epsilon]}\ \uparrow L^{\alpha}}{\Gamma, \uparrow A[\epsilon] \vdash \uparrow A[\epsilon]}\ \uparrow R$$

We would not expect the opposite order to work out, which would be evidence that $\uparrow L$ is not invertible.

$$\dfrac{\dfrac{??}{\uparrow A[\epsilon], A[\alpha] \vdash \uparrow A[\alpha]}\ \uparrow L^{\alpha}}{\uparrow A[\epsilon] \vdash \uparrow A[\epsilon]}$$

Luckily we do get stuck here, because we can not apply the $\uparrow R$ rule. If A is atomic, the only option is to apply $\uparrow L$ again and again, obtaining many copies of A but never being able to apply the right rule.

In order to account for this we have to update our adequacy proof for the translation. In the first direction:

(i) If $\Delta_{\mathsf{S}}, \Delta_{\mathsf{L}} \vdash A_{\mathsf{L}}$ then $\Delta_{\mathsf{S}}[\epsilon], \Delta_{\mathsf{L}}[\overline{\alpha}] \vdash A_{\mathsf{L}}[\overline{\alpha}]$

(ii) If $\Delta_{\mathsf{S}} \vdash A_{\mathsf{S}}$ then $\Delta_{\mathsf{S}}[\epsilon] \vdash A_{\mathsf{S}}[\epsilon]$

This direction takes advantage of weakening in resource logic (Lemma 1).

For the other direction, we have:

If $\Gamma \vdash A[p]$ with $\Gamma = \Gamma_1[\epsilon], \Gamma_2[\overline{\alpha}]$ then $\Gamma|_p \vdash A$.

Here, $\Gamma|_p$ restricts Γ to antecedents $B[\epsilon]$ and $B[\alpha_i]$ for α_i in p. For this direction we use strengthening which applies due to our presuppositions on antecedents and resource terms.

4 Untethering

All left rules for linear antecedents refer to the resources annotating the succedent. We say the left rules are *tethered* to the succedent. This is by design, since we would like to model the rules of linear logic as closely as possible. This feature makes it somewhat difficult to turn the system into a *translation* from propositional linear logic to intuitionistic logic.

As a step in the direction of a translation we'll *untether* the rules for the *negative connectives*. We don't investigate this system in its own right, just using it for intuition. The first step is untethering is to allow complex resource terms for antecedents. We think of $A[p]$ as saying that the justification of the antecedent A requires resources p.

The identity is straightforward; it is where the resources among the antecedents are eventually tied to the succedent.

$$\frac{}{\Gamma, A[p] \vdash A[p]}\ \mathsf{id}$$

Next consider implication. While the right rule is unchanged, the new left rule is untethered.

$$\frac{\Gamma, A[\alpha] \vdash B[p * \alpha]}{\Gamma \vdash A \multimap B[p]} \multimap R^{\alpha} \qquad \frac{\Gamma \vdash A[q] \quad \Gamma, B[p * q] \vdash C[r]}{\Gamma, A \multimap B[p] \vdash C[r]} \multimap L$$

We can read the left rule as: "*if the implication $A \multimap B$ requires resources p and A requires resources q, then B requires resources $p * q$*". Thinking about the cut reduction (in a new form), we would instantiate the α in $\multimap R^{\alpha}$ with q so we can reduce the cut between $\multimap R$ and $\multimap L$.

External choice can be untethered more directly.

$$\frac{\Gamma \vdash A[p] \quad \Gamma \vdash B[p]}{\Gamma \vdash A \,\&\, B[p]}\ \&R$$

$$\frac{\Gamma, A \,\&\, B[p], A[p] \vdash C[r]}{\Gamma, A \,\&\, B[p] \vdash C[r]}\ \&L_1 \qquad \frac{\Gamma, A \,\&\, B[p], B[p] \vdash C[r]}{\Gamma, A \,\&\, B[p] \vdash C[r]}\ \&L_2$$

It is interesting to consider what happens if, say, $p = \alpha$. In that case, after the $\&L_1$ rule, we have both $A \,\&\, B[\alpha]$ and $A[\alpha]$ among the antecedents. But eventually the connection to the resources in the succedent will have to made, and then only one of these two can be used. We could even take it a step further, applying $\&L_2$ so we have both $A[\alpha]$ and $B[\alpha]$ among the antecedents, but only one of them can be used because of resource constraints. From these considerations is should be clear that the structure of the proofs no longer matches up so directly.

We now turn the idea of untethering into a translation $A \mathbin{@} p$.

$$\begin{array}{lcl} (P) \mathbin{@} p & = & P(p) \\ (A \multimap B) \mathbin{@} p & = & \forall\alpha.(A \mathbin{@} \alpha) \supset (B \mathbin{@} (p * \alpha)) \\ (A \,\&\, B) \mathbin{@} p & = & (A \mathbin{@} p) \wedge (B \mathbin{@} p) \end{array}$$

Recall also the equations on resource terms:

Associativity	$(p * q) * r = p * (q * r)$
Unit	$\epsilon * p = p = p * \epsilon$
Commutativity	$p * q = q * p$

As a simple example, $(P \,\&\, Q) \multimap P \mathbin{@} \epsilon$ is translated to $\forall\alpha.\, P(\alpha) \wedge Q(\alpha) \supset P(\alpha)$ (which is true). $P \multimap (Q \multimap P) \mathbin{@} \epsilon$ is $\forall\alpha.P(\alpha) \supset \forall\beta.\, Q(\beta) \supset P(\alpha * \beta)$ which is not true.

The adequacy theorem states that $\cdot \vdash A$ if and only if $\cdot \vdash A \mathbin{@} \epsilon$ in first-order intuitionistic logic with the stated laws of equality. These laws, plus reflexivity, symmetry, transitivity, and congruence can be written as axioms in the predicate calculus to complete this translation. The proof roughly models the untethered rules given earlier in this section.

You can find an extension to the positive connectives and a proof of adequacy in an unpublished paper by Reed and Pfenning [2010]. This paper further shows that the same recipe applies to ordered logic, and that one can organize the translation so that focusing phases are preserved across the translation. The close relationship to *hybrid logic* is further explored by Reed [2009].

References

Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

Patrick Lincoln, John Mitchell, Andre Scedrov, and Natarajan Shankar. Decision problems for propositional linear logic. *Annals of Pure and Applied Logic*, 56:239–311, April 1992.

Jason C. Reed. *A Hybrid Logical Framework*. PhD thesis, Carnegie Mellon University, September 2009. Available as Technical Report CMU-CS-09-155.

Jason C. Reed and Frank Pfenning. Focus-preserving embeddings of substructural logics in intuitionistic logic. Unpublished Manuscript, January 2010. URL http://www.cs.cmu.edu/~fp/papers/substruct10.pdf.

Alfred Tarski. The concept of truth in formalized languages. In John Corcoran and J. H. Woodger, editors, *Logic, Semantics, Metamathematics*, pages 152–278. Clarendon Press, Oxford, 1956. Translation of a paper from 1931.

Lecture Notes on Logical Frameworks

15-836: Substructural Logics
Frank Pfenning

Lecture 20
November 28, 2023

1 Introduction

A *logical framework* consists of a *formal metalanguage* for the definition of logics and other deductive systems and a *representation methodology*. The seminal work on logical frameworks is LF [Harper et al., 1987, 1993], with a full-scale implementation in the Twelf system [Pfenning and Schürmann, 1999], available at `www.twelf.org`. Logical frameworks distill the essence of the conceptual notions that are used to define logics and other deductive systems, such as the statics and dynamics of programming languages. They are distinguished from general type theories and their implementations in systems such as Agda[1] and Coq[2] in that they designed for the specific domain of deductive systems rather than general (constructive or classical) mathematics.

LF itself is *structural*, and this limitation has led to substructural generalizations in the form of Linear LF (LLF) [Cervesato and Pfenning, 1996, 2002][3] and Concurrent LF (CLF) [Watkins et al., 2002, Cervesato et al., 2002, Schack-Nielsen and Schürmann, 2008, Schack-Nielsen, 2011][4] Each of these is designed to address some shortcomings of its predecessors, suitably extending both the formal metalanguage and the representation methodology.

We will follow a similar path, introducing LF in today's lecture and then consider substructural extensions in the next two lectures. We emphasize the universality of the underlying principles, which mirror the principles we have employed throughout in this course. One might even say that these principles are manifest in the design of the logical frameworks we represent.

[1] `https://wiki.portal.chalmers.se/agda/pmwiki.php`
[2] `https://coq.inria.fr/`
[3] `https://github.com/clf/llf`
[4] `https://github.com/clf/celf`

What is a logical framework used for? In this and the three remaining lectures we don't have the time to cover the full rangle of applications, but we can broadly categorize them as follows:

Definition: Logical frameworks are used to *define* logics and other deductive systems under consideration, ideally at a very high level of abstraction. Logics are generally characterized by propositions and rules of inference, programming languages by programs, type systems, and rules of computation. The main principle used in the definition of logics is summarized as *judgments as types* and *proofs as objects*.

Algorithms: The most fundamental algorithm is that of proof checking for an object logic represented in a logical framework, but there are many others such as proof search, proof reduction, translations between logics, type checking, or evaluation of programs on an object language. The logical frameworks we consider support algorithms via *computation as proof construction* which encompasses both backward [Miller et al., 1991, Miller and Nadathur, 2012] and forward proof construction [López et al., 2005].

Metareasoning: Once a deductive system has been defined, we usually prove a number of important properties of it. For logics, these include cut elimination, identity elimination, focusing, soundness and completeness of translations, etc. For programming languages they are progress and preservation, soundness and completeness of type-checking algorithms, compiler correctness, etc. We can exploit the nature of representations in logical framework to formally prove such metatheorems Pfenning and Schürmann [1999], Schürmann [2000]. The general methodology is to represent that computational content of proof of the metatheorem algorithmically and then verify its totality. There are some gaps in our understanding of how to achieve this for substructural frameworks (see some approaches by McCreight and Schürmann [2008], Reed [2009], Georges et al. [2017])

For today's lecture where we only consider LF we will focus on logic definition and proof checking.

2 Judgments as Types

One of the fundamental representation techniques is *judgments as types*. A *judgment* in this context are what is subject to deductive inference, as mapped out by Martin-Löf [1983]. Common judgments are $A\ \mathit{true}$ or $A\ \mathit{false}$ or $A\ \mathit{valid}$. We use the very simply example from Lecture 1 of defining a path through a directed graph.

$$\frac{\mathsf{edge}(x,y)}{\mathsf{path}(x,y)}\ \mathsf{step} \qquad \frac{\mathsf{path}(x,y) \quad \mathsf{path}(y,z)}{\mathsf{path}(x,z)}\ \mathsf{trans}$$

We previously thought of edge(x, y) and path(x, y) as propositions, but we will now think of them as judgments since they are directly subject to inference. This means that edge(x, y) and path(x, y) for vertices x and y should be represented by *types* in our formal metalanguage (which we have yet to define). We think of both edge and path as constructors for types, taking *vertices* as arguments. So we have

```
vertex : type.
edge : vertex -> vertex -> type.
path : vertex -> vertex -> type.
```

In the terminology of logical frameworks we call edge and path *type families*, each indexed by two vertices.

It is easy to see what the type vertex represents. Here is the example from Lecture 1: Our initial state of knowledge is edge(a, b), edge(b, c), edge(b, d) for some vertices a, b, c, and d. Therefore:

```
a : vertex.
b : vertex.
c : vertex.
d : vertex.
```

We also have to have objects of type edge a b, edge b c and edge b d. These are constants represent (trivial) *proofs* of these judgments.

```
eab : edge a b.
ebc : edge b c.
ebd : edge b d.
```

What happens to the inference rules? They become *proof constructors*. As a first approximation, we might write

```
step : edge x y -> path x y.
trans : path x y -> path y z -> path x z.
```

It remains to clarify the status of x, y and z. Somehow we have to express that any instantiation of the constructor with vertices x, y and z is a valid instance of the rule.

When we developed predicate calculus, we used universal quantification to express this, where we purposely let the quantifier range over arbitrary "individuals". In LF we would like to be more precise and specify that they must be vertices. Syntactically, we just replace $\forall x.\, B(x)$ with $\Pi x{:}A.B(x)$ where A is a type. Our concrete syntax for this is $\{x : A\}B$.

```
step : {x:vertex} {y:vertex} edge x y -> path x y.
trans : {x:vertex} {y:vertex} {z:vertex}
           path x y -> path y z -> path x z.
```

Now we can show a little example of a proof representation. We represent

$$\dfrac{\dfrac{}{\mathsf{edge}(a,b)}\,\mathsf{eab}}{\mathsf{path}(a,b)}\,\mathsf{step}$$

as

```
step a b eab : path a b
```

Here is a slightly larger proof:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{edge}(a,b)}\,\mathsf{eab}}{\mathsf{path}(a,b)}\,\mathsf{step} \quad \dfrac{\dfrac{}{\mathsf{edge}(b,c)}\,\mathsf{ebc}}{\mathsf{path}(b,c)}\,\mathsf{step}}{\mathsf{path}(a,c)}\,\mathsf{trans}$$

which becomes

```
trans a b c (step a b eab) (step b c ebc) : path a c
```

3 The Formal Metalanguage

So far, we have learned about "*judgments as types*" and "*proofs as objects*" through a very simple example. A proof of a judgment is represented by an object of the corresponding type. Clearly, this should be a *bijection*: every valid proof should be an object that has the expected type, and every object that has a given type should represent a type. If typing in the framework is decidable (which it will be) this means we can model proof checking by type checking.

Before we go further, we should be more precise about the metalanguage that we have written some code in without actually defining it. Even though LF was not originally conceived this way, we think of it as arising from [*drum roll*] *focusing*. This point of view is helpful because it will extend to the substructural frameworks we start discussing in the next lecture.

What have we used to far? We have used atomic types vertex, edge x y and path x y. We have also used function types $A \to B$ and quantification $\Pi x{:}A.B(x)$ that generalizes $\forall x.\, B(x)$. We observe that all of these are *negative*! This also alleviates any stress regarding atoms: let's just be consistent and also make them negative. Here is what we have so far.

Negative types	A, B	$::=$	$P \mid A \to B \mid \Pi x{:}A.\, B(x)$
Atoms	P	$::=$	$\ldots$
Objects	M	$::=$	$\ldots$
Kinds	K	$::=$	$\mathbf{type} \mid A \to K \mid \ldots$
Signatures	Σ	$::=$	$\cdot \mid \Sigma, a : K \mid \Sigma, c : A$

A *signature* has declarations for term constructors c and also for type families a that may depend on objects like edge and path.

The language of objects (and, by analogy, the language of types) is now determined by what it means to *focus* on a type among the antecedents and what it means to *invert* a type as a succedent. The declarations $c : A$ in signature Σ (which is generally fixed for a particular encoding) act as antecedents.

We start with left focus, first the rules that starts left focus from a *stable sequent*. At the end of this section we will see what the succedent δ of a stable sequent must look like.

$$\frac{c : A \in \Sigma \quad \Gamma, [A] \vdash_\Sigma \delta}{\Gamma \vdash_\Sigma \delta}\ \mathsf{FL/C} \qquad \frac{c : A \in \Sigma \quad \Gamma, [A] \vdash_\Sigma S : \delta}{\Gamma \vdash_\Sigma c\ S : \delta}\ \mathsf{FL/C}$$

The kind of proof term to we assign to the left focus judgment is called a *spine* [Cervesato and Pfenning, 2003], which we write as S. This harkens back to earlier term assignments, although with a different purpose.

We now omit the signature Σ from the turnstile for brevity since it never changes in the typing of objects and spines.

$$\frac{\Gamma \vdash [A] \quad \Gamma, [B] \vdash \delta}{\Gamma, [A \to B] \vdash \delta}\ {\to}L \qquad \frac{\dfrac{\Gamma \vdash M : A}{\Gamma \vdash M : [A]}\ \mathsf{IR/FR} \quad \Gamma, [B] \vdash S : \delta}{\Gamma, [A \to B] \vdash (M\ ;\ S) : \delta}\ {\to}L$$

Since A is negative, we will lose focus on $[A]$ in the first premise and start inversion which is the judgment to type objects (not spines).

Universal quantification is interesting. Recall that a proof of $\forall x.\, B(x)$ was a function which for every individual t returned a proof of $B(t)$. So both quantification and implication correspond to functions. Here, there is no separate class of terms t—we just use objects M.

$$\frac{\dfrac{\Gamma \vdash M : A}{\Gamma \vdash M : [A]}\ \mathsf{IR/FR} \quad \Gamma, [B(M)] \vdash S : \delta}{\Gamma, [\Pi x{:}A.\, B(x)] \vdash (M\ ;\ S) : \delta}\ \Pi L$$

The tricky part of this rule is the substitution $B(M)$. So the only difference between $A \to B$ and $\Pi x{:}A.\, B(x)$ that in the latter, B may depend on x, while not so in the former. We discuss this operation further in Section 4.

For the final left rule, we consider atoms, which in LF are all considered negative. The left focus only succeeds if the succedent is the same suspended atom. Because of this, there is no real information content in the rule and the spine is just empty.

$$\frac{}{\Gamma, [P] \vdash \langle P \rangle}\ \mathsf{id}^- \qquad \frac{}{\Gamma, [P] \vdash (\,) : \langle P \rangle}\ \mathsf{id}^-$$

We revisit our grammar. Atoms are just like constants applied to spines, except that the constant itself is a type family. Also, variables can be used just like constants.

Negative types	A, B	$::=$	$P \mid A \to B \mid \Pi x{:}A.\, B(x)$
Atoms	P	$::=$	$a\ S$
Objects	M	$::=$	$c\ S \mid x\ S \mid \ldots$
Spines	S	$::=$	$(M\ ;\ S) \mid (\,)$
Kinds	K	$::=$	$\mathbf{type} \mid A \to K \mid \ldots$
Signatures	Σ	$::=$	$\cdot \mid \Sigma, a : K \mid \Sigma, c : A$

With this, we can give a formal representation of our earlier example, abbreviating $c\ (\,)$ and $x\ (\,)$ as just c and x. We also omit trailing empty spines and write $(M_1\ ;\ \ldots\ ;\ M_n\ ;\ (\,))$ as $(M_1\ ;\ \ldots\ ;\ M_n)$.

$\mathsf{vertex} : \mathbf{type}$
$\mathsf{edge} : \mathsf{vertex} \to \mathsf{vertex} \to \mathbf{type}$
$\mathsf{path} : \mathsf{vertex} \to \mathsf{vertex} \to \mathbf{type}$
$\mathsf{step} : \Pi x{:}\mathsf{vertex}.\, \Pi y{:}\mathsf{vertex}.\, \mathsf{edge}\ (x\ ;\ y) \to \mathsf{path}\ (x\ ;\ y)$
$\mathsf{trans} : \Pi x{:}\mathsf{vertex}.\, \Pi y{:}\mathsf{vertex}.\, \Pi z{:}\mathsf{vertex}.$
$\qquad \mathsf{path}\ (x\ ;\ y) \to \mathsf{path}\ (y\ ;\ z) \to \mathsf{path}\ (x\ ;\ z)$

$a : \mathsf{vertex}$
$b : \mathsf{vertex}$
$c : \mathsf{vertex}$
$d : \mathsf{vertex}$

$\mathsf{eab} : \mathsf{edge}\ (a\ ;\ b)$
$\mathsf{ebc} : \mathsf{edge}\ (b\ ;\ c)$
$\mathsf{ebd} : \mathsf{edge}\ (b\ ;\ d)$

$\vdash \mathsf{trans}\ (a\ ;\ b\ ;\ c\ ;\ (\mathsf{step}\ (a\ ;\ b\ ;\ \mathsf{eab}))\ ;\ (\mathsf{step}\ (b\ ;\ c\ ;\ \mathsf{ebc}))) : \mathsf{path}\ (a\ ;\ b)$

There is still a lot of redundancy in this representation with multiple occurrences of a, b, and c, but implementations can further mitigate this by allowing the user to elide some of these, and in some cases even eliminate them from the representation altogether.

Even if it is only needed to suspend atoms in this example, we should return to the inversion phase of focusing. Since all constructors are negative, these will be the right rules. For the same reason, we can dispense with the usual ordered antecedents Ω. For example, the right rule for $A \to B$ would add A to the ordered context, but since A is negative and therefore stable, it will be immediately transferred to the structural context Γ that consists entirely of negative types (since suspended positive atoms are not part of the language).

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ {\to}R \qquad \frac{\Gamma, x : A \vdash M(x) : B}{\Gamma \vdash \lambda x.\, M(x) : A \to B}\ {\to}R$$

As we might expect by now, quantifiers are just dependent function types can behave the same.

$$\frac{\Gamma, x : A \vdash M(x) : B(x)}{\Gamma \vdash \lambda x.\, M(x) : \Pi x{:}A.\, B(x)}\ \Pi R$$

Finally, atoms on the right are suspended because their are negative.

$$\frac{\Gamma \vdash \langle P \rangle}{\Gamma \vdash P}\ \mathsf{C/IR} \qquad \frac{\Gamma \vdash M : \langle P \rangle}{\Gamma \vdash M : P}\ \mathsf{C/IR}$$

We see that in stable sequents the succedent δ always has the form $\langle P \rangle$ for some atom P.

This allows us to complete the grammar, where we additional allow kinds to be dependent.

Negative types	A, B	$::=$	$P \mid A \to B \mid \Pi x{:}A.\, B(x)$
Atoms	P	$::=$	$a\ S$
Objects	M	$::=$	$c\ S \mid x\ S \mid \lambda x.\, M(x)$
Spines	S	$::=$	$(M\ ;\ S) \mid (\,)$
Kinds	K	$::=$	$\mathbf{type} \mid A \to K \mid \Pi x{:}A.\, B(x)$
Stable antecedents	Γ	$::=$	$\cdot \mid \Gamma, x : A$
Stable succedents	δ	$::=$	$\langle P \rangle$
Signatures	Σ	$::=$	$\cdot \mid \Sigma, a : K \mid \Sigma, c : A$

We have already seen the most critical typing rules; they are summarized in Figure 1. Others are similar and elided and can be found in the literature. Still missing is the definition of $B(M)$, which looks like ordinary substitution but is more complicated.

In the next lecture we will extend the representation methodology and show some representations of the sequent calculus and the semi-axiomatic sequent calculus.

4 Hereditary Substitution

Consider a type $\Pi x{:}A.\, B(x)$. When typing an application we need to substitute a term $M : A$ for x, written so far as $B(M)$. But does this substitution actually make sense? Consider the term $x\ (\,)$ that is typed from $x : P$ with left focus. Just plugging in the object $M : P$ would be $M\ (\,)$, but that's not actually a valid object. Similarly, if $A = P \to Q$ then the term will be $\lambda y.\, N(y)$ and after just plugging into $x\ (M\ ;\ (\,))$ we would have $(\lambda y.\, N(y))\ (M\ ;\ (\,))$, again not even syntactically valid.

We use a more traditional notation $[M/x]_A B(x)$ instead of $B(M)$, indexing the operation also with the type A of x. This quickly reduces to $[M/x]_A N$ and $[M/x]_A S$. The idea is that if x is at the head of a spine we then reduce further, initiating more

$$\frac{c : A \in \Sigma \quad \Gamma, [A] \vdash S : \delta}{\Gamma \vdash c\ S : \delta}\ \mathsf{FL/C} \qquad \frac{x : A \in \Gamma \quad \Gamma, [A] \vdash S : \delta}{\Gamma \vdash c\ S : \delta}\ \mathsf{FL/C/}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma, [B] \vdash S : \delta}{\Gamma, [A \to B] \vdash (M\ ;\ S) : \delta}\ {\to}L \qquad \frac{\Gamma \vdash M : A \quad \Gamma, [B(M)] \vdash S : \delta}{\Gamma, [\Pi x{:}A.\ B(x)] \vdash (M\ ;\ S) : \delta}\ \Pi L$$

$$\frac{}{\Gamma, [P] \vdash (\,) : \langle P \rangle}\ \mathsf{id}^-$$

$$\frac{\Gamma, x : A \vdash M(x) : B}{\Gamma \vdash \lambda x.\ M(x) : A \to B}\ {\to}R \qquad \frac{\Gamma, x : A \vdash M(x) : B(x)}{\Gamma \vdash \lambda x.\ M(x) : \Pi x{:}A.\ B(x)}\ \Pi R$$

$$\frac{\Gamma \vdash M : \langle P \rangle}{\Gamma \vdash M : P}\ \mathsf{C/IR}$$

Figure 1: LF Type Theory (excerpt), given a fixed signature Σ

substitutions and so on. Why does this terminate? Similar to cut elimination, it is by a nested induction first on the type A and second on the object M and spine S we substitute into. In fact, it is the operational reading of cut elimination for the focusing calculus on negative types.

We write h for a *head*, that is, a constant c or a variable y.

$$\begin{array}{lcll}
[M/x]_A(\lambda y.\ N) & = & \lambda y.\ [M/x]_A N & y \text{ not free in } M \\
[M/x]_A(h\ S) & = & h\ [M/x]_A S & \text{where } x \neq h \\
[M/x]_A(x\ S) & = & M\ |_A\ [M/x]_A S & \text{(application)} \\
[M/x]_A(N\ ;\ S) & = & [M/x]_A N\ ;\ [M/x]_A S & \\
[M/x]_A(\,) & = & (\,) & \\
 & & & \\
(h\ S)\ |_P\ (\,) & = & h\ S & \\
(\lambda x.\ M)\ |_{A \to B}\ (N\ ;\ S) & = & [N/x]_A M\ |_B\ S & \\
(\lambda x.\ M)\ |_{\Pi x:A.\ B(x)}\ (N\ ;\ S) & = & [N/x]_A M\ |_{B(x)}\ S &
\end{array}$$

The condition in the first case can be satisfied by renaming the bound variable y, which is always (silently) possible. For the purpose of hereditary substitution, ordinary and dependent function types are treated identically; the free variable x in $B(x)$ is not relevant to the termination argument.

Another interesting point is that hereditary substitution may be undefined, but is always computable by the nested induction argument. The notion of hereditary substitution was originally developed for a substructural logical framework [Watkins et al., 2002] which contains LF as a fragment.

References

Iliano Cervesato and Frank Pfenning. A linear logical framework. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, pages 264–275, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information & Computation*, 179(1):19–75, November 2002. Revised and expanded version of an extended abstract, LICS 1996, pp. 264-275.

Iliano Cervesato and Frank Pfenning. A linear spine calculus. *Journal of Logic and Computation*, 13(5):639–688, 2003.

Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

Aïna Linn Georges, Agata Murawska, Shawn Otis, and Brigitte Pientka. LINCX: A linear logical frameworks with first-class contexts. In Hongseok Yang, editor, *26th European Symposium on Programming (ESOP 2017)*, pages 530–555, Uppsala, Sweden, April 2017. Springer LNCS 10201.

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204. IEEE Computer Society Press, June 1987.

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In A.Felty, editor, *Proceedings of the 7th International Symposium on Principles and Practice of Declarative Programming (PPDP'05)*, pages 35–46, Lisbon, Portugal, July 2005. ACM Press.

Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. Notes for three lectures given in Siena, Italy. Published in *Nordic Journal of Philosophical Logic*, 1(1):11-60, 1996, April 1983. URL http://www.hf.uio.no/ifikk/forskning/publikasjoner/tidsskrifter/njpl/vol1no1/meaning.pdf.

Andrew McCreight and Carsten Schürmann. A meta linear logical framework. In *4th International Workshop on Logical Frameworks and Meta-Languages (LFM 2004)*, volume 199 of *Electronic Notes in Theoretical Computer Science*, pages 129–147, February 2008.

Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012.

Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

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, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

Jason C. Reed. *A Hybrid Logical Framework*. PhD thesis, Carnegie Mellon University, September 2009. Available as Technical Report CMU-CS-09-155.

Anders Schack-Nielsen. *Implementing Substructural Logical Frameworks*. PhD thesis, IT University of Copenhagen, January 2011.

Anders Schack-Nielsen and Carsten Schürmann. Celf - a logical framework for deductive and concurrent systems. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning (IJCAR'08)*, pages 320–326, Sydney, Australia, August 2008. Springer LNCS 5195.

Carsten Schürmann. *Automating the Meta Theory of Deductive Systems*. PhD thesis, Department of Computer Science, Carnegie Mellon University, August 2000. Available as Technical Report CMU-CS-00-146.

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

Lecture Notes on Substructural Frameworks

15-836: Substructural Logics
Frank Pfenning

Lecture 21
November 30, 2023

1 Introduction

In the last lecture we introduced LF, although we only started to talk about the representation methodology. We continue this today, discuss some of the shortcomings, and explore how they might be addressed in a substructural logical framework. References to LF and its implementation in Twelf are provided in the previous lecture.

2 Representing Sequent Derivations

A more typical example for the use of a logical framework is the representation of a logic. Actually, we should be more precise: it is not a logic we represent but a specific inference system. So, for example, sequent calculus and the semi-axiomatic sequent calculus will have different representations.

First, we start with the representation of the propositions of (intuitionistic) logic. That's straightforward:

```
prop : type.
and : prop -> prop -> prop.
or : prop -> prop -> prop.
imp : prop -> prop -> prop.
```

Atomic propositions are either variables of type prop, or declared in addition to the logical connectives we already have.

For different constituents of an object logic like propositions or proofs, we have an (overloaded) representation function $\ulcorner - \urcorner$. For example, for propositions we have

$$P_1{:}\mathsf{prop}, \ldots, P_k{:}\mathsf{prop} \vdash \ulcorner A \urcorner : \mathsf{prop}$$

defined by

$$\begin{array}{lcl} \ulcorner P \urcorner & = & P \\ \ulcorner A \wedge B \urcorner & = & \mathsf{and}\ \ulcorner A \urcorner\ \ulcorner B \urcorner \\ \ulcorner A \vee B \urcorner & = & \mathsf{or}\ \ulcorner A \urcorner\ \ulcorner B \urcorner \\ \ulcorner A \supset B \urcorner & = & \mathsf{imp}\ \ulcorner A \urcorner\ \ulcorner B \urcorner \end{array}$$

Here, instead of $c\ (M_1\ ;\ \ldots\ ;\ M_n)$ from focusing we write $c\ M_1\ \ldots M_n$, which is the familiar source-level syntax from logical frameworks based on natural deduction.

When it comes to judgments, recall that basic judgments are represented as types. But what are the basic judgments here? It turns out the correct representation uses two: "A is an antecedent" and "A is a succedent", which we write as $\mathsf{ante}\ \ulcorner A \urcorner$ and $\mathsf{succ}\ \ulcorner A \urcorner$ respectively. A proof

$$\begin{array}{c} \mathcal{D} \\ A_1, \ldots, A_n \vdash C \end{array}$$

is then represented by the LF sequent

$$x_1 : \mathsf{ante}\ \ulcorner A_1 \urcorner, \ldots x_n : \mathsf{ante}\ \ulcorner A_n \urcorner \vdash_\Sigma \ulcorner \mathcal{D} \urcorner : \mathsf{succ}\ \ulcorner C \urcorner$$

where $\ulcorner \mathcal{D} \urcorner$ is an object of LF and the signature Σ contains the constructors for propositions and proofs. Actually, we have to modify this slightly if $A_1, \ldots, A_n, C$ contain propositional variables P_i, in which case it becomes

$$P_1 : \mathsf{prop}, \ldots, P_k : \mathsf{prop}, x_1 : \mathsf{ante}\ \ulcorner A_1 \urcorner, \ldots x_n : \mathsf{ante}\ \ulcorner A_n \urcorner \vdash_\Sigma \ulcorner \mathcal{D} \urcorner : \mathsf{succ}\ \ulcorner C \urcorner$$

Since atomic propositions may also be represented as constants in the signature, we'll ignore this detail and focus on the representation of proofs.

Let's start with the right rule for implication.

$$\mathcal{D} \quad = \quad \dfrac{\begin{array}{c} \mathcal{D}' \\ \Gamma, A \vdash B \end{array}}{\Gamma \vdash A \supset B}\ \supset R$$

Writing out the representation of the two sequents in LF:

$$\dfrac{\ulcorner \Gamma \urcorner, x : \mathsf{ante}\ \ulcorner A \urcorner \vdash \ulcorner \mathcal{D}' \urcorner : \mathsf{succ}\ \ulcorner B \urcorner}{\ulcorner \Gamma \urcorner \vdash \ulcorner \mathcal{D} \urcorner : \mathsf{succ}\ (\mathsf{imp}\ \ulcorner A \urcorner\ \ulcorner B \urcorner)}$$

We might conjecture we could achieve that if

$$\ulcorner \mathcal{D} \urcorner = \mathsf{impR}\ (\lambda x. \ulcorner \mathcal{D}' \urcorner)$$

so that

$$\begin{array}{l}\mathsf{impR} : (\mathsf{ante}\ \ulcorner A \urcorner \to \mathsf{succ}\ \ulcorner B \urcorner) \\ \qquad \to \mathsf{succ}\ (\mathsf{imp}\ \ulcorner A \urcorner\ \ulcorner B \urcorner)\end{array}$$

but we also need to abstract over the propositions A and B:

$$\begin{array}{l}\mathsf{impR} : \Pi A : \mathsf{prop}.\, \Pi B : \mathsf{prop}. \\ \qquad (\mathsf{ante}\ A \to \mathsf{succ}\ B) \\ \qquad \to \mathsf{succ}\ (\mathsf{imp}\ A\ B)\end{array}$$

We play through what happens if we focus on this on the left, writing $?A$ and $?B$ for an as yet unknown object of type prop and omitting contexts $\ulcorner \Gamma \urcorner$.

$$\dfrac{\begin{array}{c}\vdots \\ \vdash ?A : \mathsf{prop}\end{array} \qquad \dfrac{\begin{array}{c}\vdots \\ \vdash ?B : \mathsf{prop}\end{array} \qquad \dfrac{\dfrac{\dfrac{\mathsf{ante}\ ?A \vdash \langle \mathsf{succ}\ ?B \rangle}{\mathsf{ante}\ ?A \vdash \mathsf{succ}\ ?B}}{\vdash \mathsf{ante}\ ?A \to \mathsf{succ}\ ?B} \to R \quad \dfrac{\delta = \langle \mathsf{succ}\ (\mathsf{imp}\ ?A\ ?B) \rangle}{[\mathsf{succ}\ (\mathsf{imp}\ ?A\ ?B)] \vdash \delta}\ \mathsf{id}^-}{[(\mathsf{ante}\ ?A \to \mathsf{succ}\ ?B) \to \mathsf{succ}\ (\mathsf{imp}\ ?A\ ?B)] \vdash \delta} \to L}{[\Pi B{:}\mathsf{prop}.\, (\mathsf{ante}\ ?A \to \mathsf{succ}\ B) \to \mathsf{succ}\ (\mathsf{imp}\ ?A\ B)] \vdash \delta}\ \Pi L}{[\Pi A{:}\mathsf{prop}.\, \Pi B{:}\mathsf{prop}.\, (\mathsf{ante}\ A \to \mathsf{succ}\ B) \to \mathsf{succ}\ (\mathsf{imp}\ A\ B)] \vdash \delta}\ \Pi L$$

Due to the restriction on focusing with negative atoms, we see that focusing on impR can only succeed if the (metalevel) succedent has the form $\langle \mathsf{succ}\ (\mathsf{imp}\ \ulcorner A \urcorner\ \ulcorner B \urcorner) \rangle$ for some propositions A and B. Applying this throughout, we get the derived rule

$$\dfrac{\ulcorner \Gamma \urcorner, \mathsf{ante}\ \ulcorner A \urcorner \vdash \langle \mathsf{succ}\ \ulcorner B \urcorner \rangle}{\ulcorner \Gamma \urcorner \vdash \langle \mathsf{succ}\ (\mathsf{imp}\ \ulcorner A \urcorner\ \ulcorner B \urcorner) \rangle}$$

which is exactly what we were aiming for. If we add proof terms we get

$$\dfrac{\ulcorner \Gamma \urcorner, \mathsf{ante}\ \ulcorner A \urcorner \vdash M : \langle \mathsf{succ}\ \ulcorner B \urcorner \rangle}{\ulcorner \Gamma \urcorner \vdash (\mathsf{impR}\ \ulcorner A \urcorner\ \ulcorner B \urcorner\ (\lambda x.\, M(x))) : \langle \mathsf{succ}\ (\mathsf{imp}\ \ulcorner A \urcorner\ \ulcorner B \urcorner) \rangle}$$

Using focusing in this manner allows us to establish a bijection: If M is the representation of a proof $\mathcal{D}$, then $\mathsf{impR}\ \ulcorner A \urcorner\ \ulcorner B \urcorner\ (\lambda x.\, M(x))$ will also be one. Conversely, if $\ulcorner \Gamma \urcorner \vdash M : \langle \mathsf{succ}\ C \rangle$ then M must be the proof term for one of the inference rules encoded in the signature Σ. In each case, the same must be true for the unknown object in the premise(s), and we can systematically translate well-typed terms to sequent calculus proofs.

Let's look at the left rule for implication as one more example.

$$\dfrac{\Gamma \vdash A \quad \Gamma, B \vdash C}{\Gamma, A \supset B \vdash C}\ \supset L$$

We conjecture

$$\begin{array}{l}\mathsf{impL} : \Pi A : \mathsf{prop}.\, \Pi B : \mathsf{prop}.\, \Pi C : \mathsf{prop}. \\ \quad \mathsf{succ}\ A \to (\mathsf{ante}\ B \to \mathsf{succ}\ C) \\ \quad \to (\mathsf{ante}\ (\mathsf{imp}\ A\ B) \to \mathsf{succ}\ C)\end{array}$$

We won't go through the details of focusing, but what we arrive at is

$$\frac{\ulcorner\Gamma\urcorner \vdash M : \langle \mathsf{succ}\ \ulcorner A\urcorner\rangle \quad \ulcorner\Gamma\urcorner, y : \mathsf{ante}\ \ulcorner B\urcorner \vdash N(y) : \langle \mathsf{succ}\ \ulcorner C\urcorner\rangle}{\ulcorner\Gamma\urcorner, x : \mathsf{ante}\ \ulcorner A \supset B\urcorner \vdash \mathsf{impL}\ \ulcorner A\urcorner\ \ulcorner B\urcorner\ \ulcorner C\urcorner\ M\ (\lambda y.\, N(y))\ x : \langle \mathsf{succ}\ \ulcorner C\urcorner\rangle}$$

We see here that the encoding for a structural sequent calculus critically relies on the fact that the LF metalanguage is also structural. So there may be occurrences of x in M and N, just like the hypothesis $A \supset B$ can be used again in both premises of the sequent calculus rule.

You might also notice that modulo the explicit propositions and the argument order, this is exactly the proof term representation we used in Assignment 2 for ordered proofs. This is, of course, no coincidence. In practical implementations of logical frameworks such as LF, the arguments of type prop to the constructors can be omitted and will be determined by the implementation from context. In general, this problem is undecidable for LF and may require more information from the programmer, but in the vast majority of the cases the constraints on them are sufficient.

We can see how proof checking in an object logic (structural, for the moment) can be reduced to type-checking the proof representation in LF (also structural). Moreover, the representation is a bijection between proofs and the well-typed terms over a signature (the encoding of a proof system) and a particular context (the encoding of the antecedents).

Before moving on, let's look back at the type of impR.

$$\begin{array}{l}\mathsf{impR} : \Pi A : \mathsf{prop}.\, \Pi B : \mathsf{prop}. \\ \quad (\mathsf{ante}\ \ulcorner A\urcorner \to \mathsf{succ}\ \ulcorner B\urcorner) \\ \quad \to \mathsf{succ}\ (\mathsf{imp}\ \ulcorner A\urcorner\ \ulcorner B\urcorner)\end{array}$$

The adequacy of this encoding as mentioned in the preceding paragraph relies critically on the fact that the function spaces here are weak. For example, a function cannot examine the structure of its argument and base its result on it. Instead, an object of type $A \to B$ must be $\lambda x.\, M(x)$ which is parametric in x so that $M(N)$ is just the result of hereditary substitution of N for x in $M(x)$ without further computation.

3 A Linear Logical Framework

When encoding a substructural type system, the methodology we have exemplified in the preceding section no longer works for LF. That's because LF antecedents

of the form $x : \mathsf{ante}\ \ulcorner A \urcorner$ are structural, so we cannot use them for the linear antecedents of linear logic. The most direct solution is to generalize the framework so it also admits linear antecedents. We cannot throw out the nonlinear functions, so the framework will be a mixed linear/nonlinear type theory call Linear LF (or LLF, for short) [Cervesato and Pfenning, 1996, 2002].

Given what we know now regarding adjoint logic, we might have designed the framework differently. We start by just adding a linear function type, but leaving the remainder of the language the same; later on we'll need something one more type. We also just reuse λ-abstraction for linear functions and spines $(M\ ;\ S)$ for linear application since this ambiguity is not relevant here.

$$\begin{array}{lrcl} \text{Negative types} & A, B & ::= & P \mid A \to B \mid \Pi x : A.\, B(x) \mid A \multimap B \\ \text{Stable antecedents} & \Delta & ::= & \cdot \mid \Delta, x_{\mathsf{s}} : A \mid \Delta, x_{\mathsf{l}} : A \end{array}$$

Stable antecedents may be structural $x_{\mathsf{s}} : A$ or linear $x_{\mathsf{l}} : A$. All declarations in the signature remain structural: even in a linear logic, inference rules can be used multiple times.

Focusing works similar to the way it worked before—we highlight only two rules for the key differences between the linear and nonlinear left rules, namely that the first premise of $\to L$ can only depend on structural antecedents.

$$\frac{\Gamma_{\mathsf{s}} \vdash M : [A] \quad \Gamma_{\mathsf{s}}, \Delta', [B] \vdash S : \delta}{\Gamma_{\mathsf{s}}, \Delta', [A \to B] \vdash (M\ ;\ S) : \delta}\ \to L$$

$$\frac{\Gamma_{\mathsf{s}}, \Delta \vdash M : [A] \quad \Gamma_{\mathsf{s}}, \Delta', [B] \vdash (M\ ;\ S) : \delta}{\Gamma_{\mathsf{s}}, \Delta, \Delta', [A \multimap B] \vdash (M\ ;\ S) : \delta}\ \multimap L$$

The focus on A in the first premise of both of these rules will be lost immediately since the framework consists entirely of negative types. The succedent δ will always be a suspended negative atom $\langle P \rangle$, as for LF.

As our example we use the (purely linear) semi-axiomatic sequent calculus (SAX). We represent a proof

$$\begin{array}{c} \mathcal{D} \\ A_1, \ldots, A_n \vdash C \end{array}$$

by

$$x_{\mathsf{l}}^1 : \mathsf{ante}\ \ulcorner A_1 \urcorner, \ldots, x_{\mathsf{l}}^n : \mathsf{ante}\ \ulcorner A_n \urcorner \vdash \ulcorner \mathcal{D} \urcorner : \langle \mathsf{succ}\ \ulcorner C \urcorner \rangle$$

The right rule for linear implication parallels what we have seen in the structural case, but the left rule is replace by the axiom

$$\frac{}{A, A \multimap B \vdash B}\ \multimap X$$

which simplifies the representation slightly.

$\mathsf{lolli} : \mathsf{prop} \to \mathsf{prop} \to \mathsf{prop}$

$\mathsf{lolliR} : \Pi A : \mathsf{prop}.\, \Pi B : \mathsf{prop}.$
$\quad (\mathsf{ante}\ A \multimap \mathsf{succ}\ B)$
$\quad \multimap \mathsf{succ}\ (\mathsf{lolli}\ A\ B)$

$\mathsf{lolliX} : \Pi A : \mathsf{prop}.\, \Pi B : \mathsf{prop}.$
$\quad \mathsf{ante}\ A \multimap \mathsf{ante}\ (\mathsf{lolli}\ A\ B) \multimap \mathsf{succ}\ B$

The identity is similar to the axiom.

$\mathsf{id} : \Pi A : \mathsf{prop}.\, \mathsf{ante}\ A \multimap \mathsf{succ}\ A$

he fram The additive conjunction represents a new challenge.

$$\frac{\Delta \vdash A \quad \Delta \vdash B}{\Delta \vdash A \,\&\, B}\ \&R \qquad \frac{}{A \,\&\, B \vdash A}\ \&X_1 \quad \frac{}{A \,\&\, B \vdash B}\ \&X_2$$

The challenge here is how to "duplicate" the antecedents Δ to both premises of the $\&R$ rule. In the framework we have so far, there is not easy way to accomplish this. Fortunately, external choice is negative so we can just add it to the framework without disturbing much of its structure.

Negative types	A, B	$::=$	$P \mid A \to B \mid \Pi x : A.\, B(x) \mid A \multimap B \mid A \,\&\, B$
Objects	M	$::=$	$c\ S \mid x\ S \mid \lambda x.\, M(x) \mid (M_1, M_2)$
Spines	S	$::=$	$M\ ;\ S \mid (\,) \mid \pi_1\ ;\ S \mid \pi_2\ ;\ S$
Stable antecedents	Δ	$::=$	$\cdot \mid \Delta, x_{\mathsf{s}} : A \mid \Delta, x_{\mathsf{l}} : A$

With this addition we can define

$\mathsf{with} : \mathsf{prop} \to \mathsf{prop} \to \mathsf{prop}$

$\mathsf{withR} : \Pi A : \mathsf{prop}.\, \Pi B : \mathsf{prop}.$
$\quad (\mathsf{succ}\ A \,\&\, \mathsf{succ}\ B)$
$\quad \multimap \mathsf{succ}\ (\mathsf{with}\ A\ B)$

$\mathsf{withX}_1 : \Pi A : \mathsf{prop}.\, \Pi B : \mathsf{prop}.$
$\quad \mathsf{ante}\ (\mathsf{with}\ A\ B) \multimap \mathsf{succ}\ A$

$\mathsf{withX}_1 : \Pi A : \mathsf{prop}.\, \Pi B : \mathsf{prop}.$
$\quad \mathsf{ante}\ (\mathsf{with}\ A\ B) \multimap \mathsf{succ}\ B$

Because right inversion on $\mathsf{succ}\ A \,\&\, \mathsf{succ}\ B$ will propagate all linear antecedents to both premises, the translation of

$$\frac{\begin{array}{cc}\mathcal{D} & \mathcal{E} \\ \Delta \vdash A & \Delta \vdash B\end{array}}{\Delta \vdash A \,\&\, B}\ \&R$$

as

$$\mathsf{withR}\ \ulcorner A \urcorner\ \ulcorner B \urcorner\ (\ulcorner \mathcal{D} \urcorner, \ulcorner \mathcal{E} \urcorner) : \langle \mathsf{succ}\ (\mathsf{with}\ \ulcorner A \urcorner\ \ulcorner B \urcorner) \rangle$$

is adequately typed.

If we also encode cut

$$\frac{\Delta \vdash A \quad \Delta', A \vdash C}{\Delta, \Delta' \vdash C}\ \mathsf{cut}$$

as

$$\begin{array}{l}\mathsf{cut} : \Pi A : \mathsf{prop}.\ \Pi B : \mathsf{prop}. \\ \quad \mathsf{succ}\ A \multimap (\mathsf{ante}\ A \multimap \mathsf{succ}\ C) \\ \quad \multimap \mathsf{succ}\ C\end{array}$$

then we can, for example, show that the usual sequent calculus left rules are derivable in SAX. The derivation

$$\frac{\dfrac{}{A \,\&\, B \vdash A}\ \&X_1 \qquad \Delta, A \vdash C}{\Delta, A \,\&\, B \vdash C}\ \mathsf{cut}$$

becomes

$$\begin{array}{l}\vdash (\lambda f.\, \lambda y.\, \mathsf{cut}\ \ulcorner A \urcorner\ \ulcorner C \urcorner\ (\mathsf{with}\mathsf{X}_1\ \ulcorner A \urcorner\ \ulcorner B \urcorner\ y)\ (\lambda x. fx)) : (\mathsf{ante}\ \ulcorner A \urcorner \multimap \mathsf{succ}\ \ulcorner C \urcorner) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \multimap (\mathsf{ante}\ (\mathsf{with}\ \ulcorner A \urcorner\ \ulcorner B \urcorner) \multimap \mathsf{succ}\ \ulcorner C \urcorner)\end{array}$$

4 Metatheoretic Reasoning

One payoff for using high-level encodings is that they can enable elegant and concise metatheoretic reasoning [Schürmann, 2000]. For substructural logics, this is much less well understood and I believe Jason Reed's PhD Thesis 2009 using resource semantics is probably currently the high water mark, although more recent work with entirely different techniques is also promising [Sano et al., 2023, Crary, 2010].

Let's first consider the structural case, and let's assume we have formalized the sequent calculus *without the cut rule*. Then, at some informal level, the (constructive!) admissibility proof for cut would correspond to a function from a proof of A and a proof of C from A to a proof of C. In LF, this might be written as

$$\begin{array}{l}\mathsf{cutadmit} : \Pi A : \mathsf{prop}.\ \Pi C : \mathsf{prop}. \\ \qquad \mathsf{succ}\ A \to (\mathsf{ante}\ A \to \mathsf{succ}\ C) \to \mathsf{succ}\ C\end{array}$$

Unfortunately, such a function is not representable in LF because it would have to distinguish all the different cases for the proofs of succ A and the hypothetical proof of ante $A \to$ succ C. Allowing such case distinction would destroy the adequacy of the encoding, although there are systems such as $\mathcal{M}_2^+$ [Schürmann, 2000] and

Beluga [Pientka and Cave, 2015] that support multiple different kinds of function spaces. In Twelf [Pfenning and Schürmann, 1999], the solution is to represent the metatheoretic proof instead as a *relation*.

$$\begin{aligned}\mathsf{cutadmit} : \Pi A : \mathsf{prop}.\, \Pi C : \mathsf{prop}.\\ \mathsf{succ}\ A \to (\mathsf{ante}\ A \to \mathsf{succ}\ C) \to \mathsf{succ}\ C\\ \to \mathbf{type}\end{aligned}$$

We can then check properties of this relation to verify our theorem. Specifically, it should be total in A, C and the two given derivations. You can read more about this in the following two papers [Pfenning, 1995, 2000].

It turns out this relational method is quite general, and there are many examples and case studies in the Twelf distribution and on the website.[1]

References

Iliano Cervesato and Frank Pfenning. A linear logical framework. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, pages 264–275, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information & Computation*, 179(1):19–75, November 2002. Revised and expanded version of an extended abstract, LICS 1996, pp. 264-275.

Karl Crary. Higher-order representation of substructural logics. In P.Hudak and S.Weirich, editors, *Proceedings of the 15th International Conference on Functional Programming (ICFP 2010)*, pages 131–142, Baltimore, Maryland, September 2010. ACM.

Frank Pfenning. Structural cut elimination. In D. Kozen, editor, *Proceedings of the Tenth Annual Symposium on Logic in Computer Science*, pages 156–166, San Diego, California, June 1995. IEEE Computer Society Press.

Frank Pfenning. Structural cut elimination I. Intuitionistic and classical logic. *Information and Computation*, 157(1/2):84–141, March 2000.

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, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[1] http://twelf.org/

Brigitte Pientka and Andrew Cave. Inductive Beluga: Programming proofs. In A. Felty and A. Middeldorp, editors, *25th International Conference on Automated Deduction (CADE 2015)*, pages 272–281, Berlin, Germany, August 2015. Springer LNCS 9195.

Jason C. Reed. *A Hybrid Logical Framework*. PhD thesis, Carnegie Mellon University, September 2009. Available as Technical Report CMU-CS-09-155.

Chuta Sano, Ryan Kavanagh, and Brigitte Pientka. Mechanizing session-types using a structural view: Enforcing linearity without linearity. In *Proceedings of the ACM on Programming Languages*, volume 7 (OOPSLA2), pages 374–399. ACM, 2023. Extended version available at https://arxiv.org/abs/2309.12466.

Carsten Schürmann. *Automating the Meta Theory of Deductive Systems*. PhD thesis, Department of Computer Science, Carnegie Mellon University, August 2000. Available as Technical Report CMU-CS-00-146.

Lecture Notes on
The Concurrent Logical Framework

15-836: Substructural Logics
Frank Pfenning

Lecture 22
December 5, 2023

1 Introduction

In the last lecture we introduced Linear LF (LLF) as a substructural framework and we saw how to represents proofs in the linear semi-axiomatic sequent calculus. But it turns out that even some very simple systems of linear inference are difficult to represent, particularly those with multiple conclusions that we used at the beginning of the course.

In order to handle these we carefully extend the logical framework with some positive types, leading us to Concurrent LF (CLF) [Watkins et al., 2002, Cervesato et al., 2002, Watkins et al., 2004] which is implemented in the Celf language [Schack-Nielsen and Schürmann, 2008, Schack-Nielsen, 2011][1]. It turns out that this will allow us to capture some *concurrency* in the computations we represent. We then encode the dynamics of futures as an example that is significantly more direct than possible in LLF.

2 Coin Exchange Revisited

Recall the rules for a linear coin exchange from Lecture 1, where q is a quarter, d is a dime, and n is a nickel.

$$\frac{q}{d \quad d \quad n}\ \mathsf{fromQ} \qquad \frac{d \quad d \quad n}{q}\ \mathsf{toQ} \qquad \frac{d}{n \quad n}\ \mathsf{fromD} \qquad \frac{n \quad n}{d}\ \mathsf{toD}$$

Here is a simple proof that from a quarter and a nickel we can get three dimes.

$$\dfrac{\dfrac{q}{d \quad d}\ \mathsf{fromQ} \qquad \dfrac{n \qquad n}{d}\ \mathsf{toD}}{}$$

[1] https://github.com/clf/celf

In linear logic, we can internalize these as (structural!) propositions

$q \multimap (d \otimes d \otimes n)$
$(d \otimes d \otimes n) \multimap q$
$d \multimap (n \otimes n)$
$(n \otimes n) \multimap d$

We could make the second and the forth into something entirely negative by Currying and represent it in LLF, but not the other two.

$\mathsf{fromQ} : q \multimap (d \otimes d \otimes n)$??
$\mathsf{toQ} : d \multimap d \multimap n \multimap q$
$\mathsf{fromD} : d \multimap (n \otimes n)$??
$\mathsf{toD} : n \multimap n \multimap d$

How can we solve this problem and represent this form of inference in LLF? Think about it before you move on, because you actually have seen the technique we need already (specifically in Lecture 2).

The technique is to move the inference steps into the *antecedents*, also flipping their direction. The rules the are (with some arbitrary succedent C):

$$\frac{\Delta, d, d, n \vdash C}{\Delta, q \vdash C}\ \mathsf{fromQ} \qquad \frac{\Delta, q \vdash C}{\Delta, d, d, n \vdash C}\ \mathsf{toQ}$$

$$\frac{\Delta, n, n \vdash C}{\Delta, d \vdash C}\ \mathsf{fromD} \qquad \frac{\Delta, d \vdash C}{\Delta, n, n \vdash C}\ \mathsf{toD}$$

This we can represent in LLF, thinking of q, d, n, and C as judgments (and writing c in lowercase), rather than propositions to avoid an extra level of indirection.

$$\begin{aligned}
&\mathsf{fromQ} : (d \multimap d \multimap n \multimap c) \\
&\qquad\qquad \multimap (q \multimap c) \\
&\mathsf{toQ} : \quad (q \multimap c) \\
&\qquad\qquad \multimap (d \multimap d \multimap n \multimap c) \\
&\mathsf{fromD} : (n \multimap n \multimap c) \\
&\qquad\qquad \multimap (d \multimap c) \\
&\mathsf{toD} : \quad (d \multimap c) \\
&\qquad\qquad \multimap (n \multimap n \multimap c)
\end{aligned}$$

In functional programming this technique could be called *continuation-passing style* where c stands for the continuation.

Now if we want to show that we can get three dimes from a quarter and a nickel, it would be represented as

$$\vdash (d \multimap d \multimap d \multimap c) \multimap (q \multimap c)$$

which we prove as follows:

$$\dfrac{\dfrac{\dfrac{\vdots}{d \multimap d \multimap d \multimap c, d, d, d \vdash c}}{d \multimap d \multimap d \multimap c, d, d, n, n \vdash c}\ \mathsf{toD}}{d \multimap d \multimap d \multimap c, d, d, n, n \vdash c} = \qquad \dfrac{}{d \multimap d \multimap d \multimap c, q, n \vdash c}\ \mathsf{fromQ}}{\vdash (d \multimap d \multimap d \multimap c) \multimap (q \multimap n \multimap c)}\ \multimap R \times 3$$

The omitted part of the proof above is entirely straightforward.

If we represent this derivation as a term in LLF, it would be

$$\vdash \lambda f.\, \lambda q_1.\, \lambda n_1.\, \mathsf{fromQ}\ (\lambda d_1.\, \lambda d_2.\, \lambda n_2.\, \mathsf{toD}\ (\lambda d_3.\, f\ d_1\ d_2\ d_3)\ n_1\ n_2)\ q_1$$

where, as before, we write $h\ (M_1\ ;\ \ldots M_k)$ as $h\ M_1\ \ldots\ M_k$.

There are few notes about this particular term representation. One is that the coins have unique identities. For example, if we swap the arguments to f as in $f\ d_3\ d_1\ d_2$ we obtain a different term. In order to avoid this and reduce the number of possible proofs, we can use *proof irrelevance* [Ley-Wild and Pfenning, 2007].

The other is that no matter what coins and exchange opportunities we have, the proof term will always consist of nested constructors. One possible answer to this is *multifocusing* [Chaudhuri et al., 2008]. Another is to explicitly construct a logical framework with positive connectives, as we'll do now.

3 CLF

The Concurrent Logical Framework was explicitly designed to allow natural encodings of linear forward inference and also the kind of concurrency from the previous example. We only give here a very brief description before we start using it—you are referred to the technical reports and the implementation mentioned in the introduction to the lecture for more information.

Below we have in red the LLF additions to LF and in blue the further additions that CLF makes.

Negative types	A, B	$::=$	$P \mid A \to B \mid \Pi x : A.\, B(x) \mid A \multimap B \mid A \mathbin{\&} B \mid \{A^+\}$
Positive types	A^+, B^+	$::=$	$\mathbf{1} \mid A^+ \otimes B^+ \mid \exists x : A.\, B^+(x) \mid A$
Objects	M	$::=$	$c\ S \mid x\ S \mid \lambda x.\, M(x) \mid (M_1, M_2) \mid \{E\}$
Spines	S	$::=$	$M\ ;\ S \mid (\,) \mid \pi_1\ ;\ S \mid \pi_2\ ;\ S$
Expressions	E	$::=$	$\ldots$
Stable antecedents	Δ	$::=$	$\cdot \mid \Delta, x_{\mathsf{s}} : A \mid \Delta, x_{\mathsf{l}} : A$

There are a number of things to note here. Expressions E are the objects of positive type, yet to be specified. The positive types are included in the negative ones as $\{A^+\}$ which, nowadays, we would recognize as a form of shift. Similarly, the negative types are included directly in the positive ones (again, that should probably be via a shift). Also, there are no positive atoms, even though there should be because at the time we designed CLF we didn't understand the type theory well enough. Also, the antecedent A of $A \to B$, $A \multimap B$, and $\Pi x : A.\, B(x)$ should be positive, and probably also the A in $\exists x : A.\, B(x)$. This last set of issues was recognized and repaired in the design of Celf.

We omitted sums $A \oplus B$ because the branching nature of expressions complicated the form of equality on terms we wanted to allow; a few further remarks later. Before we come to the extension of objects/expressions, let's write some types to illustrate the use of positive types in the encoding of the coin exchange.

$\mathsf{fromQ} : q \multimap \{\, d \otimes d \otimes n \,\}$
$\mathsf{toQ} : d \multimap d \multimap n \multimap \{\, q \,\}$

$\mathsf{fromD} : d \multimap \{\, n \otimes n \,\}$
$\mathsf{toD} : n \multimap n \multimap \{\, d \,\}$

here, we need to wrap even the singletons in the conclusions of the implications because otherwise these clauses would be eligible for backchaining and not for forward chaining.

Now to the proof terms for forward chaining. Since we do not have expressions are just let bindings instead of general matches.

$$\begin{array}{lrcl} \text{Expressions} & E & ::= & \mathbf{let}\ \{p\} = M\ \mathbf{in}\ E \mid T \\ \text{Terms} & T & ::= & [\,] \mid [T_1, T_2] \mid [M, T] \mid M \\ \text{Patterns} & p & ::= & [\,] \mid [p_1, p_2] \mid [x, p] \mid x \end{array}$$

We now show the CLF signature including the proof term for the previous example.

```
1  q : type.
2  d : type.
3  n : type.
4
5  toQ : d -o d -o n -o { q }.
6  fromQ : q -o { d * d * n }.
7  toD : n -o n -o { d }.
8  fromD : d -o { n * n }.
9
10 ex1 : q -o n -o { d * d * d } =
11 \q1. \n1. { let {[d1, [d2, n2]]} = fromQ q1 in
12             let { d3 } = toD n1 n2 in
13             [d1, d2, d3] }.
```

In this particular example, fromQ must come before toD because the argument n_2 to toD is bound in the pattern that is matched against the result of fromQ.

More generally, though we consider two let expressions to be equivalent if they can be swapped without any variable capture. That is,

$$\begin{array}{rl} & (\mathbf{let}\ p = M\ \mathbf{in}\ \mathbf{let}\ q = N\ \mathbf{in}\ T) \\ = & (\mathbf{let}\ q = N\ \mathbf{in}\ \mathbf{let}\ p = M\ \mathbf{in}\ T) \\ & \text{provided } \mathsf{FV}(p) \cap \mathsf{FV}(N) = \emptyset = \mathsf{FV}(q) \cap \mathsf{FV}(M) \end{array}$$

This equality is baked into the definition of CLF at a fundamental level, just like the renaming of bound variables. This allows is to think of the expressions as capturing "*true concurrency*", that is, different independent interleavings of concurrent actions are indistinguishable.

It might be interesting to explore whether the notion of CBA-graphs we sketched in the first two lectures would be an abstract representation of the quivalence classes that arise from commuting independent actions. This may be related to the notion of multifocusing mentioned earlier.

4 Representing the Dynamics of Futures

Without further theory, which can be found in the given references, we show an encoding of the dynamics of the positive fragment of linear SAX, which gives us linear futures. This can easily be extended to encompass the whole language and is much more abstract than the implementation in SML we used in this course.

```
val : type.
exp : type.
cont : type.

addr : type.

unit : val.   % 1
pi1 : addr -o val. % A + B
pi2 : addr -o val.
pair : addr -o addr -o val. % A * B

unit_cont : exp -o cont.
plus_cont : (addr -o exp) & (addr -o exp) -o cont.
pair_cont : (addr -o addr -o exp) -o cont.

cut : (addr -o exp) -o (addr -o exp) -o exp.
id : addr -o addr -o exp.
write : addr -o val -o exp.
read : addr -o cont -o exp.

cell : addr -> val -> type.
proc : exp -> type.

pass : val -> cont -> exp -> type.
pass/unit : pass unit (unit_cont P) P.
pass/plus1 : pass (pi1 A) (plus_cont <(\x. P x), (\y. Q y)>) (P A).
pass/plus2 : pass (pi2 B) (plus_cont <(\x. P x), (\y. Q y)>) (Q B).
pass/pair : pass (pair A B) (pair_cont (\x. \y. P x y)) (P A B).

exec/cut : proc (cut (\x. P x) (\x. Q x))
        -o { Exists a:addr. proc (P a) * proc (Q a) }.
exec/id : proc (id A B) -o cell B V -o { cell A V }.
exec/write : proc (write A V) -o { cell A V }.
exec/read : proc (read A K) -o cell A V -o pass V K P
          -o { proc P }.
```

There are some subtle points here, such as the use of external choice in the encoding of `plus_cont`, but in most respects it is an entirely straightforward representation. Note also the use of the existential to create a fresh address dynamically when executing a cut.

Unlike the coin exchange, we can actually execute SAX programs in this en-

coding because of the don't care nondeterminism that underlies the dynamics. In the coin exchange, we never reach quiescence, but here we do for terminating programs using futures.

In the first two examples we execute a SAX program for negation of a boolean value represented by store with two cells.

```
#query * 1 * 1
Pi c0:addr. Pi c1:addr. Pi c2:addr.
    cell c0 unit * cell c1 (pi1 c0)
    * proc (read c1 (plus_cont (<(\u. write c2 (pi2 u)),
                                (\u. write c2 (pi1 u))>)))
    -o { cell c0 unit * cell c2 (pi2 c0) }.

#query * 1 * 1
Pi c0:addr. Pi c1:addr. Pi c2:addr.
    cell c0 unit * cell c1 (pi2 c0)
    * proc (read c1 (plus_cont (<(\u. write c2 (pi2 u)),
                                (\u. write c2 (pi1 u))>)))
    -o { cell c0 unit * cell c2 (pi1 c0) }.
```

The implementation will print a trace of the computations in the form of terms of the given type. It shows structural bindings with `\!x. M`, while linear bindings are just `\x. M`.

```
Query (*, 1, *, 1) ...
Solution: \!c0. \!c1. \!c2. \[X1, [X2, X3]]. {
    let {X4} = exec/read X3 X2 pass/plus1 in
    let {X5} = exec/write X4 in [X1, X5]}
Query ok.

Query (*, 1, *, 1) ...
Solution: \!c0. \!c1. \!c2. \[X1, [X2, X3]]. {
    let {X4} = exec/read X3 X2 pass/plus2 in
    let {X5} = exec/write X4 in [X1, X5]}
Query ok.
```

In the last query we use an existential quantifier in the succedent so as not to anticipate the answer and let CLF's forward inference compute it for us.

```
#query * 1 * 1
    Pi c0:addr. Pi c1:addr. Pi c2:addr.
    cell c0 unit * cell c1 (pi2 c0)
    * proc (read c1 (plus_cont (<(\u. write c2 (pi2 u)),
                                (\u. write c2 (pi1 u))>)))
    -o { Exists V0. Exists V2. cell c0 V0 * cell c2 V2 }.
```

Here, the answers `!unit` and `!(pi1 c0)` are presented in the results as witness for the existentials. The exclamation mark shows that they are not linear.

```
Query (*, 1, *, 1) ..
```

```
Solution: \!c0. \!c1. \!c2. \[X1, [X2, X3]]. {
    let {X4} = exec/read X3 X2 pass/plus2 in
    let {X5} = exec/write X4 in [!unit, [!(pi1 c0), [X1, X5]]]}
Query ok.
```

References

Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

Kaustuv Chaudhuri, Dale Miller, and Alexis Saurin. Canonical sequent proofs via multi-focusing. In *5th International Conference on Theoretical Computer Science*, pages 383–396, Milano, Italy, September 2008. IFIPAICT 273.

Ruy Ley-Wild and Frank Pfenning. Avoiding causal dependencies via proof irrelevance in a concurrent logical framework. Technical Report CMU-CS-07-107, Carnegie Mellon University, February 2007.

Anders Schack-Nielsen. *Implementing Substructural Logical Frameworks*. PhD thesis, IT University of Copenhagen, January 2011.

Anders Schack-Nielsen and Carsten Schürmann. Celf - a logical framework for deductive and concurrent systems. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning (IJCAR'08)*, pages 320–326, Sydney, Australia, August 2008. Springer LNCS 5195.

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework: The propositional fragment. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs*, pages 355–377. Springer-Verlag LNCS 3085, 2004. Revised selected papers from the *Third International Workshop on Types for Proofs and Programs*, Torino, Italy, April 2003.

Lecture Notes on Linear Natural Deduction

15-836: Substructural Logics
Sophia Roshal

Lecture 23
December 7, 2023

1 Introduction

Throughout the course we have been focusing on the sequent calculus and its variations SAX [DeYoung et al., 2020] and SNAX [DeYoung and Pfenning, 2022] all of which follow a bottom up reasoning method. In this lecture we introduce *natural deduction* [Gentzen, 1935] which instead has 2 directions of reasoning, with introduction rules which follow bottom up reasoning, and elimination rules which follow top down reasoning. Using all the same tools as we have used so far, we will develop these rules, and prove some important properties as well as relate natural deduction back to the sequent calculus.

2 The Base Rules

As stated in the introduction, we have two forms of rules. Introduction rules (which we think of reasoning upwards) correspond directly to the sequent calculus right rules. The elimination rules (which we think of reasoning downwards) will need a bit more work. All the rules can be found in the appendix, but in this section we will focus on one positive ($\otimes$) and one negative ($\multimap$) connective. First, for completeness sake, we state the introduction rule for $\otimes$.

$$\frac{\Delta_1 \vdash M : A \quad \Delta_2 \vdash N : B}{\Delta_1, \Delta_2 \vdash (M, N) : (A \otimes B)} \otimes I$$

Now, we think about the elimination form. We should be starting, in our premise, with

$$\frac{\Delta_1 \vdash M : (A \otimes B)}{} \otimes E \text{ incomplete}$$

Since we are in a linear setting, we are required to somehow use both A and B, however, we aren't allowed to have multiple conclusions in our rules, so, similarly to the sequent calculus, we will break apart $A \otimes B$ into its parts, and use these parts to prove some new right hand side term C producing the following rule:

$$\frac{\Delta_1 \vdash M : A \otimes B \quad \Delta_2, x : A, y : B \vdash N : C}{\Delta_1, \Delta_2 \vdash \text{match } M \text{ with } (x, y) \text{ in } N : C} \otimes E$$

For $\multimap$, again the introduction rule is identical to the sequent calculus right rule.

$$\frac{\Delta, x : A \vdash M : B}{\Delta \vdash \lambda x.M : A \multimap B} \multimap I$$

For the elimination rule, we start with

$$\frac{\Delta_1 \vdash M : A \multimap B}{} \multimap E \text{ partial}$$

Thinking about linearity, we know we need to consume M, and approaching this from programming perspective, $\multimap$ is a the type of a function. To consume a function, we apply it, so this rule should correspond to function application.

$$\frac{\Delta_1 \vdash M : A \multimap B \quad \Delta_2 \vdash N : A}{\Delta_1, \Delta_2 \vdash M\ N : B} \multimap E$$

Other rules can be constructed in a similar fashion, with positive connectives corresponding almost directly to sequent calculus rules, and negative connectives requiring a bit more work.

3 Bidirectional Type Checking

In section 2, we presented some rules, and claimed that we read the introduction rules bottom up, as in the sequent calculus, and the elimination rules top down, but this isn't made in any way explicit. We can make this explicit via bidirectional type checking [Dunfield and Krishnaswami, 2022] by splitting the judgement $M : A$ into two judgements $M \Leftarrow C$ (read as M checks against C) and $M \Rightarrow A$ (read as M synthesizes A.) From an implementation perspective, we think of the checking judgement as type checking (where both M and C are provided as inputs) and the synthesis judgement as type inference (where only the term M is given, and we infer its type). The checking judgement is the upward reasoning direction, while the synthesis judgement is the downwards reasoning direction. We start with $\otimes$. The introduction rule is fairly straightforward. To check if the pair (M, N) has the type $A \otimes B$, we need to check the individual components.

$$\frac{\Delta_1 \vdash M \Leftarrow A \quad \Delta_2 \vdash N \Leftarrow B}{\Delta_1, \Delta_2 \vdash (M, N) \Leftarrow A \otimes B} \otimes I$$

The elimination rule is a bit trickier.

$$\frac{\Delta_1 \vdash M \Rightarrow (A \otimes B) \quad \Delta_2, x : A, y : B \vdash N\ ?\ C}{\Delta_1, \Delta_2 \vdash \text{match } M \text{ with } (x, y) \text{ in } N\ ?\ C} \otimes E \text{ incomplete}$$

The first premise we have labeled as a synthesis, this is to embody the downward direction reasoning. To label the other two, there are several ways to do so that would be "correct" in the sense that we would still have a complete system with respect to standard natural

deduction. One way to decide, is to consider the sequent calculus rule, and label everything that appears on the left as a synthesis judgement, and everything that appears on the right as a checking judgement. Internally to natural deduction, we can come to the same conclusion for this rule, by thinking about the order that rules apply in. We first want to apply all the introduction rules we can before we begin apply elimination rules. Once we have reached the point of applying the $\otimes$ elimination rule, we are still in a checking mode, and from this rule we don't have information about what C (or N) is, so we might as well remain in a checking mode. This gives us the following rule

$$\frac{\Delta_1 \vdash M \Rightarrow (A \otimes B) \quad \Delta_2, x : A, y : B \vdash N \Leftarrow C}{\Delta_1, \Delta_2 \vdash \text{match } M \text{ with } (x, y) \text{ in } N \Leftarrow C} \otimes E$$

Now for $\multimap$. Again, the introduction rule is fairly straightforward.

$$\frac{\Delta, x : A \vdash M \Leftarrow B}{\Delta \vdash \lambda x.M \Leftarrow A \multimap B} \multimap I$$

For the elimination rule, we again think back to the sequent calculus, and which propositions appear on which side in the implication left rule. If we follow that, we are left with

$$\frac{\Delta_1 \vdash M \Rightarrow A \multimap B \quad \Delta_2 \vdash N \Leftarrow A}{\Delta_1, \Delta_2 \vdash M\ N \Rightarrow B} \multimap E$$

Internally to natural deduction, and thinking about the proof terms, we could come to the same conclusion. Starting from the top, we know $A \multimap B$ should be a synthesis as that's the term we are applying the elimination rule to. Once we have that type, we now have A as well so the second premise can be a checking judgement. Finally, for the conclusion, having it be a checking judgement would not be helpful to the proof. We'd still need to synthesize a type for M, so this should remain as a synthesis.

Lastly we also present the hypothesis rule.

$$\frac{}{x : A \vdash x \Rightarrow A}\ \mathsf{hyp}(x)$$

From the separation into upwards and downwards reasoning point of view, it doesn't make sense to make this an upwards rule as there is no upward direction to go. From a programming perspective, we want this rule to be a synthesis, as we don't want to have to annotate our variables with a type if it isn't necessary to do so.

While this looks like the identity rule of the sequent calculus, it has a different function and purpose (as we'll see later in this lecture), so we call it the *hypothesis* rule.

4 An example

Here, we prove one direction of currying in our bidirectional system. Through this derivation, we will see that on top of the base rules, we actually need one more.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f : (A \multimap (B \multimap C)) \vdash f \Rightarrow (A \multimap (B \multimap C))}\,\mathsf{hyp}(f) \quad \dfrac{\dfrac{}{a : A \vdash a \Rightarrow A}\,\mathsf{hyp}(a)}{a : A \vdash a \Leftarrow A}\,?}{f : (A \multimap (B \multimap C)), a : A \vdash f \Rightarrow (A \multimap (B \multimap C))}\multimap E \quad \dfrac{\dfrac{}{b : B \vdash b \Rightarrow B}\,\mathsf{hyp}(b)}{b : B \vdash b \Leftarrow B}\,?}{f : (A \multimap (B \multimap C)), a : A, b : B \vdash ((f\ a)\ b) \Rightarrow C}\multimap E \quad \dfrac{}{ab : (A \otimes B) \vdash ab \Longrightarrow A \otimes B}\,\mathsf{hyp}(ab)}{f : (A \multimap (B \multimap C)), a : A, b : B \vdash ((f\ a)\ b) \Leftarrow C}\,? \quad \dfrac{}{ab : (A \otimes B) \vdash ab \Longrightarrow A \otimes B}\,\mathsf{hyp}(ab)}{f : (A \multimap (B \multimap C)), ab : A \otimes B \vdash \text{match } ab \text{ with } (a, b) \text{ in } ((f\ a)\ b) \Leftarrow C}\otimes E$$

$$\dfrac{f : (A \multimap (B \multimap C)), ab : A \otimes B \vdash \text{match } ab \text{ with } (a, b) \text{ in } ((f\ a)\ b) \Leftarrow C}{f : (A \multimap (B \multimap C)) \vdash \lambda ab.\text{match } ab \text{ with } (a, b) \text{ in } ((f\ a)\ b) \Leftarrow (A \otimes B) \multimap C}\multimap I$$

$$\dfrac{f : (A \multimap (B \multimap C)) \vdash \lambda ab.\text{match } ab \text{ with } (a, b) \text{ in } ((f\ a)\ b) \Leftarrow (A \otimes B) \multimap C}{\cdot \vdash \lambda f.\lambda ab.\text{match } ab \text{ with } (a, b) \text{ in } ((f\ a)\ b) \Leftarrow (A \multimap (B \multimap C)) \multimap ((A \otimes B) \multimap C)}\multimap I$$

There are a few places in this proof marked with a question mark, which leads us to one more rule:

$$\dfrac{\Delta \vdash M \Rightarrow A}{\Delta \vdash M \Leftarrow A}\ \Rightarrow/\Leftarrow^*$$

From an implementation perspective however, we need to modify this rule just a bit. Synthesis only takes the term as input so this rule doesn't entirely make sense as written, given that M synthesizes some type, and we don't necessarily know that it will be A specifically. This is something we need to check. We make a small modification.

$$\dfrac{\Delta \vdash M \Rightarrow A' \quad A = A'}{\Delta \vdash M \Leftarrow A}\ \Rightarrow/\Leftarrow$$

While we've written $A = A'$ for type equality here, we could also generalize this to be a sub-typing call, if our type system supports sub-typing. This one rule is the only place where we need to do type equality/sub-typing checks.

5 System Correctness

So far, we have completed an example proof in natural deduction that we knew should work. Unfortunately (or fortunately) proof by one example is not usually an accepted form of proof in most publication venues, and so we need to do a bit more work before being satisfied that the system we have developed is correct. There are several criteria we can consider when we want to decide whether a system we have is "correct". For the sequent calculus, we had an internal way to do so via admissibility of identity and cut which established harmony between the left and right rules. In natural deduction, we also have a similar internal notion via local soundness and completeness of the introduction and elimination rules. Another way we can establish correctness is by relating to some outside system. Throughout this course, we have established the sequent calculus as "the source of truth" and have proven many properties about it, so it would make sense to want to verify that natural deduction corresponds in some way to the sequent calculus.

5.1 Harmony

We have two properties to prove: local soundness (or proof normalization) which corresponds to cut elimination/admissibility in the sequent calculus and local completeness which corresponds to admissibility of identity in the sequent calculus. We start with local completeness. Here, we want to prove that we given an arbitrary proof of $\Delta \vdash M : A$, and applying the corresponding elimination rule, we should be able to reconstruct A again (of course, with a different proof term).

Proof: Case: $\otimes$

$$\overset{\mathcal{D}}{\Delta \vdash M : A \otimes B} \Longrightarrow \dfrac{\overset{\mathcal{D}}{\Delta \vdash M : A \otimes B} \quad \dfrac{\dfrac{}{x : A \vdash x : A}\,\mathsf{hyp} \quad \dfrac{}{y : B \vdash y : B}\,\mathsf{hyp}}{x : A, y : B \vdash (x, y) : (A \otimes B)}\otimes I}{\Delta \vdash \text{match } M \text{ with } (x, y) \text{ in } (x, y) : (A \otimes B)}\otimes E$$

Case: $\multimap$

$$\overset{\mathcal{D}}{\Delta \vdash M : A \multimap B} \Longrightarrow \dfrac{\dfrac{\overset{\mathcal{D}}{\Delta \vdash M : A \multimap B} \quad \dfrac{}{x : A \vdash x : A}\,\mathsf{hyp}}{\Delta, x : A \vdash M\ x : B}\multimap E}{\Delta \vdash \lambda x.(M\ x) : A \multimap B}\multimap I$$

Other cases proceed similarly. □

Soundness requires a bit more work. We want to establish that if we apply an introduction rule then an elimination rule, we can actually simplify that proof. this corresponds to cut elimination (and more specifically the principal case of cut elimination with a right rule "introducing" the cut proposition and the left rule "eliminating" it). To do so, we need one more lemma: substitution. This corresponds to the more general cut reduction, where we may not be cutting together a right and left rule. We need to prove that given a derivation that relies on a variable, we can substitute that variable for a term of the same type.

Lemma 1 *The following rule is admissible in the system without it*

$$\dfrac{\Delta \vdash M : A \quad \Delta', x : A \vdash N(x) : C}{\Delta, \Delta' \vdash N(M) : C}\ subst$$

Proof: Proof proceeds by induction on the second given derivation. We provide an interesting case

Case: the second derivation ends in a $\multimap E$. This case actually splits into two possible cases, we show just the first (the second case is almost identical, with the difference of how the context is split).

$$\dfrac{\Delta \vdash M : A \quad \dfrac{\overset{\mathcal{D}_1}{\Delta_1', x : A \vdash N(x) : B \multimap C} \quad \overset{\mathcal{D}_2}{\Delta_2' \vdash N'(x) : B}}{\Delta_1', \Delta_2', x : A \vdash N(x)\ N'(x) : C}\multimap E}{\Delta, \Delta_1', \Delta_2' \vdash N(M)\ N'(M) : C}\ subst$$

First, we need to realize that x cannot appear in N' as it must appear in N due to linearity (and x is not present in Δ_2'), so $N'(x) = N'$ which in turn gives us $N'(M) = N'$. We can now push the substitution upwards, reducing this proof to the following:

$$\dfrac{\dfrac{\Delta \vdash M : A \quad \overset{\mathcal{D}_2}{\Delta_1', x : A \vdash N(x) : B \multimap C}}{\Delta, \Delta_1' \vdash N(M) : B \multimap C}\ subst \qquad \overset{\mathcal{D}_2}{\Delta_2' \vdash N' : B}}{\Delta, \Delta_1', \Delta_2' \vdash N(M)\ N' : C} \multimap E$$

Other cases proceed similarly. □

Moving on to proving local soundness.

Proof: Case: $\otimes$

$$\dfrac{\dfrac{\overset{\mathcal{D}_1}{\Delta_a \vdash M : A} \quad \overset{\mathcal{D}_2}{\Delta_b \vdash M' : B}}{\Delta_a, \Delta_b \vdash (M, M') : A \otimes B} \otimes I \qquad \overset{\mathcal{E}}{\Delta', x : A, y : B \vdash N : C}}{\Delta_a, \Delta_b, \Delta' \vdash \text{match } (M, M') \text{ with } (x, y) \text{ in } N : \text{C}} \otimes E$$

We know N relies on both x and y, and this is where substitution comes into play. We construct the following proof reduction via admissibility of substitution:

$$\dfrac{\overset{\mathcal{D}_1}{\Delta_a \vdash M : A} \qquad \dfrac{\overset{\mathcal{D}_2}{\Delta_b \vdash M' : B} \quad \overset{\mathcal{E}}{\Delta', x : A, y : B \vdash N(x, y) : C}}{\Delta_b, \Delta', x : A \vdash N(x, M') : C}\ subst}{\Delta_a, \Delta_b, \Delta' \vdash N(M, M') : C}\ subst$$

Case: $\multimap$

$$\dfrac{\dfrac{\overset{\mathcal{D}}{\Delta_1, x : A \vdash M : B}}{\Delta_1 \vdash \lambda x.M : A \multimap B} \multimap I \qquad \overset{\mathcal{E}}{\Delta_2 \vdash N : A}}{\Delta_1, \Delta_2 \vdash (\lambda x.M)N : B} \multimap E \implies$$

$$\dfrac{\overset{\mathcal{E}}{\Delta_2 \vdash N : A} \quad \overset{\mathcal{D}}{\Delta_1, x : A \vdash M(x) : B}}{\Delta_1, \Delta_2 \vdash M(N) : B}\ subst$$

other cases proceed similarly. □

5.2 Soundness/Completeness wrt Sequent Calculus

We have developed two natural deduction systems. We would like to be able to relate them to each other as well as to the sequent calculus, and to do so we can prove three theorems. To separate notation, we will use $\overset{nd}{\vdash}$ to represent derivations in standard natural deduction, $\overset{\uparrow\downarrow}{\vdash}$ to represent derivations in bidirectional natural deduction, and $\overset{seq}{\vdash}$ to represent derivations in the sequent calculus. The three theorems we need now are

Theorem 2 *If* $\Delta \overset{nd}{\vdash} M : C$ *then* $\Delta \overset{seq}{\vdash} C$

Theorem 3 *If* $\Delta \overset{seq}{\vdash} C$ *then* $\Delta' \overset{\uparrow\downarrow}{\vdash} M \Leftarrow C$ *for some* M *where* $\Delta' \overset{sub}{\vdash} \Delta$

Theorem 4 *If* $\Delta \overset{\uparrow\downarrow}{\vdash} M \Leftarrow C$ *then* $\Delta \overset{nd}{\vdash} M : C$

Notice that in the second theorem, we need a modification to what we initially might think of as the theorem. We will come back to that in the proof.

Proof: Theorem 1
We proceed by induction over the natural deduction derivation. We provide the cases for $\otimes$ and $\multimap$ as we have been throughout these notes. However, we leave out the cases for the introduction rules as they follow directly via application of the induction hypothesis.

Case: $\otimes E$

$$\frac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Delta_1 \overset{nd}{\vdash} M : A \otimes B & \Delta_2, x : A, y : B \overset{nd}{\vdash} N : C\end{array}}{\Delta_1, \Delta_2 \overset{nd}{\vdash} \text{match } M \text{ with } (x, y) \text{ in } N : C} \otimes E$$

From the inductive hypothesis on $\mathcal{D}_1$ and $\mathcal{D}_2$ we can conclude

$$\Delta_1 \overset{seq}{\vdash} A \otimes B$$

$$\Delta_2, A, B \overset{seq}{\vdash} C$$

We need to prove $\Delta_1, \Delta_2 \overset{seq}{\vdash} C$.

Looking at the statements we have from our inductive hypothesis, there doesn't seem to be a way to proceed directly. The only applicable rule is $\otimes R$ but that isn't particularly useful. However, we have one more tool that we can use. We have the admissability of cut in the sequent calculus. We proceed with the proof as follows.

$$\frac{\begin{array}{cc} IH(\mathcal{D}_1) & \dfrac{IH(\mathcal{D}_2)}{\Delta_2, A, B \overset{seq}{\vdash} C} \otimes L \\ \Delta_1 \overset{seq}{\vdash} A \otimes B & \Delta_2, A \otimes B \overset{seq}{\vdash} C \end{array}}{\Delta_1, \Delta_2 \overset{seq}{\vdash} C} \; cut$$

Case: $\multimap E$

$$\frac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Delta_1 \overset{nd}{\vdash} M : A \multimap B & \Delta_2 \overset{nd}{\vdash} N : A\end{array}}{\Delta_1, \Delta_2 \overset{nd}{\vdash} M\ N : B} \multimap E$$

From the inductive hypothesis we conclude

$$\Delta_1 \vdash A \multimap B$$

$$\Delta_2 \vdash A$$

Again, there is no clear way forward unless we use the admissibility of cut (and in this case we also use the admissibility of identity). Once we do so, we obtain the following proof.

$$\dfrac{IH(\mathcal{D}_1) \quad \Delta_1 \overset{seq}{\vdash} A \multimap B \qquad \dfrac{IH(\mathcal{D}_2) \quad \Delta_2 \overset{seq}{\vdash} A}{\Delta_2, A \multimap B \overset{seq}{\vdash} B}\ \multimap L}{\Delta_1, \Delta_2 \overset{seq}{\vdash} B}\ cut \qquad \dfrac{}{B \overset{seq}{\vdash} B}\ \mathsf{id}$$

Cases for other connectives proceed in a similar fashion. □

Proof: Theorem 2
We now work through why we need the modification in theorem 2. Assume first that we had not made the modification and instead had just

$$\text{If } \Delta \overset{seq}{\vdash} C \text{ then } \Delta \overset{\uparrow\downarrow}{\vdash} M \Leftarrow C$$

We try to proceed with a proof of $\otimes L$

$$\dfrac{\begin{array}{c}\mathcal{D}' \\ \Delta, A, B \vdash C\end{array}}{\Delta, A \otimes B \vdash C}\ \otimes L$$

If we were to apply the inductive hypothesis based on the incorrect theorem statement, we get

$$\Delta, x : A, y : B \overset{\uparrow\downarrow}{\vdash} M \Leftarrow C$$

and get stuck. There are no clear rules we can apply here, and we also arbitrarily labeled propositions in the context with some variables without any clear reasoning. So instead, we want some way of translating a sequent calculus context into a natural deduction one, that allows us to apply rules to what was in the sequent calculus context. To do so we define the following rules:

$$\dfrac{}{\cdot \overset{sub}{\vdash} \cdot} \qquad \dfrac{\Delta_1' \overset{sub}{\vdash} \Delta \quad \Delta_2' \overset{\uparrow\downarrow}{\vdash} M \Rightarrow A}{\Delta_1', \Delta_2' \overset{sub}{\vdash} \Delta, A}$$

Δ_1' and Δ_2' are natural deduction contexts, while Δ is a sequent calculus context. We are able to go back to our corrected proof statement. Repeated here for convenience.

$$\text{If } \Delta \overset{seq}{\vdash} C \text{ then } \Delta' \overset{\uparrow\downarrow}{\vdash} M \Leftarrow C \text{ for some } M \text{ where } \Delta' \overset{sub}{\vdash} \Delta$$

Case: $\otimes L$

$$\dfrac{\begin{array}{c}\mathcal{D}'\\ \Delta, A, B \vdash C\end{array}}{\Delta, A \otimes B \vdash C}\otimes L$$

While we can no longer apply the induction hypothesis directly, we have some new assumptions to work with.

$$\Delta_{ab} \overset{\uparrow\downarrow}{\vdash} M \Rightarrow A \otimes B \tag{1}$$

$$\Delta' \overset{sub}{\vdash} \Delta \tag{2}$$

And we want to show

$$\Delta_{ab}, \Delta' \overset{\uparrow\downarrow}{\vdash} M' \Leftarrow C$$

We are now able to apply $\otimes E$ to (1) producing

$$\dfrac{\begin{array}{cc}(1) & ?\\ \Delta_{ab} \overset{\uparrow\downarrow}{\vdash} M \Rightarrow A \otimes B & \Delta', x : A, y : B \overset{\uparrow\downarrow}{\vdash} N \Leftarrow C\end{array}}{\Delta_{ab} \overset{\uparrow\downarrow}{\vdash} \text{match } M \text{ with } (x, y) \text{ in } N \Leftarrow C}\otimes E$$

We still need a proof of the second premise. Luckily, we can finally apply the inductive hypothesis! We can do so because from assumption we know $\Delta' \overset{sub}{\vdash} \Delta$, and from identity rules we have

$$x : A \overset{\uparrow\downarrow}{\vdash} x \Rightarrow A \tag{3}$$

$$y : B \overset{\uparrow\downarrow}{\vdash} y \Rightarrow B \tag{4}$$

We complete the proof

$$\dfrac{\begin{array}{cc}(1) & IH(\mathcal{D}', (2, 3, 4))\\ \Delta_{ab} \overset{\uparrow\downarrow}{\vdash} M \Rightarrow A \otimes B & \Delta', x : A, y : B \overset{\uparrow\downarrow}{\vdash} N \Leftarrow C\end{array}}{\Delta_{ab} \overset{\uparrow\downarrow}{\vdash} \text{match } M \text{ with } (x, y) \text{ in } N \Leftarrow C}\otimes E$$

Case: $\multimap L$

$$\dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2\\ \Delta_1 \overset{seq}{\vdash} A & \Delta_2, B \overset{seq}{\vdash} C\end{array}}{\Delta_1, \Delta_2, A \multimap B \overset{seq}{\vdash} C}\multimap L$$

Assumptions:

$$\Delta_1' \overset{sub}{\vdash} \Delta_1 \tag{5}$$

$$\Delta_2' \overset{sub}{\vdash} \Delta_2 \tag{6}$$

$$\Delta_{ab} \overset{\uparrow\downarrow}{\vdash} M \Rightarrow A \multimap B \tag{7}$$

We begin by applying $\multimap E$ on our last assumption.

$$\frac{(7) \qquad IH(\mathcal{D}_1, (5)) \\ \Delta_{ab} \overset{\uparrow\downarrow}{\vdash} M \Rightarrow A \multimap B \quad \Delta_1' \overset{\uparrow\downarrow}{\vdash} N \Leftarrow A}{\Delta_{ab}, \Delta_1' \overset{\uparrow\downarrow}{\vdash} M\ N \Rightarrow B} \multimap E \tag{8}$$

Now we can complete the proof by an application of the inductive hypothesis on $\mathcal{D}_2$ and the derivation (8) as well as assumption 6.

$$\frac{IH(\mathcal{D}_2, (6, 8))}{\Delta_1', \Delta_2', \Delta_{ab} \overset{\uparrow\downarrow}{\vdash} M' \Leftarrow C}$$

While we don't know specifically what term we have, we know it is possible to construct such a term. Other cases proceed similarly. □

Proof: Theorem 3 To prove the final theorem, we need to simultaneously prove one more theorem, since $\Rightarrow$ and $\Leftarrow$ are defined with references to each other. So this proof proceeds by simultaneous induction on the following two statements, over the derivation.

$$\text{If } \Delta \overset{\uparrow\downarrow}{\vdash} M \Leftarrow A \text{ then } \Delta \overset{nd}{\vdash} M : A$$

$$\text{If } \Delta \overset{\uparrow\downarrow}{\vdash} M \Rightarrow A \text{ then } \Delta \overset{nd}{\vdash} M : A$$

□

This proof should be self evident as a bidirectional proof with the two directions collapsed back into the single judgement : should still be a valid natural deduction proof (with some possible collapsing of the change of direction rule). We omit this proof and leave it as an exercise.

6 Curry-Howard Correspondence

Howard [1969] discovered the correspondence between one of the most fundamental models of computation, the lambda calculus and the implication fragment of natural deduction. This goes beyond just type checking. The proof reductions we did to demonstrate local soundness correspond to computational rules in the lambda calculus.

$$A \to B \qquad\qquad (\lambda x.M)N \Longrightarrow [N/x]M$$

This correspondence of course extends beyond just the lambda calculus, and encompasses the full natural deduction system and functional programming.

7 Full Bidirectional Rules

$$\frac{\Delta \vdash M \Longrightarrow A' \quad A = A'}{\Delta \vdash M \Longleftarrow A} \Rightarrow/\Leftarrow \qquad \frac{\Delta \vdash M \Longleftarrow A}{\Delta \vdash (M : A) \Longrightarrow A} \Leftarrow/\Rightarrow$$

$$\frac{}{x : A \vdash x \Longrightarrow A}\ \mathsf{hyp}$$

$$\frac{\Delta, x : A \vdash e \Longleftarrow B}{\Delta \vdash \lambda x.\, M \Longleftarrow A \multimap B} \multimap I$$

$$\frac{\Delta \vdash M \Longrightarrow A \multimap B \quad \Delta' \vdash N \Longleftarrow A}{\Delta, \Delta' \vdash M\, N \Longrightarrow B} \multimap E$$

$$\frac{\Delta \vdash M_\ell \Longleftarrow A_\ell \quad (\forall \ell \in L)}{\Delta \vdash \{\ell \Rightarrow M_\ell\}_{\ell \in L} \Longleftarrow \&\{\ell : A_\ell\}_{\ell \in L}} \&I \qquad \frac{\Delta \vdash M \Longrightarrow \&\{\ell : A_\ell\}_{\ell \in L} \quad (\ell \in L)}{\Delta \vdash M.\ell \Longrightarrow A_\ell} \&E$$

$$\frac{\Delta \vdash e_1 \Longleftarrow M \quad \Delta' \vdash N \Longleftarrow B}{\Delta, \Delta' \vdash (M, N) \Longleftarrow A \otimes B} \otimes I$$

$$\frac{\Delta \vdash M \Longrightarrow A \otimes B \quad \Delta', x_1 : A, x_2 : B \vdash N \Longleftarrow C}{\Delta, \Delta' \vdash \mathbf{match}\ M\ ((x_1, x_2) \Rightarrow N) \Longleftarrow C} \otimes E$$

$$\frac{}{\cdot \vdash (\,) \Longleftarrow \mathbf{1}}\ \mathbf{1}I \qquad \frac{\Delta \vdash M \Longrightarrow \mathbf{1} \quad \Delta' \vdash N \Longleftarrow C}{\Delta, \Delta' \vdash \mathbf{match}\ M\ ((\,) \Rightarrow N) \Longleftarrow C}\ \mathbf{1}E$$

$$\frac{\Delta \vdash M \Longleftarrow A_\ell}{\Delta \vdash \ell(M) \Longleftarrow \oplus\{\ell : A_\ell\}_{\ell \in L}} \oplus I$$

$$\frac{\Delta \vdash M \Longrightarrow \oplus\{\ell : A_\ell\}_{\ell \in L} \quad \Delta', x : A_\ell \vdash N_\ell \Longleftarrow C \quad (\forall \ell \in L)}{\Delta, \Delta' \vdash \mathbf{match}\ M\ (\ell(x) \Rightarrow N_\ell)_{\ell \in L} \Longleftarrow C} \oplus E$$

References

Henry DeYoung and Frank Pfenning. Data layout from a type-theoretic perspective. In *38th Conference on the Mathematical Foundations of Programming Semantics (MFPS 2022)*. Electronic Notes in Theoretical Informatics and Computer Science 1, 2022. URL https://arxiv.org/abs/2212.06321v6. Invited paper. Extended version available at https://arxiv.org/abs/2212.06321v3.pdf.

Henry DeYoung, Frank Pfenning, and Klaas Pruiksma. Semi-axiomatic sequent calculus. In Z. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, pages 29:1–29:22, Paris, France, June 2020. LIPIcs 167.

Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Computing Surveys*, 98 (5):1–38, 2022.

Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

W. A. Howard. The formulae-as-types notion of construction. Unpublished note. An annotated version appeared in: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, 479–490, Academic Press (1980), 1969.