Lecture Notes on Deductive Inference

15-816: Substructural Logics
Frank Pfenning

Lecture 1
August 30, 2016

According to Wikipedia[1], the ultimate authority on *everything*:

> **Logic** [...] is the formal systematic study of the principles of valid inference and correct reasoning.

We therefore begin the course with the study of deductive inference. This starting point requires surprisingly little machinery and is sufficient to understand the central idea behind substructural logics, including linear logic. We aim to develop all other concepts and properties of linear logic systematically from this seed.

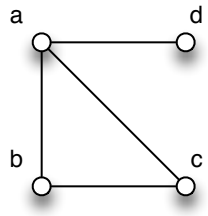
Our approach is quite different from that of Girard [Gir87], whose discovery of linear logic originated from semantic considerations in the theory of programming languages. We arrive at almost the same spot. The convergence of multiple explanations of the same phenomena is further evidence for the fundamental importance of linear logic. At some point in the course we will explicitly talk about the relationship between Girard's linear logic and our reconstruction of it.

1 Example: Reasoning about Graphs

As a first example we consider graphs. Mathematically, (undirected) graphs are often defined as consisting of a set of vertices V and a set of edges E, where E is a set of unordered pairs of vertices.

[1] in January 2012

In the language of logic, we represent the nodes (vertices) as *constants* $(\mathsf{a}, \mathsf{b}, \ldots)$ and a unary predicate $\mathsf{node}(x)$ that holds for all vertices. The edges are represented with a binary *predicate* $\mathsf{edge}(x, y)$ relating connected nodes.



The sample graph above could be represented by the *propositions*

$$\mathsf{node}(\mathsf{a}), \mathsf{node}(\mathsf{b}), \mathsf{node}(\mathsf{c}), \mathsf{node}(\mathsf{d}),$$
$$\mathsf{edge}(\mathsf{a}, \mathsf{b}), \mathsf{edge}(\mathsf{b}, \mathsf{c}), \mathsf{edge}(\mathsf{a}, \mathsf{c}), \mathsf{edge}(\mathsf{a}, \mathsf{d})$$

One mismatch one may notice immediately is that the edges in the picture seem to be undirected, while the representation of the edges is not symmetric (for example, $\mathsf{edge}(\mathsf{b}, \mathsf{a})$ is not there). We can repair this inadequacy by providing a *rule of inference* postulating that the edge relation is symmetric.

$$\frac{\mathsf{edge}(x, y)}{\mathsf{edge}(y, x)}\ \mathsf{sym}$$

We can apply this rule of inference to the fact $\mathsf{edge}(\mathsf{a}, \mathsf{b})$ to deduce $\mathsf{edge}(\mathsf{b}, \mathsf{a})$. In this application we instantiated the *schematic variables* x and y with a and b. We will typeset schematic variables in italics to distinguish them from constants. The propositions above the horizontal line are called the *premises* of the rule, the propositions below the line are called *conclusions*. This example rule has only one premise and one conclusion. sym is the *name* or *label* of the rule. We often omit rule names if there is no specific need to refer to the rules.

From this single rule and the facts describing the initial graph, we can now deduce the following additional facts:

$$\mathsf{edge}(\mathsf{b}, \mathsf{a}), \mathsf{edge}(\mathsf{c}, \mathsf{b}), \mathsf{edge}(\mathsf{c}, \mathsf{a}), \mathsf{edge}(\mathsf{d}, \mathsf{a})$$

Having devised a logical representation for graphs, we now define a relation over graphs. We write $\mathsf{path}(x, y)$ if there is a path through the graph

from x to y. A first path has length zero and goes from a node to itself:

$$\frac{\mathsf{node}(x)}{\mathsf{path}(x,x)}\ \mathsf{refl}$$

The following rule says is that we can extend an existing path by following an edge.

$$\frac{\mathsf{path}(x,y) \quad \mathsf{edge}(y,z)}{\mathsf{path}(x,z)}\ \mathsf{step}$$

From the representation of our example graph, when can then supply the following proof that there is a path from c to d:

$$\dfrac{\dfrac{\dfrac{\mathsf{node}(\mathsf{c})}{\mathsf{path}(\mathsf{c},\mathsf{c})}\ \mathsf{refl} \quad \dfrac{\mathsf{edge}(\mathsf{a},\mathsf{c})}{\mathsf{edge}(\mathsf{c},\mathsf{a})}\ \mathsf{sym}}{\mathsf{path}(\mathsf{c},\mathsf{a})}\ \mathsf{step} \qquad \mathsf{edge}(\mathsf{a},\mathsf{d})}{\mathsf{path}(\mathsf{c},\mathsf{d})}\ \mathsf{step}$$

We can examine the proof and see that it carries some information. It is not just there to convince us that there is a path from c to d, but it tells us the path. The path goes from c to a and then from a to d. This is an example of *constructive content* in a proof, and we will see many other examples. For the system we have so far it will be true in general that we can read off a path from a proof, and if we have a path in mind we can always construct a proof. But with the rules we chose, some paths do not correspond to a unique proof. Think about why before turning the page. . .

Proofs are not unique because we can go from $\mathsf{edge}(\mathsf{c},\mathsf{a})$ back to $\mathsf{edge}(\mathsf{a},\mathsf{c})$ and back $\mathsf{edge}(\mathsf{c},\mathsf{a})$, and so on, producing infinitely many proofs of $\mathsf{edge}(\mathsf{c},\mathsf{a})$. Here are two techniques to eliminate this ambiguity which are of general interest.

One solution uses a new predicate pre_edge and defines all the edges we have used so far as pre-edges. Then we have two rules

$$\frac{\mathsf{pre\_edge}(x,y)}{\mathsf{edge}(x,y)} \qquad \frac{\mathsf{pre\_edge}(x,y)}{\mathsf{edge}(y,x)}$$

To explain the second solution, we begin by making proofs explicit as objects. We write $e : \mathsf{edge}(x,y)$ to name the edge from x to y, and $p : \mathsf{path}(x,y)$ for the path from x to y. Where the proofs are not interesting, we just use an underscore _. We annotate our rules accordingly, which now read:

$$\frac{e : \mathsf{edge}(x,y)}{e^{-1} : \mathsf{edge}(y,x)}\ \mathsf{sym} \qquad \frac{\_ : \mathsf{node}(x)}{\mathsf{r} : \mathsf{path}(x,x)}\ \mathsf{refl} \qquad \frac{p : \mathsf{path}(x,y) \quad e : \mathsf{edge}(y,z)}{p \cdot e : \mathsf{path}(x,z)}\ \mathsf{step}$$

Our initial knowledge base would be annotated

$$\begin{array}{l} \_ : \mathsf{node}(\mathsf{a}), \_ : \mathsf{node}(\mathsf{b}), \_ : \mathsf{node}(\mathsf{c}), \_ : \mathsf{node}(\mathsf{d}), \\ \mathsf{e}_{ab} : \mathsf{edge}(\mathsf{a},\mathsf{b}), \mathsf{e}_{bc} : \mathsf{edge}(\mathsf{b},\mathsf{c}), \mathsf{e}_{ac} : \mathsf{edge}(\mathsf{a},\mathsf{c}), \mathsf{e}_{ad} : \mathsf{edge}(\mathsf{a},\mathsf{d}) \end{array}$$

and then the earlier proof would carry the path information:

$$\dfrac{\dfrac{\dfrac{\_ : \mathsf{node}(\mathsf{c})}{\mathsf{r} : \mathsf{path}(\mathsf{c},\mathsf{c})}\ \mathsf{refl} \quad \dfrac{\mathsf{e}_{ac} : \mathsf{edge}(\mathsf{a},\mathsf{c})}{\mathsf{e}_{ac}^{-1} : \mathsf{edge}(\mathsf{c},\mathsf{a})}\ \mathsf{sym}}{\mathsf{r} \cdot \mathsf{e}_{ac}^{-1} : \mathsf{path}(\mathsf{c},\mathsf{a})}\ \mathsf{step} \qquad \mathsf{e}_{ad} : \mathsf{edge}(\mathsf{a},\mathsf{d})}{\mathsf{r} \cdot \mathsf{e}_{ac}^{-1} \cdot \mathsf{e}_{ad} : \mathsf{path}(\mathsf{c},\mathsf{d})}\ \mathsf{step}$$

To force uniqueness of proofs of the edge predicate we can assert the equation

$$(e^{-1})^{-1} = e$$

so that we do not distinguish between the original edge and one that has been reversed twice. Note that for any given path there will still be many formal deductions using the inference rules, but they would lead to the same proof object.

Both solutions, restructuring the predicates or identifying proofs, are common solutions to create better correspondences between proofs and the properties of information we would like them to convey.

As a last example here, let's consider what happens when we apply inference indiscriminately, trying to learn as much as possible about the paths in a graph. Unfortunately, even if the edge relation has unique proofs, any cycle in the graph will lead to infinitely many paths and therefore infinitely many proofs. So inference will never stop with complete immediate knowledge of all paths.

One solution[2] is to retreat and say we do not care about the path, we just want to know if there exists a path between any two nodes. In that case inference will reach a point of *saturation*, that is, any further inference would have a conclusion already in our database of facts. The original inference system achieves that, or we can stipulate that any two paths between the same two vertices x and y should be identified:

$$p = q : \mathsf{path}(x, y)$$

Again, inference would reach saturation because after a time new paths would be equal to the one we already have and not recorded as a separate fact.

Another solution would be to allow only non-repetitive paths, by which we mean that edges are not being reused. Then we would have to modify our step rule

$$\frac{p : \mathsf{path}(x, y) \quad e : \mathsf{edge}(y, z) \quad \_ : \mathsf{notin}(e, p)}{p \cdot e : \mathsf{path}(x, z)}\ \mathsf{step}$$

and define a predicate $\mathsf{notin}(e, p)$ which is true if e is not in the path p. As was noted in class, however, this crosses a threshold: propositions (like notin) now refer to proofs. This is the distinction between *logic*, in which proofs entirely separate from propositions, and *type theory* in which propositions can refer to proofs. This particular example shows that type theory can be more expressive in certain helpful ways.

A third solution would be to somehow *consume* the edges during inference so that we cannot reuse them later. Because there is no restriction on how often primitive or derived facts are used in a proof, we will need the expressive power of *substructural logics* to follow this idea.

[2] not discussed in lecture

2 Example: Coin Exchange

So far, deductive inference has always accumulated knowledge since propositions whose truth we are already aware of remain true. Linear logic arises from a simple observation:

Truth is ephemeral.

For example, while giving this lecture "*Frank is holding a whiteboard marker*" was true, and right now it is (most likely) not. So truth changes over time, and this phenomenon is studied with *temporal logic*. In *linear logic* we are instead concerned with the change of truth with a *change of state*. We model this in a very simple way: when an inference rule is applied we *consume* the propositions used as premises and *produce* the propositions in the conclusions, thereby effecting an overall change in state.

As an example, we consider *nickels* (n) worth 5¢, *dimes* (d) worth 10¢, and *quarters* (q) worth 25¢. We have the following rules for exchange between them

$$\frac{\mathsf{d} \quad \mathsf{d} \quad \mathsf{n}}{\mathsf{q}}\ Q \qquad \frac{\mathsf{q}}{\mathsf{d} \quad \mathsf{d} \quad \mathsf{n}}\ Q^{-1} \qquad \frac{\mathsf{n} \quad \mathsf{n}}{\mathsf{d}}\ D \qquad \frac{\mathsf{d}}{\mathsf{n} \quad \mathsf{n}}\ D^{-1}$$

The second and fourth rules are the first rules we have seen with more than one conclusion. Inference now changes state. For example, if we have three dimes and a nickel, the state would be written as

$$\mathsf{d}, \mathsf{d}, \mathsf{d}, \mathsf{n}$$

Applying the first rule, we can turn two dimes and a nickel into a quarter to get the state

$$\mathsf{d}, \mathsf{q}$$

Note that the total value of the coins (35¢) remains unchanged, which is the point of a coin exchange. One way to write down the inference is to cross out the propositions that are consumed and add the ones that are produced. In the above example we would then write something like

$$\mathsf{d}, \mathsf{d}, \mathsf{d}, \mathsf{n} \qquad \rightsquigarrow \qquad \not{\mathsf{d}}, \not{\mathsf{d}}, \mathsf{d}, \not{\mathsf{n}}, \mathsf{q}$$

In order to understand the meaning of proof, consider how to change three dimes into a quarter and a nickel: first, we change one dime into two nickels, and then the other two dimes and one of the nickels into a quarter. As two state transitions:

$$\mathsf{d}, \mathsf{d}, \mathsf{d} \qquad \rightsquigarrow \qquad \mathsf{d}, \mathsf{d}, \not{\mathsf{d}}, \mathsf{n}, \mathsf{n} \qquad \rightsquigarrow \qquad \not{\mathsf{d}}, \not{\mathsf{d}}, \not{\mathsf{d}}, \not{\mathsf{n}}, \mathsf{q}, \mathsf{n}$$

Using inference rule notation, this deduction is shown on the left, and the corresponding derived rule of inference on the right.

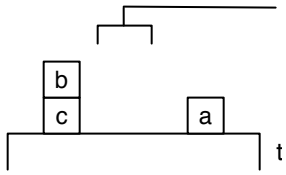
$$\frac{\mathsf{d} \quad \mathsf{d} \quad \dfrac{\mathsf{d}}{\mathsf{n}} \quad \mathsf{n}}{\mathsf{q}} \qquad\qquad \frac{\mathsf{d} \quad \mathsf{d} \quad \mathsf{d}}{\mathsf{q} \quad \mathsf{n}}$$

To summarize: we can change the very nature of inference if we *consume* the propositions used in the premise to *produce* the propositions in the conclusions. This is the foundation of linear logic and we therefore call it *linear inference*. The requisite pithy saying to remind ourselves of this:[3]

> *Linear inference can change the world.*

3 Example: Blocks World

Next we consider *blocks world*, which is a venerable example in the history of artificial intelligence. We have blocks ($\mathsf{a}, \mathsf{b}, \ldots$) stacked on a table ($\mathsf{t}$). We also have a robot hand which may pick blocks that are not obstructed and put them down on the table or some other block. We assume that the hand can hold just one block.



We represent the state in linear logic using the following predicates

$\mathsf{on}(x, y)$	Block x is on top of y
empty	Hand is empty
$\mathsf{holds}(x)$	Hand holds block x

[3] with a tip of the hat to Phil Wadler

For example, the state above would be represented by

$$\mathsf{empty}, \mathsf{on}(\mathsf{c}, \mathsf{t}), \mathsf{on}(\mathsf{b}, \mathsf{c}), \mathsf{on}(\mathsf{a}, \mathsf{t})$$

The two preconditions for picking up a block x are that the hand is empty, and that nothing is on top x. In (ordinary) logic, we might try to express this last condition as $\neg\exists y.\, \mathsf{on}(y, x)$. However, negation is somewhat problematic in linear logic. For example, it is true in the above state the $\neg\mathsf{on}(\mathsf{b}, \mathsf{t})$. However, after two moves (picking up b and then putting b on the table) it could become true, and we might have created a contradiction! Moreover, proving $\neg\exists y.\mathsf{on}(y, x)$ in linear logic would consume whatever resources we refer to, which is problematic since our intention is just to check a property of the state without changing it. This kind of problem is common in applications of linear logic, so we have developed some techniques of addressing it.

A common technique to avoid such paradoxes is to introduce additional predicates that describe properties of the state. Such predicates are maintained by the rules in order to keep the description of the state consistent. Before reading on, you might consider how you can define a new predicate, add appropriate initial facts, and then write rules for picking up and putting down blocks.

Here is one possibility. We use a new predicate $\mathsf{clear}(x)$ to express that there is no other block on top of x. In addition to the propositions above describing the initial state, we would also have

$$\mathsf{clear}(\mathsf{b}), \mathsf{clear}(\mathsf{a})$$

but *not* $\mathsf{clear}(\mathsf{c})$.

Then we only need two rules, one for picking up a block and one for putting one down:

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

The two rules are inverses of each other, which makes sense since picking up or putting down a block are reversible actions.

The encoding so far would work under the condition that there is a fixed number of spots on the table where blocks can be deposited, and each is explicitly represented by a $\mathsf{clear}(\mathsf{t})$ proposition. We cannot accidentally pick up the table because it is not *on* anything else, so the last premise of the pickup rule cannot be satisfied when x is t.

If we instead want to follow the usual assumption assume that the table is big enough for arbitrarily many blocks we can revise our encoding in two ways. One would be to introduce a special predicate $\mathsf{ontable}(x)$ and reserve $\mathsf{on}(x,y)$ for blocks x and y. Then we need two additional rules to pick up and put down blocks on the table. Another is to have a *persistent* proposition

$$\underline{\mathsf{clear}}(\mathsf{t})$$

which *can not be consumed by inference*. The intrinsic attribute of the predicate of being persistent is indicated by underlining it.

We view whether propositions are ephemeral (linear, will be consumed by inference) or persistent (not linear, can not be consumed by inference) as intrinsic attributes of the proposition. This allows us to easily mix linear and nonlinear inference, which is a powerful tool in constructing encodings. We can continue to use any encoding in "ordinary"[4] logic but we have some new means of expression.

Substructural logics generalize ordinary logics

One interesting question that arises here is whether we can use persistent facts to instantiate ephemeral premises of rules. This is the intention

[4] by which we mean "not substructural"

here: $\underline{\mathsf{clear}}(t)$ can be used as a premise of the putdown rule in order to put a block on the table. This is justified since a fact we can use as often as we want should certainly be usable this once. We just have to be careful *not* to consume $\underline{\mathsf{clear}}(t)$ so that it remains available for future inferences.

As before, we should examine the meaning of proofs in this example. Given some initial state, a proof describes a sequence of moves that leads to a final state. Therefore, the process of planning, when given particular goal and final states, becomes a process of proof search.

Another important aspect of problem representations in linear logic are *state invariants* that are necessary so that inference has the desired meaning. For example, there should always be *exactly* one of empty and $\mathsf{holds}(x)$ in w state, otherwise it would not conform to our problem domain and inference is potentially meaningless. Similarly, there shouldn't be cycles such as $\mathsf{on}(\mathsf{a},\mathsf{b}), \mathsf{on}(\mathsf{b},\mathsf{a})$, which does not correspond to any physically possible situation. When showing that our representations in linear logic capture what we intend, we should make such state invariants explicit and verify that they are preserved under the possible inferences. In our example, any rule application replaces empty by $\mathsf{holds}(x)$ for some x, or $\mathsf{holds}(x)$ by empty. So if the state invariant holds initially, it must hold after any inference.

4 Example: King Richard III

As a first example from literature, consider the following quote:

> *My kingdom for a horse!* — King Richard in *Richard III* by William Shakespeare

How do we represent this in linear logic? Let's fix a vocabulary:

$\mathsf{richard}$	King Richard III
$\mathsf{owns}(x, y)$	x owns y
$\mathsf{horse}(x)$	x is a horse
$\mathsf{kingdom}(x)$	x is a kingdom

The offer corresponds to a change of ownership: Richard "owns" a kingdom before (which we know to be England, but which is not part of his utterance), and another person p owns a horse, and after the swap Richard owns the horse while p owns the kingdom.

$$\frac{\mathsf{owns}(\mathsf{richard}, k) \quad \underline{\mathsf{kingdom}}(k) \quad \mathsf{owns}(p, h) \quad \underline{\mathsf{horse}}(h)}{\mathsf{owns}(\mathsf{richard}, h) \quad \mathsf{owns}(p, k)}$$

As is commonly the case, we model an intrinsic attribute of an object (such as h being a horse or k being a kingdom) with a persistent predicate, while ownership changes and is therefore ephemeral.

It is implied in the exclamation that the rule can only be used once, because there is only one k that qualifies as "*my kingdom*". Otherwise, he would have said "*One of my kingdoms for a horse!*". Another way to capture this aspect of the offer would be to consider the inference rule itself to be *ephemeral* and hence can only be used once. We do not have a good informal notation for such rules, but they will play a role again in the next lecture. If the rule above were persistent, it would more properly correspond to the offer "*My kingdoms for horses!*".

5 Example: Grammatical Inference

We have seen how we can add expressive power to logic simply by designating some propositions as being consumable by logical inference. We now explore going even further: we also impose an order on the propositions describing our state of knowledge. This is the fundamental idea behind the *Lambek calculus* [Lam58][5]. Lambek's goal was to describe the syntax of natural (and formal) languages and apply logical tools to problems such as parsing. We only scratch the surface of the work, just enough to appreciate how Lambek employs logical inference to describe grammatical inference. We will also see our first logical connectives, called "over" and "under", although you would be unlikely to recognize them as such at first glance.

We begin by introducing so-called *syntactic types* that describe the role of words and phrases. The primitive types we consider here are only n for *names* and s for *sentences*. We ascribe n for proper names such as *Alice* or *Bob*, but phrases that can stand for names in sentences will also have type n. s is the type of complete sentences such as *Alice works* or *Bob likes Alice*.

There are two constructors two combine types: Given types x and y, we can form $x \,/\, y$ (read: x *over* y) and $x \setminus y$ (read: x *under* y). If a phrase w has type $x \,/\, y$ it means that we obtain an x if we find a y to its right. For example, *poor Bob* can act as a name because we can (syntactically) use it where we use *Bob*. This means we can assign *poor* the type $n \,/\, n$: If we find a name to its right the result forms a name. Conversely, $x \setminus y$ describes a phrase that acts as a y if there is an x to its *left*. For example, an intransitive

[5] I highly recommend this paper, which has stood the test of time almost 50 years after its publication.

verb like *works* will have type $n \setminus s$, because if we find a name to its left we obtain a sentences, as in *Alice works*.

This informal definition can be seen as a form of logical inference. We have the two rules

$$\frac{x \,/\, y \quad y}{x}\ \text{over} \qquad \frac{y \quad y \setminus x}{x}\ \text{under}$$

For these to make sense from the grammatical point of view, it is critical that the premises of the rules must be adjacent and in the right order.

Under this point of view, proofs correspond to parse trees and the words of the phrase we are trying to parse are labeled by their type. Using p and q to stand for phrases (that is, sequences of words), we would have

$$\frac{p : x \,/\, y \quad q : y}{(p\,q) : x}\ \text{over} \qquad \frac{q : y \quad p : y \setminus x}{(q\,p) : x}\ \text{under}$$

For example, we might start with the phrase *poor Alice works* as

$$(\textit{poor} : n \,/\, n)\ (\textit{Alice} : n)\ (\textit{works} : n \setminus s)$$

In this situation, there are two possible inferences, either connecting *poor* with *Alice* or connecting *Alice* with *works*. Let's try the first one. We then obtain

$$\frac{\textit{poor} : n \,/\, n \quad \textit{Alice} : n}{(\textit{poor Alice}) : n}\ \text{over} \qquad \textit{works} : n \setminus s$$

After one more step we obtain a complete parse

$$\frac{\dfrac{\textit{poor} : n \,/\, n \quad \textit{Alice} : n}{(\textit{poor Alice}) : n}\ \text{over} \qquad \textit{works} : n \setminus s}{((\textit{poor Alice})\,\textit{works}) : s}\ \text{under}$$

Trying the other order fails, since *poor* does not properly modify *Alice works*. Let's see how this failure arises during deduction. After one step we have

$$\textit{poor} : n \,/\, n \qquad \frac{\textit{Alice} : n \quad \textit{works} : n \setminus s}{(\textit{Alice}\,\textit{works}) : s}\ \text{under}$$

At this point in the deductive process, no further inference is possible.

We now consider a few more words to find appropriate types for them. The word *here* transform a sentence to another sentence when attached to

the end. For example, writing the typing judgment vertically and indicating the expected parse tree by parentheses, we have

$$\begin{array}{ccc} (\textit{Alice} & \textit{works}) & \textit{here} \\ : & : & : \\ n & n \setminus s & ? \end{array}$$

We see that the type of *here* should be $s \setminus s$.

$$\begin{array}{ccc} (\textit{Alice} & \textit{works}) & \textit{here} \\ : & : & : \\ n & n \setminus s & s \setminus s \end{array}$$

or, making the inference more explicit

$$\dfrac{\dfrac{\begin{array}{cc} \textit{Alice} & \textit{works} \\ : & : \\ n & n \setminus s \end{array}}{s} \qquad \begin{array}{c} \textit{here} \\ : \\ s \setminus s \end{array}}{s}$$

How about *never*, which modifies an intransitive verb?

$$\begin{array}{ccc} \textit{Bob} & (\textit{never} & \textit{works}) \\ : & : & : \\ n & ? & n \setminus s \end{array}$$

We suggest you work this out before going on to the next page...

We see that *never* should have type $(n \setminus s) \,/\, (n \setminus s)$.

$$\begin{array}{ccc} \textit{Bob} & (\textit{never} & \textit{works}) \\ : & : & : \\ n & (n \setminus s) \,/\, (n \setminus s) & n \setminus s \end{array}$$

or, as a complete inference (= parse) tree:

$$\dfrac{\textit{Bob} : n \qquad \dfrac{\textit{never} : (n \setminus s) \,/\, (n \setminus s) \qquad \textit{works} : (n \setminus s)}{n \setminus s}}{s}$$

What about a transitive verb such as *likes*? Setting up a standard sentence

$$\begin{array}{ccc} \textit{Alice} & \textit{likes} & \textit{Bob} \\ : & : & : \\ n & ? & n \end{array}$$

it seems like there are two possibilities, depending on whether we prefer the parse *(Alice likes) Bob* or *Alice (likes Bob)*. The first yields the type $n \setminus (s \,/\, n)$, while the second yields $(n \setminus s) \,/\, n$. Using the tools of logic we will see in the next lecture that these two types are equivalent in the sense that when we can parse a phrase with the first we can always parse it with the second and vice versa. So it should not matter which type we assign, although if we think of parsing as proceeding from left to right one might have an intuitive preference for the first one.

Of course, natural language has many ambiguities and the same work can be used in different ways. For example, the work *and* can conjoin sentences (for example, *Alice works and Bob rests* with type $\textit{and} : s \setminus (s \,/\, s)$) but also names (for example, *Alice and Bob work* with type $\textit{and} : n \setminus (n^* \,/\, n)$ where n^* denotes plural ouns such as *women*).

Here is a little summary table, showing some of the possible types.

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

6 Summary

We have examined the notion of *logical inference* from three different angles. The first was to accumulate knowledge by deducing new facts from facts we already knew, as in describing graphs using propositions $\mathsf{vertex}(x)$ and $\mathsf{edge}(x, y)$ and paths through graphs using $\mathsf{path}(x, y)$. Proofs here contain important information. For example, the proof that there is a path from x to y contains the actual edges traversed.

We then generalized the situation by stipulating that inference *consumes* the propositions it uses unless they are specified as *persistent*. This allows us to represent change of state in a logical manner, which we illustrated using a coin exchange and blocks world, a classic artificial intelligence domain. A proof here corresponds to a (partially ordered) sequence of actions.

Finally, we followed Lambek [Lam58] by requiring our facts not only to be consumed by inference, but remain *ordered* so we could represent grammatical inference. We saw our first two logical connectives, $x \mathbin{/} y$ and $x \mathbin{\backslash} y$ in order to represent the syntactic types of various parts of speech. In this example, a proof corresponds to a parse tree.

7 Bonus Materials

We complete the lecture notes here with some bonus materials from previous instance of the course, not covered this year.

7.1 [Example: Natural Numbers]

As a second example, we consider natural numbers $0, 1, 2, \ldots$. A convenient way to construct them is via iterated application of a successor function s to $\mathsf{0}$, written as

$$\mathsf{0}, \mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{s}(\mathsf{0})), \ldots$$

We refer to s as a *constructor*. Now we can define the even and odd numbers through the following three rules.

$$\frac{}{\mathsf{even}(\mathsf{0})} \qquad \frac{\mathsf{even}(x)}{\mathsf{odd}(\mathsf{s}(x))} \qquad \frac{\mathsf{odd}(x)}{\mathsf{even}(\mathsf{s}(x))}$$

As an example of a derived rule of inference, we can summarize the proof on the left with the rule on the right:

$$\dfrac{\dfrac{\mathsf{even}(x)}{\mathsf{odd}(\mathsf{s}(x))}}{\mathsf{even}(\mathsf{s}(\mathsf{s}(x)))} \qquad \frac{\mathsf{even}(x)}{\mathsf{even}(\mathsf{s}(\mathsf{s}(x)))}$$

The structure of proofs in these examples is not particularly interesting, since proofs that number a number of n is even or odd just follow the structure of the number n.

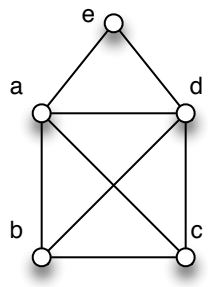
7.2 [Example: Graph Drawing]

We proceed to a slightly more sophisticated example involving linear inference. Before we used ordinary deductive inference to define the notion of path. This time we want to model drawing a graph without lifting the pen. This is the same as traversing the whole graph, going along each edge exactly once. This second formulation suggest the following idea: as we go along an edge we *consume* this edge so that we cannot follow it again. We also have to keep track of where we are, so we introduce another predicate $\mathsf{at}(x)$ which is true if we are at node x. The only rule of *linear* inference then is

$$\frac{\mathsf{at}(x) \quad \mathsf{edge}(x, y)}{\mathsf{at}(y)}\ \mathsf{step}$$

We start with an initial state just as before, with an $\mathsf{edge}(x, y)$ for each edge from x to y, the rule of symmetry (since have an undirected graph), and a starting position $\mathsf{at}(x_0)$. We can see that every time we take a step (by applying the step rule shown above), we consume a fact $\mathsf{at}(x)$ and produce another fact $\mathsf{at}(y)$, so there will always be exactly one fact of the form $\mathsf{at}(-)$ in the state. Also, at every step we consume one $\mathsf{edge}(-)$ fact, so we can take at most as many steps as there are edges in the graph initially. Of course, if we are at a point x there may be many outgoing edges, and if we pick the wrong one we may not be able to complete the drawing, but at least the number of steps we can try is limited at each point. We succeed, that is, we have found a way to draw the graph witout lifting the pen if we reach a state without an $\mathsf{edge}(-)$ fact and some final position $\mathsf{at}(x_n)$.

The following example graph is from a German children's rhyme[6] and can be drawn in one stroke if we start at b or c, but not if we start at a, d, or e.



We leave it to the reader to construct a solution and then translate it to a proof. Also, if we remove node e and its edges to a and d, no solution is possible.

Let's make the meaning of proofs explicit again. Because we have only one inference rule concerned with a move, a proof in general will have the following shape:

$$\dfrac{\dfrac{\dfrac{\mathsf{at}(x_0) \quad \mathsf{edge}(x_0, x_1)}{\ddots} \mathsf{step} \quad \mathsf{edge}(x_{n-2}, x_{n-1})}{\mathsf{at}(x_{n-1})} \mathsf{step} \quad \mathsf{edge}(x_{n-1}, x_n)}{\mathsf{at}(x_n)} \mathsf{step}$$

[6] *"Das ist das Haus vom Ni-ko-laus."*

This proof represents the path $x_0, x_1, \ldots, x_{n-1}, x_n$. Of course, some of these nodes may be the same, as can be seen by examining the example, but no *edge* can be used twice. If we need to traverse an edge in the opposite direction from its initial specification, we need to use symmetry, which will consume the edge and produce its inverse. We can then use the inverse to make a step. Being able to continually flip the direction of edges is a potential source of nontermination in the inference process. This does not invalidate the observation that a path along non-repeating edges can be written as a proof, and from a proof we can read off a path of non-repeating edges. This path represents a solution if the initial and final states are as described above.

7.3 [Example: Graph Traversal]

If we just want to traverse the graph rather than draw it, we should not destroy the edges as we move. A standard way to accomplish this is to explicitly recreate edges as we move:

$$\frac{\mathsf{at}(x) \quad \mathsf{edge}(x,y)}{\mathsf{at}(y) \quad \mathsf{edge}(x,y)}\ \mathsf{step}$$

Another way is to distinguish *ephemeral propositions* from *persistent propositions*. Ephemeral propositions are consumed during inference, while persistent propositions are never consumed. This allows us to have a uniform framework encompassing both the ordinary logical inference where we just add the conclusions to our store of knowledge, and linear logical inference.

We indicate the disposition of propositions that are known to be true by writing $A\ \mathit{eph}$ and $A\ \mathit{pers}$. Here, A stands for a proposition and $A\ \mathit{eph}$ and $A\ \mathit{pers}$ are called *judgments*. Distinguishing judgments from propositions is one of the cornerstones of Per Martin-Löf's approach to the foundation of logic and programming languages [ML83]. From now on we will follow the idea that the subjects of inference rules are judgments *about* propositions, not the propositions themselves. Mostly, they express that propositions are true, but in a variety of ways: ephemerally true, persistently true, true at time t, etc. There are a number of different terms that have been used for the particular distinction we are making here:

A *ephemeral*	A *persistent*
A *linear*	A *unrestricted*
A *true*	A *valid*
A *contingently true*	A *necessarily true*

Our high-level message is:

> *Truth is ephemeral; validity is forever.*

In the example, we can use this by declaring edges to be persistent, in which case the premise and conclusion of the symmetry rule should also be persistent. We abbreviate $ephemeral$ by eph, and $persistent$ by $pers$.

$$\frac{\mathsf{at}(x)\ eph \quad \mathsf{edge}(x,y)\ pers}{\mathsf{at}(y)\ eph}\ \mathsf{step}$$

As an even more compact notation, we underline persistent propositions; if they are not underlined, they should be considered ephemeral. For example, there would appear to be little reason for the properties of being even or odd to be ephemeral, since they should be considered intrinsic properties of the natural numbers. We therefore write

$$\frac{}{\underline{\mathsf{even}}(0)} \qquad \frac{\underline{\mathsf{even}}(x)}{\underline{\mathsf{odd}}(\mathsf{s}(x))} \qquad \frac{\underline{\mathsf{odd}}(x)}{\underline{\mathsf{even}}(\mathsf{s}(x))}$$

The first rule here is an inference rule with no premise. Persistent conclusions of such rules are sometimes called *axioms* in the sense that they are persistently true.

7.4 [Example: Opportunity]

A common proverb states:

> *Opportunity doesn't knock twice.* — Anonymous

Again, let's fix the vocabulary:

$\mathsf{opportunity}$	opportunity
$\mathsf{knocks}(x)$	x knocks

Then the preceding saying is just

$$\mathsf{knocks}(\mathsf{opportunity})\ eph$$

where we have written out the judgment $ephemeral$ for emphasis.

Clearly, when this fact is used it cannot be used again. If it is never used, we do not consider opportunity to having ever knocked, so the judgment above captures the idea that opportunity knocks at most once (and therefore not twice).

Exercises

Exercise 1 As an alternative way to defining paths through graph, we can say that paths represent the reflexive and transitive closure of the (symmetric) edge relation. Write out the inference rules as in Section 1, first without proof terms. Now add proof terms and an appropriate equality relation on paths. Explain your choice of equational theory.

Exercise 2 Write out proof for the example graph from subsection 7.2 that demonstrates that the figure can be drawn in one stroke.

Exercise 3 Consider the representation of undirected graphs from Section 1, with predicates $\mathsf{node}(x)$ and $\mathsf{edge}(x, y)$.

A *Hamiltonian cycle* is a path through a graph that visits each vertex exactly once and also returns to the starting vertex. Define any additional predicates you need and give inference rules such that inference corresponds to traversing the graph, and there is a simple condition that can be checked at the end to see if the traversal constituted a Hamiltonian cycle.

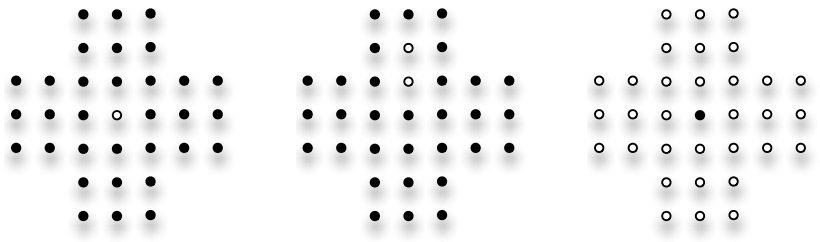
Exercise 4 Consider the representation of undirected graph from Section 1, with predicates $\mathsf{node}(x)$ and $\mathsf{edge}(x, y)$. We add a new predicate $\mathsf{color}(x)$ to express the color of node x. A *valid coloring* is one where no two adjacent nodes (that is, two nodes connected by an edge) have the same color.

(i) Give inference rules that can deduce $\mathsf{invalid}(x, y)$ if and only if there are adjacent nodes x and y with the same color.

(ii) Give inference rules where proofs from an initial state to a final state satisfying an easily checkable condition correspond to valid colorings. Carefully describe your initial state and the property of the final state.

Exercise 5 Consider the game *Peg Solitaire*. We have a layout of holes (shown as hollow circles), all but one of which are filled with pegs (shown as filled circles). We move by taking one peg, jumping over an adjacent one into an empty hole behind it, removing the peg in the process. The situation after one of the four possible initial moves is shown in the second diagram. The

third diagram shows the desired target position.



initial position after one move target position

Define a vocabulary and give inference rules in a representation of peg solitaire so that linear inference corresponds to making legal jumps. Explain how you represent the initial position, and how to test if you have reached the target position and thereby won the solitaire game.

Exercise 6 We consider the blocks world example from Section 3. As for graphs where we had the node predicate, it is convenient to add a new ephemeral predicate block(x) which is true for every block in the configuration.

Write a set of rules such that they can consume all ephemeral facts in the state (leaving only the persistent clear(t)) if and only if all of the following conditions are satisfied:

(i) the configuration of blocks is a collection of simple stacks,

(ii) the top of each stack is known to be clear, and

(iii) either the hand is empty or holds a block x, but not both.

If you believe it cannot be done, solve as many of the conditions as you can, explain why not all of them can be checked, and explore alternatives. Note linear inference allows rules with no premises or no conclusions.

Exercise 7 Render the following statement by an American president as an inference rule in linear logic:

If you can't stand the heat, get out of the kitchen. — Harry S. Truman

Use the following vocabulary:

toohot(x)	x cannot stand the heat
in(x, y)	x is in y
kitchen	the kitchen

Exercise 8 Render the following statement in linear logic (without the use of any logical connectives):

> *Truth is ephemeral; validity is forever.* — Frank Pfenning, page 19

Use the vocabulary

truth	truth
validity	validity

References

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Lam58] Joachim Lambek. The mathematics of sentence structure. *The American Mathematical Monthly*, 65(3):154–170, 1958.

[ML83] 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.

Lecture Notes on
From Rules to Propositions

15-816: Substructural Logics
Frank Pfenning

Lecture 2
September 1, 2016

We review the ideas of ephemeral truth and linear inference with another example from graph theory: constructing spanning trees for graphs. Next we consider how the process of grammatical inference in the Lambek calculus [Lam58], where an order of facts is prescribed, can be used to define other forms of computation. Finally, we generalize our formalization of logical inference to encompass *hypothetical reasoning* which will give rise to Gentzen's sequent calculus [Gen35]. The sequent calculus will be the gateway that allows us to move from pure logical inference to a definition of logical connectives.

1 Example: Spanning Trees

A *spanning tree* for a connected graph is a graph that has the same nodes but only a subset of the edges such that there is no cycle. In order to define rules for constructing a spanning tree for a graph we will simultaneously manipulate two graphs: the original graph and its spanning tree. We therefore add a third argument to our representation of graphs (from Lecture 1) which identifies *which* graph a node or edge belongs to.

$\mathsf{node}(x, g)$	x is a node in graph g
$\mathsf{edge}(x, y, g)$	there is an edge from x to y in graph g

The rule of symmetry stays within one graph g:

$$\frac{\mathsf{edge}(x, y, g)}{\mathsf{edge}(y, x, g)}\ \mathsf{sym}$$

Now assume we have a graph g and want to build a spanning tree t. Here is a simple algorithm for building t. We begin by picking an arbitrary node x from g and create t with x as its only node. Now we repeatedly pick an edge that connects a node x already in the tree with a node y not yet in the tree and add that edge and the node y into the tree. When no such edges exist any more, we either must have a spanning tree already or the original graph was not connected. We can determine this, for example, by checking if there are any nodes left in the graph that haven't been added to the tree.

This algorithm has two kinds of steps, so its representation in linear logic has two rules. The first step moves an arbitrary node from the graph to the tree.

$$\frac{\mathsf{node}(x, \mathsf{g})}{\mathsf{node}(x, \mathsf{t})}\ \mathsf{start}?$$

This rule can be used only once, at the very beginning of the algorithm and must be prohibited afterwards, or we could just use it to move all nodes from the graph to the tree without moving any edges. So we can either say the rule must be ephemeral itself, or we create a new ephemeral proposition init which only exists in the initial state and is consumed by the first step.

$$\frac{\mathsf{init} \quad \mathsf{node}(x, \mathsf{g})}{\mathsf{node}(x, \mathsf{t})}\ \mathsf{start}$$

The next rule implements the idea we described in the text above. All propositions are ephemeral, so we can implement "*a node y not yet in the tree*" by checking whether it is still in the graph, thereby consuming it.

$$\frac{\mathsf{node}(x, \mathsf{t}) \quad \mathsf{edge}(x, y, \mathsf{g}) \quad \mathsf{node}(y, \mathsf{g})}{\mathsf{node}(x, \mathsf{t}) \quad \mathsf{edge}(x, y, \mathsf{t}) \quad \mathsf{node}(y, \mathsf{t})}\ \mathsf{move}$$

A proof using these two rules describes a particular sequence of moves, taking edges from the graph and adding them to the spanning tree.

In order to convince ourselves that this is correct, it is important to understand the state invariants. Initially, we have

init	
$\mathsf{node}(x, \mathsf{g})$	for every node x in g
$\mathsf{edge}(x, y, \mathsf{g})$	for every edge from x to y in g

Rule move does not apply, because we do not yet have a node in t, so any inference must begin with rule start, consuming init and producing one

node x_0 in t.

$\mathsf{node}(x_0, \mathsf{t})$	for some node x_0
$\mathsf{node}(x, \mathsf{g})$	for every node $x \neq x_0$ in g
$\mathsf{edge}(x, y, \mathsf{g})$	for every edge from x to y in g

Now rule start can no longer be applied, and we apply move as long as we can. The rule preserves the invariant that each node x from the initial graph is either in t ($\mathsf{node}(x, \mathsf{t})$) or in g ($\mathsf{node}(x, \mathsf{g})$). It further preserves the invariant that each edge in the original graph is either in t ($\mathsf{edge}(x, y, \mathsf{t})$) or still in g ($\mathsf{edge}(x, y, \mathsf{g})$).

If the algorithm stops and no nodes are left in g, we must have moved all n nodes originally in g. One is moved in the start rule, and $n - 1$ are moved in applications of the move rule. In every application of the move rule we also move exactly one edge from g to t, so t now has n nodes and $n - 1$ edges. Further, it is connected since anytime we move an edge it connects to something already in the partial spanning tree. A connected graph with n nodes and $n - 1$ edges must be a tree, and it spans g because it has all the nodes of g.

If the algorithms stops and there are some nodes left in g, then the original graph must have been disconnected. Assume that g is connected, y is left in g, and we started with x_0 in the first step. Because g is connected, there must be a path from x_0 to y. We prove that this is impossible by induction on the structure of this path. The last edge connects some node y' to y. If y' is in the tree, then the rule move would apply, but we stipulated that the algorithm only stops if move does not apply. If y' is in the graph but not in the tree, then we apply the induction hypothesis to the subpath from x_0 to y'.

2 Example: Counting in Binary

In this section we see how to encode binary counting via ordered inference as in the Lambek calculus. We represent a binary number 1011_2 (which is eleven) by the following ordered propositions:

eps b1 b0 b1 b1

where b1 represents bit 1, b0 represents the bit 0, and eps represents the empty string, thereby marking the end of the binary string. We think of increment as another proposition we add at the *right end* of the string. For

example, if we want to increment the number above twice, we would write

```
eps b1 b0 b1 b1 inc inc
```

If we define the correct rules we would like to infer

```
eps b1 b0 b1 b1 inc inc
⋮
eps b1 b1 b0 b1
```

Before you turn the page you might consider if you can define ordered inference rules to define the increment operation.

We need the following three rules:

$$\frac{\mathsf{b0\ inc}}{\mathsf{b1}}\ \mathsf{inc0} \qquad \frac{\mathsf{b1\ inc}}{\mathsf{inc\ b0}}\ \mathsf{inc1} \qquad \frac{\mathsf{eps\ inc}}{\mathsf{eps\ b1}}\ \mathsf{inceps}$$

The inc1 rule implements the carry bit by incrementing the remainder of the bit string, while inceps deposits the carry as the highest bit in case we have reached the end of the bit string.

These rules encode some parallelism. For example, after a single step of inference we have

eps b1 b0 b1 b1 inc inc
eps b1 b0 b1 inc b0 inc
⋮
eps b1 b1 b0 b1

Here we only show the state after each inference and not the rule used (which is inc1) for the sake of conciseness. In the second line, we can apply inc1 or inc0 or (because they are independent) both of them simultaneously, which gives us

eps b1 b0 b1 b1 inc inc
eps b1 b0 b1 inc b0 inc
eps b1 b0 inc b0 b1
⋮
eps b1 b1 b0 b1

Now we can obtain the desired conclusion with one more step of inference.

eps b1 b0 b1 b1 inc inc
eps b1 b0 b1 inc b0 inc
eps b1 b0 inc b0 b1
eps b1 b1 b0 b1

3 Ordered Hypothetical Judgments

The notion of grammatical inference represents parsing as the process of constructing a proof. For example, if we have a phrase (= sequence of words) $w_1 \dots w_n$ we find their syntactical types $x_1 \dots x_n$ (guessing if necessary if they are ambiguous) and then set up the problem

$$(w_1 : x_1) \cdots (w_n : x_n)$$
$$\vdots$$
$$? : s$$

where "?" will represent the parse tree as a properly parenthesized expression (assuming, of course, we can find a proof).

So far, we can only represent the inference itself, but not the *goal* of parsing a whole sentence. In order to express that we introduce *hypothetical judgments* as a new primitive concept. The situation above is represented as

$$(w_1 : x_1) \cdots (w_n : x_n) \vdash ? : s$$

or, more generally, as

$$(p_1 : x_1) \cdots (p_n : x_n) \vdash r : z$$

The turnstile symboxl "$\vdash$" here separates the *succedent* $r : z$ from the *antecedents* $p_i : x_i$. We sometimes call the left-hand side the *context* or the *hypotheses* and the right-hand side the *conclusion*. Calling the succedent a conclusion is accurate in the sense that it is the conclusion of a hypothetical deduction, but it can also be confusing since we also used "conclusions" to describe what is below the line in a rule of inference. We hope it will always be clear from the situation which of these we mean.

Since we are studying ordered inference right now, the antecedents that form the context are intrinsically ordered. When we want to refer to a sequence of such antecedents we write Ω where "Omega" is intended to suggest "Order". When we capture other forms of inference like linear inference we will revisit this assumption.

4 Inference with Sequents: Looking Left

Now that we have identified hypothetical judgments, written as sequents $\Omega \vdash r : z$, we should examine what this means for our logical rules of inference. Fortunately, we have had only two connectives, *over* and *under*, first shown here without the proof terms (that is, without the parse trees):

$$\frac{x \,/\, y \quad y}{x}\ \text{over} \qquad \frac{y \quad y \setminus x}{x}\ \text{under}$$

Now that the propositions we know appear as antecedents, the direction of the rules appears to be reversed when considered on sequents.

$$\frac{\Omega_L\ x\ \Omega_R \vdash z}{\Omega_L\ (x \,/\, y)\ y\ \Omega_R \vdash z}\ /L^* \qquad \frac{\Omega_L\ x\ \Omega_R \vdash z}{\Omega_L\ y\ (y \setminus x)\ \Omega_R \vdash z}\ \setminus L^*$$

We have written Ω_L and Ω_R to indicate the rest of the context, which remains unaffected by the inference. These rules operate on the left of the turnstile, that is, on antecedents, and we have therefore labeled them $/L^*$ and $\backslash L^*$, pronounced *over left* and *under left*. While helpful for today's lecture, we will have to revise these rules at the beginning of the next lecture, so we have marked them with an asterisk to remind us that they are only preliminary.

Redecorating the rules with proof terms (that is, parse trees in grammatical inference):

$$\frac{p : x \,/\, y \quad q : y}{(p\,q) : x}\ \mathsf{over} \qquad \frac{q : y \quad p : y \setminus x}{(q\,p) : x}\ \mathsf{under}$$

Now that the propositions we know appear as antecedents, the direction of the rules appears to be reversed when considered on sequents.

$$\frac{\Omega_L\ ((p\,q) : x)\ \Omega_R \vdash r : z}{\Omega_L\ (p : x \,/\, y)\ (q : y)\ \Omega_R \vdash r : z}\ /L^* \qquad \frac{\Omega_L\ ((q\,p) : x)\ \Omega_R \vdash r : z}{\Omega_L\ (q : y)\ (p : y \setminus x)\ \Omega_R \vdash r : z}\ \backslash L^*$$

Our inferences, now taking place on the antecedent, take us upward in the tree, so when we have a situation such as

$$\begin{array}{c}(w_1 : x_1) \cdots (w_n : x_n) \\ \vdots \\ p : s\end{array}$$

where we *have* deduced $p : s$, we are now in the situation

$$\begin{array}{c}p : s \vdash \mathsf{?} : s \\ \vdots \\ (w_1 : x_1) \cdots (w_n : x_n) \vdash \mathsf{?} : s\end{array}$$

This means we need one more rule to complete the proof and signal the success of a hypothetical proof. Both forms with and without the proof terms should be self-explanatory. We use id (for *identity*) to label this inference.

$$\frac{}{x \vdash x}\ \mathsf{id}_x \qquad \frac{}{p : x \vdash p : x}\ \mathsf{id}_x$$

Because we wanted to represent the goal of parsing a sequence of words as complete sentence, no additional antecedents besides x are permitted in this rule. Otherwise, a phrase such as *Bob likes Alice likes* could be incorrectly seen to parse as the sentence *((Bob likes) Alice)* ignoring the second *likes*.

5 Inference with Sequents: Looking Right

We already noted in Lecture 1 that $x \setminus (y \,/\, z)$ should be somehow equivalent to $(x \setminus y) \,/\, z$ since both yield a y when given and x to the left and z to the right. Setting this equivalence up as two hypothetical judgments

$$x \setminus (y \,/\, z) \vdash (x \setminus y) \,/\, z$$

and

$$(x \setminus y) \,/\, z \vdash x \setminus (y \,/\, z)$$

that we are trying to prove however fails. No inference is possible. We are lacking the ability to express when we can deduce a *succedent* with a logical connective. Lambek states that we should be able to deduce

$$\frac{z}{x \,/\, y} \qquad \text{if} \qquad \frac{z \quad y}{x}$$

So $x \,/\, y$ should follow from z if we get x if we put y to the right of z. With pure inference, as practiced in the last lecture, we had no way to turn this "*if*" into form of inference rule. However, armed with hypothetical judgments it is not difficult to express precisely this:

$$\frac{z \; y \vdash x}{z \vdash x \,/\, y}$$

Instead of a single proposition z we allow a context, so we write this

$$\frac{\Omega \; y \vdash x}{\Omega \vdash x \,/\, y} \; /R$$

This is an example of a *right rule*, because it analyzes the structure of a proposition in the succedent and we pronounce it as *over right*. The $\setminus R$ (*under right*) rule can be derived analogously.

$$\frac{y \; \Omega \vdash x}{\Omega \vdash y \setminus x} \; \setminus R$$

In the next lecture we will look at the question how we know that these rules are correct. For example, we might have accidentally swapped these two rules, in which case our logic would somehow be flawed. And, in fact, our rules are already flawed but we do not have the tools yet to see this.

Let's come back to the motivating example and try to construct a proof of

$$x \setminus (y \,/\, z) \vdash (x \setminus y) \,/\, z$$

Remember, all the rules work bottom-up, either on some antecedent (a left rule) or on the succedent (a right rule). No left rule applies here (there is no x to the left of $x \setminus (\ldots)$) but fortunately the $/R$ rule does.

$$\frac{x \setminus (y \,/\, z) \quad z \vdash x \setminus y}{x \setminus (y \,/\, z) \vdash (x \setminus y) \,/\, z}\, /R$$

Again, no left rule applies (the parentheses are in the wrong place) but a right rule does.

$$\dfrac{\dfrac{x \quad x \setminus (y \,/\, z) \quad z \vdash x}{x \setminus (y \,/\, z) \quad z \vdash x \setminus y}\, \setminus R}{x \setminus (y \,/\, z) \vdash (x \setminus y) \,/\, z}\, /R$$

Finally, now a left rule applies.

$$\dfrac{\dfrac{\dfrac{y \,/\, z \quad z \vdash y}{x \quad x \setminus (y \,/\, z) \quad z \vdash y}\, \setminus L^*}{x \setminus (y \,/\, z) \quad z \vdash x \setminus y}\, \setminus R}{x \setminus (y \,/\, z) \vdash (x \setminus y) \,/\, z}\, /R$$

One more left rule, and then we can apply identity.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{y \vdash y}\, \mathsf{id}_y}{y \,/\, z \quad z \vdash y}\, /L^*}{x \quad x \setminus (y \,/\, z) \quad z \vdash y}\, \setminus L^*}{x \setminus (y \,/\, z) \quad z \vdash x \setminus y}\, \setminus R}{x \setminus (y \,/\, z) \vdash (x \setminus y) \,/\, z}\, /R$$

The proof in the other direction is similar and left as Exercise 5.

We have left out the proof terms here, concentrated entirely on the logical connectives. We will return to proof terms for ordered hypothetical judgment in a future lecture and proceed to conjecture some logical connectives and how to define them via their left and right rules.

6 Alternative Conjunction

As already mentioned in the last lecture, some words have more than one syntactic type. For example, *and* has type $s \setminus s / s$ (omitting parentheses now since the two forms are equivalent by the reasoning the previous section) and also type $n \setminus n^* / n$, constructing a plural noun from two singular ones. We can combine this into a single type $x \,\&\, y$, pronounced x *with* y:

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

Then, in a deduction, we are confronted with a choice between the two for every occurrence of *and*. For example, in typing *Alice and Bob work and Eve likes Alice*, we choose $n \setminus n^* / n$ for the first occurence of *and*, and $s \setminus s / s$ for the second.

Lambek did not explicitly define this connective, but it would be defined by the rules

$$\frac{x \,\&\, y}{x}\ \mathsf{with}_1 \qquad \frac{x \,\&\, y}{y}\ \mathsf{with}_2$$

In the proof term we might write .1 for the first meaning and .2 for the second meaning of the word.

$$\frac{p : x \,\&\, y}{p.1 : x}\ \mathsf{with}_1 \qquad \frac{p : x \,\&\, y}{p.2 : y}\ \mathsf{with}_2$$

so that the parse tree for the sentence above might become

((Alice and.1 Bob) work) and.2 (Eve likes Bob)

where we have omitted parentheses that are redundant due to the associativity of $\setminus - /$.

As before, these rules turn into left rules in the sequent calculus, shown here only without the proof terms.

$$\frac{\Omega_L\ x\ \Omega_R \vdash z}{\Omega_L\ x \,\&\, y\ \Omega_R \vdash z}\ \&L_1 \qquad \frac{\Omega_L\ y\ \Omega_R \vdash z}{\Omega_L\ x \,\&\, y\ \Omega_R \vdash z}\ \&L_2$$

To derive the right rule we must ask ourselves under which circumstances we could use a proposition both as an x and as a y. That's true, if we can show both.

$$\frac{\Omega \vdash x \quad \Omega \vdash y}{\Omega \vdash x \,\&\, y}\ \&R$$

7 Concatenation

In a sequent, there are multiple antecedents (in order!) but only one succedent. So how could we encode the goal we had in the binary counting example:

$$\begin{array}{l}\mathsf{eps\ b1\ b0\ b1\ b1\ inc\ inc}\\ \vdots\\ \mathsf{eps\ b1\ b1\ b0\ b1}\end{array}$$

Clearly, this is a hypothetical judgment but the succedent is not a single proposition. In order to define over and under, it is important to maintain a single succedent, so we need to define a new connective that expresses adjacency as a new proposition. We write $x \bullet y$ (read: x *fuse* y). In the Lambek calculus, we would simply write

$$\frac{x \bullet y}{x\ y}\ \mathsf{fuse}$$

As a left rule, this is simple turned upside down and becomes

$$\frac{\Omega_L\ x\ y\ \Omega_R \vdash z}{\Omega_L\ x \bullet y\ \Omega_R \vdash z}\ \bullet L$$

As a right rule for $x \bullet y$, we have to divide the context into to segments, one proving x and the other proving y.

$$\frac{\Omega_1 \vdash x \quad \Omega_2 \vdash y}{\Omega_1\ \Omega_2 \vdash x \bullet y}\ \bullet R$$

Note that there is some nondeterminism in this rule if we decide to use it to prove a sequent, because we have to decide *where* to split the context $\Omega = (\Omega_1\ \Omega_2)$. For a context with n propositions there are $n + 1$ possibilities. For example, if we want to express that a phrase represented by Ω is parsed into *two sentences* we can prove the hypothetical judgment

$$\Omega \vdash s \bullet s$$

We can then prove

$$\begin{array}{cccccccc} \textit{Alice} & \textit{works} & \textit{Bob} & \textit{sleeps} & & & ? \\ : & : & : & : & & & : \\ n & n \setminus s & n & n \setminus s & \vdash & & s \bullet s \end{array}$$

but we have to split the phrase exactly between *works* and *Bob* so that both premises can be proved. Assuming a notation of $p \cdot q : x \bullet y$ if $p : x$ and $q : y$, the proof term for $s \bullet s$ in this example would be $(\textit{Alice works}) \cdot (\textit{Bob sleeps})$.

8 Emptiness

In this section we consider $\mathbf{1}$, the unit of concatenation, which corresponds to the empty context. The left and right rules are nullary versions of the binary concatenation. In particular, there must be no antecedents in the right rule for $\mathbf{1}$.

$$\frac{}{\vdash \mathbf{1}}\ \mathbf{1}R \qquad \frac{\Omega_L\ \Omega_R \vdash z}{\Omega_L\ \mathbf{1}\ \Omega_R \vdash z}\ \mathbf{1}L$$

9 An Unexpected Incompleteness

In functional programming there is a pattern called *currying* which says that instead of a functions of type $(\tau \times \sigma) \to \rho$, passing a pair with values of type τ and σ, we can pass the arguments sequentially as indicated by the type $\tau \to (\sigma \to \rho)$. Logically, this is manifested by the isomorphism between these two types when considered as propositions, where $\times$ is conjunction and $\to$ is implication.

Is there something similar in ordered logic? First we note that *over* and *under* are a form of implication, distinguished merely by whether they expect their argument on the left or on the right. Concatenation is a form of conjunction since it puts together two proofs. Let's consider $(x \bullet y) \setminus z$. This expect x followed by y to its left and concludes z. Similarly, $y \setminus (x \setminus z)$ expects a y to its left and then an x next to that which, if you had concatenated them together, would be $x \bullet y$. So these two seem like they should be intuitively equivalent. Let's try to use the tools of logic to prove that.

First, $y \setminus (x \setminus z) \vdash (x \bullet y) \setminus z$. We show here the completed proof, but you should view it step by step *going upward* from the conclusion.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{z \vdash z}\ \mathsf{id}_z}{x \quad x \setminus z \vdash z}\ \setminus L^*}{x \quad y \quad y \setminus (x \setminus z) \vdash z}\ \setminus L^*}{x \bullet y \quad y \setminus (x \setminus z) \vdash z}\ \bullet L}{y \setminus (x \setminus z) \vdash (x \bullet y) \setminus z}\ \setminus R$$

Now for the other direction. Unfortunately, this does not go as well.

$$\dfrac{\dfrac{\overset{?}{x \quad y \quad (x \bullet y) \setminus z \vdash z}}{y \quad (x \bullet y) \setminus z \vdash x \setminus z}\setminus R}{(x \bullet y) \setminus z \vdash y \setminus (x \setminus z)}\setminus R$$

The sequent at the top should be intuitively provable, since we should be able to combine x and y to $x \bullet y$ and then use the $\setminus L^*$ rule, but there is no such rule. All rules in the sequent calculus so far *decompose* connectives in the antecedent (left rules) or succedent (right rules), but here we would like to *construct* a proof of a compound proposition. We could add an ad hoc rule to handle this situation, but how do we know that the resulting system does not have other unexpected sources of incompleteness?

In the next lecture we will first fix this problem and then systematically study how to ensure that our inference rules do not exhibit similar problems.

Exercises

Exercise 1 Consider variations of the representation and rules in the spanning tree example from Section 1. Consider all four possibilities of nodes and edges in g and t being ephemeral or persistent. In each case show the form of the three rules in question: sym (possibly with two variants), start, and move, indicate if the modification would be correct, and spell out how to check if a proper spanning tree has been built in the final state.

Exercise 2 Consider the encoding of binary numbers in ordered logic as in Section 2. Assume a new proposition par for *parity* and write rules so that the binary representation of a number followed by par computes eps b0 or eps b1 if we have an even or odd number of ones, respectively.

Exercise 3 Represent the computation of a Turing machine using ordered inference as in Section 2. You will need to decide on a finite, but potentially unbounded representation of the tape, the finite number of states, and the state transitions, such that each step of the Turing machine corresponds to one or more steps of ordered inference. Make sure to describe all parts of the encoding carefully.

Exercise 4 Represent instances of Post correspondence problem in ordered logic so that ordered inference as in Section 2 from an initial state proves a distinguished proposition s (for success) if and only if the problem has a solution. One should be able to extract the actual solution from the proof. Make sure to describe all parts of the encoding carefully.

Exercise 5 Prove $(x \setminus y) \,/\, z \vdash x \setminus (y \,/\, z)$

Exercise 6 Find equivalences along the lines of associativity, currying, or distributivity laws as in the first two examples and prove both directions. Note (but do not prove) where they don't seem to exist if we restrict ourselves to the over, under, fuse, and with connectives. You may need to refer to Lecture 3 to use the stronger versions of $\setminus L$ and $/L$ that resolve the incompleteness in Section 9.

$(x \setminus y) \,/\, z \dashv\vdash x \setminus (y \,/\, z)$ (see Section 5 and Exercise 5)

$(x \bullet y) \setminus z \dashv\vdash y \setminus (x \setminus z)$ (see Section 9 and Lecture 3)

1. $x \,/\, (y \bullet z) \dashv\vdash A(x, y, z)$

2. $(x \bullet y) \,/\, z \dashv\vdash B(x, y, z)$
3. $(x \,\&\, y) \,/\, z \dashv\vdash C(x, y, z)$
4. $x \,/\, (y \,\&\, z) \dashv\vdash D(x, y, z)$

References

[Gen35] 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.

[Lam58] Joachim Lambek. The mathematics of sentence structure. *The American Mathematical Monthly*, 65(3):154–170, 1958.

Lecture Notes on Harmony

15-816: Substructural Logics
Frank Pfenning

Lecture 3
September 6, 2016

In the last lecture we started developing a sequent calculus which expresses the notion of a *hypothetical judgment* $\Omega \vdash z$ where Ω is a ordered collection of antecendents and z is the succedent. Each connective (we discussed $x \,/\, y$, $x \setminus y$, $x \bullet y$, $\mathbf{1}$ and $x \,\&\, y$) was defined by a right rule (which shows how to prove such a proposition) and a left rule (which shows how to use such a proposition).

We wrote down rules that seemed intuitively correct, but we were surprised we couldn't prove $(x \bullet y) \setminus z \vdash y \setminus (x \setminus z)$ with our rules. We need some criteria to decide if the rules are "correct". In traditional development of classical logic this is usually done by developing a Tarskian semantics, interpreting the propositions in some mathematical domain, and then assessing whether the rules are sound and complete with respect to this interpretation.

An alternative approach views the left and right rules of the sequent calculus themselves as providing the meaning of the connectives, a so-called *proof-theoretic semantics*. This idea was pioneered by Gentzen [Gen35], developed further by Dummett [Dum91], and proposed as the foundation of type theory by Martin-Löf [ML83]. We follow this approach here in the definition of substructural logic and the sequent calculus, rather than intuitionistic logic and natural deduction.

We provide two tests to verify if the left and right rules for a connective are in *harmony*, which permits us to view the rules as a semantic definition. These criteria not only supply *internal* notions of soundness and completeness, but they will also play a critical role later on, when we introduce computational interpretations of proofs.

1 Repairing the Rules for $y \setminus x$

Recall the relevant preliminary rules from Lecture 2.

$$\frac{}{x \vdash x}\ \mathsf{id}_x$$

$$\frac{y\ \Omega \vdash x}{\Omega \vdash y \setminus x}\ \setminus R \qquad \frac{\Omega_L\ x\ \Omega_R \vdash z}{\Omega_L\ y\ (y \setminus x)\ \Omega_R \vdash z}\ \setminus L^*$$

$$\frac{\Omega \vdash x \quad \Omega' \vdash y}{\Omega\ \Omega' \vdash x \bullet y}\ \bullet R \qquad \frac{\Omega_L\ x\ y\ \Omega_R \vdash z}{\Omega_L\ (x \bullet y)\ \Omega_R \vdash z}\ \bullet L$$

Read from the conclusion to the premise, each of the right and left rules removes a connective from the sequent, so they analyze the structure of the proposition in the endsequent.

When we were trying to prove $(x \bullet y) \setminus z \vdash y \setminus (x \setminus z)$ we got stuck at the following point, with no further rule applicable. As usual in the sequent calculus, you should read this partial derivation from the bottom upwards.

$$\dfrac{\dfrac{\begin{array}{cc} x & y \quad (x \bullet y) \setminus z \vdash z \end{array}}{\begin{array}{cc} y & (x \bullet y) \setminus z \vdash x \setminus z \end{array}}\ \setminus R}{(x \bullet y) \setminus z \vdash y \setminus (x \setminus z)}\ \setminus R$$

At this point, there are a couple of directions we could go in. One is to add new rules, another one is to generalize the rules we already have. One suggestion would be to add the rule

$$\frac{\Omega_L\ (x \bullet y)\ \Omega_R \vdash z}{\Omega_L\ x\ y\ \Omega_R \vdash z}$$

This would allow us to complete this proof because we could combine x and y so that $\setminus L^*$ then applies. This is a valid direction to consider (and there are solutions along these lines in other contexts), but a calculus with this rule no longer breaks down connectives as we go up the proof. If we see the rules as a semantic definition, they would now be problematic since the meaning would no longer be compositional but may depend on other propositions that we did not anticipate. We also need to modify our simple argument for decidability. Finally, it seems like a very special case: how do we know we have added enough rules?

A second approach is to generalize the rule

$$\frac{\Omega_L\ x\ \Omega_R \vdash z}{\Omega_L\ y\ (y \setminus x)\ \Omega_R \vdash z}\ \setminus L^*$$

Instead of requiring the proposition to the left of $(y \setminus x)$ to match y *exactly*, all we need is that we can *prove* y from some of the antecedents. Since order is important, we slice off a section just to the left of $(y \setminus x)$ for this purpose to arrive at the following rule.

$$\frac{\Omega' \vdash y \quad \Omega_L\ x\ \Omega_R \vdash z}{\Omega_L\ \Omega'\ (y \setminus x)\ \Omega_R \vdash z}\ \setminus L$$

A pleasing aspect of this rule is that it just breaks down a single connective, so it fits within our general program of right and left rules and meaning explanations of propositions. We will therefore adopt it. Spoiler alert: it will pass the tests for harmony we devise in the next section. Moreover, the previous rule $\setminus L^*$ can easily be justified as a *derived rule of inference*:

$$\frac{\dfrac{}{y \vdash y}\ \mathsf{id}_y \qquad \Omega_L\ x\ \Omega_R \vdash z}{\Omega_L\ y\ (y \setminus x)\ \Omega_R \vdash z}\ \setminus L$$

Finally, the rule is strong enough so we can complete the proof in our motivating example:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x \vdash x}\ \mathsf{id}_x \quad \dfrac{}{y \vdash y}\ \mathsf{id}_y}{x \quad y \vdash x \bullet y}\ \bullet R \quad \dfrac{}{z \vdash z}\ \mathsf{id}_z}{x \quad y \quad (x \bullet y) \setminus z \vdash z}\ \setminus L}{y \quad (x \bullet y) \setminus z \vdash x \setminus z}\ \setminus R}{(x \bullet y) \setminus z \vdash y \setminus (x \setminus z)}\ \setminus R$$

2 Identity and Cut

The key idea behind the tests we devise on the harmony between left and right rules is that they ensure agreement between *proving a proposition* (the right rule) and *using a proposition* (the left rule). How can we embody these

principles in the sequent calculus? Actually, we already have one rule that embodies the second one, name the identity rule.

$$\frac{}{x \vdash x}\ \mathsf{id}_x$$

It expresses that if we have an antecedent x (and nothing else) we can use it to prove the succedent x.

The other direction would say that if we can *prove* x we are justified to *use* x. In its simplest form, this would be

$$\frac{\vdash x \quad x \vdash z}{\vdash z}$$

This is too restrictive, since we should be able to use x even if it requires some of the antecedents in the conclusion. Ordering constraints mean what is used to prove x should be some segment of our context.

$$\frac{\Omega' \vdash x \quad \Omega_L\ x\ \Omega_r \vdash z}{\Omega_L\ \Omega'\ \Omega_R \vdash z}\ \mathsf{cut}_x$$

If, for the whole logical system, the left and right rules are in balance, we should never need identity or cut. That's actually not quite true: if we have propositional variables like x or y because we perform schematic inference (as we have been doing in much of the development), then identity for variables cannot be eliminated. These two properties are known as *identity elimination* and *cut elimination*.

Unfortunately, cut and identity elimination are global properties of a complete logical system, not isolated questions about the individual left and right rules. In the next lecture we will proceed to prove, as metatheorems about the Lambek calculus, that these two properties actually hold. In this lecture we will focus on isolating local transformations on proofs which we call *identity expansion* and *cut reduction* which are isolated checks on the left and right rules for each connective. Cut reduction in particular will also play a fundamental role in the computational interpretation of proofs we will discuss later in the course.

Consider a proposition $x * y$ for some connective $*$. We say it satisfies *identity expansion* if we can replace the identity at $x * y$ by uses of identities at x and y. Conversely, we say it satisfies *cut reduction* if we can replace any cut at $x*y$ that matches a right rule for $x*y$ against the left rule on the same proposition by cuts at x and y. In the next few sections we will check these property for some connectives and we will also look for counterexamples to understand what may happen if these properties are not satisfied.

3 Harmony for $y \setminus x$

We'll start with $y \setminus x$. Identity expansion is easier to check. Can we *expand* the one-line proof

$$\frac{}{y \setminus x \vdash y \setminus x}\ \mathsf{id}_{y\setminus x}$$

into a proof just using identity at x and y? There are only two possible rules that could apply, but $\setminus L$ will fail since we cannot prove y. So wee need to start with $\setminus R$.

$$\frac{\begin{array}{cc} y & \vdots \\ & y \setminus x \vdash x \end{array}}{y \setminus x \vdash y \setminus x}\ \setminus R$$

Fortunately, at this point we can use $\setminus L$ followed by identities to complete the proof.

$$\frac{\dfrac{}{y \vdash y}\ \mathsf{id}_y \quad \dfrac{}{x \vdash x}\ \mathsf{id}_x}{\dfrac{\begin{array}{cc} y & y \setminus x \vdash x \end{array}}{y \setminus x \vdash y \setminus x}\ \setminus R}\ \setminus L$$

Interesting, the restricted rule $\setminus L^*$ would have actually passed this test. It might have been slightly suspicious, though, because the expanded form does not need id_y.

Note that the expansion *introduces* uses of $\setminus R$ and $\setminus L$ into the proof. Cut reduction instead *eliminates* uses of these rules that appear just above the cut. The situation:

$$\frac{\dfrac{\begin{array}{c}\mathcal{D}\\ y\ \Omega \vdash x\end{array}}{\Omega \vdash y \setminus x}\ \setminus R \quad \dfrac{\begin{array}{cc}\mathcal{E}' & \mathcal{E}\\ \Omega' \vdash y & \Omega_L\ x\ \Omega_R \vdash z\end{array}}{\Omega_L\ \Omega'\ (y \setminus x)\ \Omega_R \vdash z}\ \setminus L}{\Omega_L\ \Omega'\ \Omega\ \Omega_R \vdash z}\ \mathsf{cut}_{y\setminus x}$$

We have named here the subproofs, $\mathcal{D}$, $\mathcal{E}'$, and $\mathcal{E}$, since these are the proofs we can now use to justify the conclusion, using cut only at y and x. Indeed, we can first cut $\mathcal{E}'$ with $\mathcal{D}$ and then the result with $\mathcal{E}$. Note that we could

also cut $\mathcal{D}$ with $\mathcal{E}$ and then the result with $\mathcal{E}'$. We show the first alternative.

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{E}'\\ \Omega' \vdash y\end{array} \quad \begin{array}{c}\mathcal{D}\\ y\ \Omega \vdash x\end{array}}{\Omega'\ \Omega \vdash x}\ \mathsf{cut}_y \qquad \begin{array}{c}\mathcal{E}\\ \Omega_L\ x\ \Omega_R \vdash z\end{array}}{\Omega_L\ \Omega'\ \Omega\ \Omega_R \vdash z}\ \mathsf{cut}_x$$

At this point we know we have established harmony. To summarize, we have the identity expansion, denoted by $\Longrightarrow_E$

$$\dfrac{}{y \setminus x \vdash y \setminus x}\ \mathsf{id}_{y\setminus x} \quad \Longrightarrow_E \quad \dfrac{\dfrac{\dfrac{}{y \vdash y}\ \mathsf{id}_y \quad \dfrac{}{x \vdash x}\ \mathsf{id}_x}{y \quad y \setminus x \vdash x}\ \setminus L}{y \setminus x \vdash y \setminus x}\ \setminus R$$

and the cut reduction, denoted by $\Longrightarrow_R$

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ y\ \Omega \vdash x\end{array}}{\Omega \vdash y \setminus x}\ \setminus R \quad \dfrac{\begin{array}{c}\mathcal{E}'\end{array} \quad \begin{array}{c}\mathcal{E}\\ \Omega_L\ x\ \Omega_R \vdash z\end{array}}{\Omega_L\ \Omega'\ (y \setminus x)\ \Omega_R \vdash z}\ \setminus L}{\Omega_L\ \Omega'\ \Omega\ \Omega_R \vdash z}\ \mathsf{cut}_{y\setminus x}$$

$$\Longrightarrow_R \quad \dfrac{\dfrac{\begin{array}{c}\mathcal{E}'\\ \Omega' \vdash y\end{array} \quad \begin{array}{c}\mathcal{D}\\ y\ \Omega \vdash x\end{array}}{\Omega'\ \Omega \vdash x}\ \mathsf{cut}_y \qquad \begin{array}{c}\mathcal{E}\\ \Omega_L\ x\ \Omega_R \vdash z\end{array}}{\Omega_L\ \Omega'\ \Omega\ \Omega_R \vdash z}\ \mathsf{cut}_x$$

Let's see what would happen if we had the weaker $\setminus L^*$ rule.

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ y\ \Omega \vdash x\end{array}}{\Omega \vdash y \setminus x}\ \setminus R \quad \dfrac{\begin{array}{c}\mathcal{E}\\ \Omega_L\ x\ \Omega_R \vdash z\end{array}}{\Omega_L\ y\ (y \setminus x)\ \Omega_R \vdash z}\ \setminus L^*}{\Omega_L\ y\ \Omega\ \Omega_R \vdash z}\ \mathsf{cut}_{y\setminus x}$$

$$\Longrightarrow_R \quad \dfrac{\begin{array}{c}\mathcal{D}\\ y\ \Omega \vdash x\end{array} \quad \begin{array}{c}\mathcal{E}\\ \Omega_L\ x\ \Omega_R \vdash z\end{array}}{\Omega_L\ y\ \Omega\ \Omega_R \vdash z}\ \mathsf{cut}_x$$

We note that, perhaps surprisingly, cut reduction can still apply, but that the situation of cut we are considering is not most general, that is, it only applies in the special case of a cut where y happens to be present in its second premise. This would eventually lead to a failure of the global cut elimination property we discuss in the next lecture.

Coming back to the original rules, let's make another (not completely implausible) mistake and consider the incorrect $\setminus R^?$ rule which adds y to the wrong side of the antecedents.

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ \Omega\ y \vdash x\end{array}}{\Omega \vdash y \setminus x}\ \setminus R^? \quad \dfrac{\begin{array}{c}\mathcal{E}'\\ \Omega' \vdash y\end{array} \quad \dfrac{\begin{array}{c}\mathcal{E}\\ \Omega_L\ x\ \Omega_R \vdash z\end{array}}{\Omega_L\ \Omega'\ (y \setminus x)\ \Omega_R \vdash z}\ \setminus L}{\Omega_L\ \Omega'\ \Omega\ \Omega_R \vdash z}\ \mathsf{cut}_{y\setminus x}$$

Now we can try to perform a similar reduction to before, which would give us:

$$\Longrightarrow^?_R \quad \dfrac{\dfrac{\begin{array}{c}\mathcal{E}'\\ \Omega' \vdash y\end{array} \quad \begin{array}{c}\mathcal{D}\\ \Omega\ y \vdash x\end{array}}{\Omega\ \Omega' \vdash x}\ \mathsf{cut}_y \quad \dfrac{\begin{array}{c}\mathcal{E}\\ \Omega_L\ x\ \Omega_R \vdash z\end{array}}{\Omega_L\ \Omega\ \Omega'\ \Omega_R \vdash z}\ \mathsf{cut}_x$$

We note that this proof has a different conclusion from before, swapping Ω' and Ω, so this is not a valid reduction. Indeed, cut reduction fails: we cannot construct, from the proofs $\mathcal{D}$, $\mathcal{E}'$ and $\mathcal{E}$ that we have a proof of the original endsequent $\Omega_L\ \Omega'\ \Omega\ \Omega_R$ using cut only at x and y.

How catastrophic is this failure? See Exercise 2. Suffice it to say here that our test fails. The left and right rules are not in harmony. Indeed, the identity expansion would fail as well:

$$\dfrac{}{y \setminus x \vdash y \setminus x}\ \mathsf{id}_{y\setminus x} \quad \Longrightarrow^?_E \quad \dfrac{\dfrac{\begin{array}{c}??\\ \vdash y\end{array} \quad \begin{array}{c}??\\ x\ y \vdash x\end{array}}{(y \setminus x)\ y \vdash x}\ \setminus L}{y \setminus x \vdash y \setminus x}\ \setminus R^?$$

Each step here is forced, and we can not prove either of the two sequents at the top since no rule applies.

We skip harmony for x/y, which is symmetric (see Exercise 1) and move on to other connectives.

4 Alternative Conjunction

As one more example, let's consider the alternative conjunction $x \& y$. Recall the sequent rules

$$\frac{\Omega \vdash x \quad \Omega \vdash y}{\Omega \vdash x \,\&\, y}\,\&R \qquad \frac{\Omega_L\ x\ \Omega_R \vdash z}{\Omega_L\ x \,\&\, y\ \Omega_R \vdash z}\,\&L_1 \qquad \frac{\Omega_L\ y\ \Omega_R \vdash z}{\Omega_L\ x \,\&\, y\ \Omega_R \vdash z}\,\&L_2$$

The identity expansion is once again straightforward, since essentially every step is forced.

$$\frac{}{x \,\&\, y \vdash x \,\&\, y}\,\mathsf{id}_{x\&y} \quad \Longrightarrow_E \quad \frac{\dfrac{\dfrac{}{x \vdash x}\,\mathsf{id}_x}{x \,\&\, y \vdash x}\,\&L_1 \quad \dfrac{\dfrac{}{y \vdash y}\,\mathsf{id}_y}{x \,\&\, y \vdash y}\,\&L_2}{x \,\&\, y \vdash x \,\&\, y}\,\&R$$

What would happen if, say, we forgot the second left rule, $\&L_2$? We would not be able to complete the proof of the second premise of $\&R$, so the identity expansion would fail. It is perhaps easier to see here than in the case of $x \setminus y$, that identity expansion verifies that, taken together, the left rules for a connective are strong enough to prove the succedent with this same connective. If we omit the second left rule here they are too weak and our test will fail.

For the cut reduction, we actually have to test two situations, since there are two possible left rules to infer $x \,\&\, y$. First, with the first left rule:

$$\frac{\dfrac{\begin{array}{cc}\mathcal{D} & \mathcal{D}' \\ \Omega \vdash x & \Omega \vdash y\end{array}}{\Omega \vdash x \,\&\, y}\,\&R \quad \dfrac{\begin{array}{c}\mathcal{E} \\ \Omega_L\ x\ \Omega_R \vdash z\end{array}}{\Omega_L\ (x \,\&\, y)\ \Omega_R \vdash z}\,\&L_1}{\Omega_L\ \Omega\ \Omega_R \vdash z}\,\mathsf{cut}_{x\&y}$$

We can easily reduce this to a cut between $\mathcal{D}$ and $\mathcal{E}$.

$$\Longrightarrow_R \quad \frac{\begin{array}{cc}\mathcal{D} & \mathcal{E} \\ \Omega \vdash x & \Omega_L\ x\ \Omega_R \vdash z\end{array}}{\Omega_L\ \Omega\ \Omega_R \vdash z}\,\mathsf{cut}_x$$

We don't need a cut on y here: the proof of $x \,\&\, y$ offers a choice between the proof of x and the proof of y and in this case the proof that uses $x \,\&\, y$ chooses x.

Of course, if the second premise of the cut is $\&L_2$, we perform a symmetric reduction, this time to a cut on y. We omit the straightforward deduction.

Note that if we had mistakenly omitted the $\&L_2$ rule, then we would have had only the first case to check, and it would pass. In other words, the left rules are not too strong. In this case, the imbalance can only be noted in the identity expansion.

Another suggested mode of failure would require two copies of x in the left rule. Then the situation would be as follows:

$$\dfrac{\dfrac{\begin{array}{cc}\mathcal{D} & \mathcal{D}' \\ \Omega \vdash x & \Omega \vdash y\end{array}}{\Omega \vdash x \,\&\, y}\,\&R \qquad \dfrac{\begin{array}{c}\mathcal{E} \\ \Omega_L\ x\ x\ \Omega_R \vdash z\end{array}}{\Omega_L\ (x \,\&\, y)\ \Omega_R \vdash z}\,\&L_1^?}{\Omega_L\ \Omega\ \Omega_R \vdash z}\,\mathsf{cut}_{x \& y}$$

A cut reduction would require two cuts on x to eliminate the two copies, but this would duplicate Ω and lead to the wrong endsequent.

$$\Longrightarrow_R^? \quad \dfrac{\begin{array}{c}\mathcal{D} \\ \Omega \vdash x\end{array} \qquad \dfrac{\begin{array}{cc}\mathcal{D} & \mathcal{E} \\ \Omega \vdash x & \Omega_L\ x\ x\ \Omega_R \vdash z\end{array}}{\Omega_L\ \Omega\ x\ \Omega_R \vdash z}\,\mathsf{cut}_x}{\Omega_L\ \Omega\ \Omega\ \Omega_R \vdash z}\,\mathsf{cut}_x$$

So this is *not* a valid cut reduction. The right rule and this modified left rule would not be in harmony. In fact, identity expansion would also fail since we have an extra copy of x in one branch on the proof, which is not allowed in applications of the identity rule.

5 Rule Summary

Here is a summary of the sequent calculus rules for the Lambek calculus so far [Lam58].[1] We often consider the *cut-free* sequent calculus, omitting the cut_x rule, and the *identity expanded* sequent calculus, restricting the id_x rule to propositional variables x.

We refer to id_x and cut_x as *judgmental rules* since they are concerned only with the nature of the ordered hypothetical judgment but not any particular propositions. They are also sometimes called *structural rules*. The other

[1] Actually, Lambek did not have $\mathbf{1}$ or $\&$ as explicit connectives.

rules, namely the right and left rules, are defining propositional connectives so we call them the *propositional rules*.

Judgmental rules

$$\frac{}{x \vdash x}\ \mathsf{id}_x \qquad \frac{\Omega \vdash x \quad \Omega_L\ x\ \Omega_R \vdash z}{\Omega_L\ \Omega\ \Omega_R \vdash z}\ \mathsf{cut}_x$$

Propositional rules

$$\frac{y\ \Omega \vdash x}{\Omega \vdash y \setminus x}\ \setminus R \qquad \frac{\Omega' \vdash y \quad \Omega_L\ x\ \Omega_R \vdash z}{\Omega_L\ \Omega'\ (y \setminus x)\ \Omega_R \vdash z}\ \setminus L$$

$$\frac{\Omega\ y \vdash x}{\Omega \vdash x\ /\ y}\ / R \qquad \frac{\Omega' \vdash y \quad \Omega_L\ x\ \Omega_r \vdash z}{\Omega_L\ (x\ /\ y)\ \Omega'\ \Omega_R \vdash z}\ / L$$

$$\frac{\Omega \vdash x \quad \Omega' \vdash y}{\Omega\ \Omega' \vdash x \bullet y}\ \bullet R \qquad \frac{\Omega_L\ x\ y\ \Omega_R \vdash z}{\Omega_L\ (x \bullet y)\ \Omega_R \vdash z}\ \bullet L$$

$$\frac{}{\vdash \mathbf{1}}\ \mathbf{1}R \qquad \frac{\Omega_L\ \Omega_R \vdash z}{\Omega_L\ \mathbf{1}\ \Omega_R \vdash z}\ \mathbf{1}L$$

$$\frac{\Omega \vdash x \quad \Omega \vdash y}{\Omega \vdash x \,\&\, y}\ \&R \qquad \frac{\Omega_L\ x\ \Omega_R \vdash z}{\Omega_L\ x \,\&\, y\ \Omega_R \vdash z}\ \&L_1 \qquad \frac{\Omega_L\ y\ \Omega_R \vdash z}{\Omega_L\ x \,\&\, y\ \Omega_R \vdash z}\ \&L_2$$

Exercises

Exercise 1 Show identity expansion and cut reduction for $x \mathbin{/} y$.

Exercise 2 As pointed in Section 3, if we replace the $\backslash R$ rule with $\backslash R^?$ which adds y at the wrong end of the antecedents, both identity expansion and cut reduction fail. Explore which, if any, of the three structural rules would now be derivable in the presence of cut and identity. In each case, show the proof if one exists or indicate that you believe it is not derivable (which you do not need to prove).

1. Exchange:
$$\frac{\Omega_L\ y\ x\ \Omega_R \vdash z}{\Omega_L\ x\ y\ \Omega_R \vdash z}\ \mathsf{exchange}$$
2. Weakening:
$$\frac{\Omega_L\ \Omega_R \vdash z}{\Omega_L\ x\ \Omega_R \vdash z}\ \mathsf{weaken}$$
3. Contraction:
$$\frac{\Omega_L\ x\ x\ \Omega_R \vdash z}{\Omega_L\ x\ \Omega_R \vdash z}\ \mathsf{contract}$$

If any of these were derivable, it would quantify the global effect of the lack of harmony for a single connective, upsetting our intended meaning of the logic. If none of these can you find another expression of the failure of semantic intent?

Exercise 3 Assume we define a new connective $*$ with the following right and left rules (which mix the right rule for alternative conjunctions with left rule for concatenation):

$$\frac{\Omega \vdash x \quad \Omega \vdash y}{\Omega \vdash x * y}\ *R \qquad \frac{\Omega_L\ x\ y\ \Omega_R \vdash z}{\Omega_L\ (x * y)\ \Omega_R \vdash z}\ *L$$

First, show which of identity expansion and cut reduction fail and which succeed. Then answer the same questions as in Exercise 2.

Exercise 4 Assume we define a new connective $\#$ with the following right and left rules (which mix the right rule for concatenation with left rules for alternative conjunction):

$$\frac{\Omega_L \vdash x \quad \Omega_R \vdash y}{\Omega_L\ \Omega_R \vdash x \# y}\ \#R \qquad \frac{\Omega_L\ x\ \Omega_R \vdash z}{\Omega_L\ (x \# y)\ \Omega_R \vdash z}\ \#L_1 \qquad \frac{\Omega_L\ y\ \Omega_R \vdash z}{\Omega_L\ (x \# y)\ \Omega_R \vdash z}\ \#L_2$$

First, show which of identity expansion and cut reduction fail and which succeed. Answer the same questions as in Exercise 2.

Exercise 5 Consider a connective $x \circ y$ (pronounced x *twist* y) defined in the original style of Lambek by

$$\frac{x \circ y}{y \quad x}\ \mathsf{twist}$$

Investigate this connective by going through the following steps.

1. Define the right and left rules for $x \circ y$.
2. Verify identity expansion and cut reduction for $x \circ y$.
3. Prove or refute that $x \circ y \vdash y \bullet x$ and $y \bullet x \vdash x \circ y$.
4. Find a curried equivalent $A(x, y, z)$ of $(x \circ y) \setminus z$ and prove $A(x, y, z) \vdash (x \circ y) \setminus z$ and $(x \circ y) \setminus z \vdash A(x, y, z)$.
5. Find a curried equivalent $B(x, y, z)$ of $x \,/\, (y \circ z)$ and prove $B(x, y, z) \vdash x \,/\, (y \circ z)$ and $x \,/\, (y \circ z) \vdash B(x, y, z)$.

Exercise 6 Consider propositions in the Lambek calculus constructed from $x \,/\, y$, $x \setminus y$, $x \bullet y$, $x \circ y$, and $\mathbf{1}$. This calculus should have some strong symmetries. Find a transformation $\overline{x}$ such that $\vdash x$ if and only if $\vdash \overline{x}$ that exhibits such a symmetry and prove that it satisfies this property.

Exercise 7 We have explained logical equivalence between x and y as $x \vdash y$ and $y \vdash x$. Can we internalize logical equivalence as a connective $x \equiv y$? Its defining rules in Lambek's original style would be

$$\frac{x \quad x \equiv y}{y}\ \mathsf{equiv}_1 \qquad \frac{x \equiv y \quad y}{x}\ \mathsf{equiv}_2$$

Answer the following questions if you find this is a proper connective, or explain if no satisfactory rules seem possible.

1. Define right and left rules for $x \equiv y$.
2. Verify identity expansion and cut reduction for $x \equiv y$.
3. Prove or refute that $\equiv$ is reflexive, symmetric, and transitive.
4. If you can define $x \equiv y$ *notationally* as proposition $A(x, y)$ with connectives already present rather than by right and left rules, show that $A \vdash x \equiv y$ and $x \equiv y \vdash A$ with your rules from part 1.

Exercise 8 Prove that $x \bullet \mathbf{1}$ and $\mathbf{1} \bullet x$ are equivalent to x.

References

[Dum91] Michael Dummett. *The Logical Basis of Metaphysics*. Harvard University Press, Cambridge, Massachusetts, 1991. The William James Lectures, 1976.

[Gen35] 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.

[Lam58] Joachim Lambek. The mathematics of sentence structure. *The American Mathematical Monthly*, 65(3):154–170, 1958.

[ML83] 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.

Lecture Notes on Cut Elimination

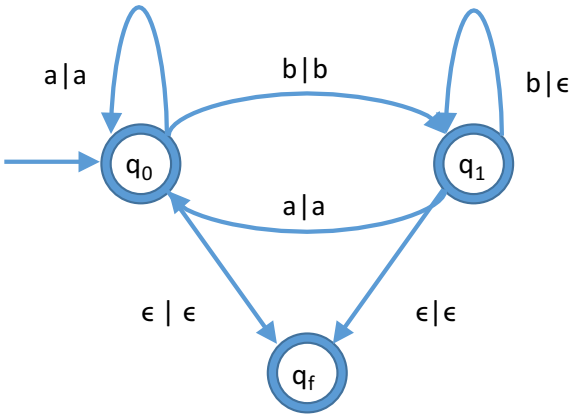
15-816: Substructural Logics
Frank Pfenning

Lecture 4
September 8, 2016

We first present some additional examples illustrating ordered inference that capture computations of finite state transducers and Turing machines. Then we return to cut elimination, also called Gentzen's Hauptsatz [Gen35], for the Lambek calculus given in its sequent formulation. Together with identity elimination, this justifies our program of viewing the inference rules as defining the meaning of the connectives even more clearly than the cut reduction and identity expansion by themselves. The idea behind these will emerge in the proof. Cut elimination also immediately yields a decidability proof for the Lambek calculus, something already observed by Lambek [Lam58].

1 Example: Finite State Transducers

A *subsequential finite-state transducer* [Sch77] (FST) consists of a finite number of states, an input alphabet and an output alphabet, and a transition function δ that takes a state and an input symbol to a new state and an output string. We also distinguish an initial state and a final state, from which no further transitions are possible. Finite state transducers have a number of important closure properties and are closely related to deterministic finite automata (DFAs). They are often depicted with transition diagrams. As an example we show a FST which transforms an input string consisting of a and b symbols by compressing all runs of b into a single b. Each transition is labeled as $x \mid w$ where x is either an input symbol or ϵ (when the input string is empty) and w is a word over the output alphabet.



In order to represent this in the Lambek calculus so that ordered inference corresponds to computation, we introduce propositions a and b to represents the symbols, here shared between the input and output alphabets. We also have a proposition \$ representing an endmarker and *reverse* the word[1]. For example, the string $bbaba$ will be represented as the ordered antecedents \$ a b a b b. Furthermore, we have a new proposition for every state in the FST, here q_0, q_1, and q_f. Initially, our antecedents will be populated by the representation of the input string followed by the initial state. In this example, we start with

$$\$\ a\ b\ a\ b\ b\ q_0$$

We now present inference rules so that each ordered inference corresponds to a transition of the finite state transducer. In the premise we have the input (represented as a proposition) followed by the state; in the conclusion we have the new state followed by the output. The empty input string is represented by \$, which we need to write when we transition into the final state.

$$\frac{a\ q_0}{q_0\ a} \qquad \frac{b\ q_0}{q_1\ b} \qquad \frac{\$\ q_0}{q_f\ \$}$$

$$\frac{a\ q_1}{q_0\ a} \qquad \frac{b\ q_1}{q_1} \qquad \frac{\$\ q_1}{q_f\ \$}$$

Since it is convenient, we add one more inference rule

$$\frac{q_f}{\cdot}$$

[1] for reasons that may nor may not become clear in a future lecture

so that the overall computation with input word w, and initial state q_0 to output v in final state q_f is modeled by inference

$$\frac{\begin{array}{c}\$\ w^R\ q_0 \\ \vdots \\ q_f\ \$\ v^R\end{array}}{\$\ v^R}$$

where s^R represents the reversal of a string s. We could also fold the last step into the rules producing q_f, replacing q_f by the empty context.

You can see why we used an endmarker \$: unlike the usual assumption for finite-state transducers, ordered inference cannot depend on whether it takes place at the end of the context. This is because any ordered inference, by its very definition, applies to any consecutive part of the state. In the sequent calculus this is explicit in all the left rules that have arbitrary Ω_L and Ω_R surrounding the principal proposition of the inference. Trying to restrict this would lead to a breakdown in the sequent calculus (see Exercise 2).

We can use this construction to represent any subsequential finite-state transducer, with one inference rule for every transition. We will not develop the formal details, which are somewhat tedious but straightforward.

We can compose transducers the way we could compose functions. If transducer T_1 transforms input w_0 into w_1 and T_2 transforms w_1 to w_2, then $T_1\ ;\ T_2$ transforms w_0 to w_2. There is a construction on the automata-theoretic descriptions of transducers to show that $T_1\ ;\ T_2$ is indeed another finite-state subsequential transducer if T_1 and T_2 are.

Here, in the setting of ordered inference, we can easily represent the composition of transducers $T_1\ ;\ \ldots\ ;\ T_n$ just by renaming the sets of states apart and then creating the initial state as

$$\$\ w^R\ q_0^1 \ldots q_0^n$$

where q_0^i is the initial state of FST T_i. As T_1 starts to produce output, the configuration will have the form

$$\$\ w_0^R\ q_k^1\ w_1^R\ q_0^2 \ldots q_0^n$$

At this point, T_2 (represented by q_0^2) can start to consume some of its input and produce its output, and so on. Effectively, we have a chain of transducers operating concurrently as long as enough input is available to each of them. Eventually, all of them will end up in their final state and we will end up with the final configuration $\$\ v^R$.

2 Example: Turing Machines

In this section we generalize the construction from the previous section to represent Turing machines. We represent the contents of the unbounded tape of the Turing machine as a *finite* context

$$\$ \ \ldots \ a_{-1} \ q \ a_0 \ a_1 \ \ldots \$$$

with two endmarkers $\$$. The proposition q represents the current state of the machine, and we imagine it "looks to its right" so that the contents of the current cell would be a_0.[2] The initial context for the initial state q_0 is just

$$\$ \ q_0 \ a_0 \ \ldots \ a_n \ \$$$

where $a_0 \ldots a_n$ is the input word written on the tape. Returning to the general case

$$\$ \ \ldots \ a_{-1} \ q \ a_0 \ a_1 \ \ldots \$$$

if the transition function for state q and symbol a_0 specifies to write symbol a_0', transition to state q', and move to the *right*, then the next configuration would be

$$\$ \ \ldots \ a_{-1} \ a_0' \ q' \ a_1 \ \ldots \$$$

This can easily be represented, in general, by the rule

$$\frac{q \ a}{a' \ q'} \ \mathsf{MR}$$

which we call MR for *move right*.

To see how to represent moving to the left, reconsider

$$\$ \ \ldots \ a_{-1} \ q \ a_0 \ a_1 \ \ldots \$$$

If we are supposed to write a_0', transition to q', and move to the left, the next state should be

$$\$ \ \ldots \ q' \ a_{-1} \ a_0' \ a_1 \ \ldots \$$$

The corresponding rule would, using b for a_{-1}:

$$\frac{b \ q \ a}{q' \ b \ a'} \ \mathsf{ML}_b$$

[2] In lecture we were looking to the left, but it is a bit unpleasant to define the initial state in that case.

We would have such a rule for each b in the (fortunately finite) tape alphabet (which excludes the endmarker), or we could represent it schematically

$$\frac{x\ q\ a}{q'\ x\ a'}\ \mathsf{ML}^*$$

except we would have a side condition that $x \neq \$$. We should also have rules that allow us to extend the tape by the designated blank symbol '$\sqcup$' (which is part of the usual definition of Turing machines).

$$\frac{\$\ q\ a}{\$\ q'\ \sqcup\ a'}\ \mathsf{ML}_\$ \qquad \frac{q\ \$}{q\ \sqcup\ \$}\ \mathsf{ER}_q$$

Finally, if we are in a final state q_f from which no further transitions are possible, we can simply eliminate it from the configuration.

$$\frac{q_f}{\cdot}\ \mathsf{F}$$

A somewhat more symmetric and elegant solution allows the tape head in state q (represented by the proposition q) to be looking either right or left, represented by $q \triangleright$ and $\triangleleft\ q$. When we look right and have to move left or vice versa, we just change the direction in which we are looking to implement the move. Then we get the following elegant set of rules, two for each possible transition, two extra ones for extending the tape, and two (if we like) for erasing the final state.

$$\frac{q\ \triangleright\ a}{a'\ q'\ \triangleright}\ \mathsf{LRMR} \qquad \frac{q\ \triangleright\ a}{\triangleleft\ q'\ a'}\ \mathsf{LRML}$$

$$\frac{a\ \triangleleft\ q}{a'\ q'\ \triangleright}\ \mathsf{LLMR} \qquad \frac{a\ \triangleleft\ q}{\triangleleft\ q'\ a'}\ \mathsf{LLML}$$

$$\frac{\triangleright\ \$}{\triangleright\ \sqcup\ \$}\ \mathsf{ER} \qquad \frac{\$\ \triangleleft}{\$\ \sqcup\ \triangleleft}\ \mathsf{EL} \qquad \frac{\triangleleft\ q_f}{\cdot}\ \mathsf{FL} \qquad \frac{q_f\ \triangleright}{\cdot}\ \mathsf{FR}$$

The initial configuration represented by the context

$$\$\ q_0\ \triangleright\ a_1\ \ldots\ a_n\ \$$$

and the final configuration as

$$\$\ b_1\ \ldots\ b_k\ \$$$

and we go from the first to the last by a process of ordered inference.

Of course, a Turing machine may not halt, in which case inference would proceed indefinitely, never arriving at a quiescent state in which no inference is possible.

Our modeling of the Turing machine is here faithful in the sense that each step of the Turing machine corresponds to one inference. There is a small caveat in that we have to extent the tape with an explicit inference, while Turing machines are usually preloaded with a two-way infinite tape with blank symbols on them. But except for those little stutter-steps, the correspondence is exact.

Composition of Turing machines in this representation is unfortunately not as simple as for FSTs since the output is not produced piecemeal, going in one direction, but will be on the tape when the final state is reached. We would have to return the tape head (presumably in the final state) to the left end of the tape and then transition to the starting state of the second machine.

Both for finite-state transducers and Turing machines, nondeterminism is easy to add: we just add multiple rules if there are multiple possible transitions from a state. This works, because the inference process is naturally nondeterministic: any applicable rule can be applied.

We will return to automata and Turing machines in a future lecture when we will look at the problem again from a different perspective.

3 Admissiblity of Cut

We return from the examples to metatheoretic considerations. Our goal in this section and the next is to show that the cut rule can be eliminated from any proof in the ordered sequent calculus. Together with identity elimination in Section 5, this gives us a global version of harmony for our logic and a good argument for thinking of the right and left rules in the sequent calculus as defining the meaning of the connectives.

A key step on the way will be the *admissibility of cut* in the cut-free sequent calculus. We say that a rule of inference is *admissible* if there is a proof of the conclusion whenever there are proofs of all the premises. This is a somewhat weaker requirement that saying that a rule is *derivable*, which means we have a closed-form hypothetical proof of the conclusion given all the premises. Derivable rules remain derivable even if we extend our logic by new propositions and inference rules (once a proof, always a proof), but admissible rules may no longer remain admissible and have to be recon-

sidered.

Since the cut-free sequent calculus will play an important role in this course, we write $\Omega \Vdash x$ for a sequent in the cut-free sequent calculus. We write admissible rules using dashed lines and parenthesized justifications, as in

$$\frac{\Omega \Vdash x \quad \Omega_L\ x\ \Omega_r \Vdash z}{\Omega_L\ \Omega\ \Omega_R \Vdash z}\ (\mathsf{cut}_x)$$

Of course, we have not yet proved that cut is indeed admissible here!

Theorem 1 (Admissibility of Cut)
If $\Omega \Vdash x$ and $\Omega_L\ x\ \Omega_R \Vdash z$ then $\Omega_L\ \Omega\ \Omega_R \Vdash z$.

Proof: We assume we are given $\begin{array}{c}\mathcal{D}\\ \Omega \Vdash x\end{array}$ and $\begin{array}{c}\mathcal{E}\\ \Omega_L\ x\ \Omega_R \Vdash z\end{array}$ and we construct $\begin{array}{c}\mathcal{F}\\ \Omega_L\ \Omega\ \Omega_R \Vdash z\end{array}$.

The proof proceeds by a so-called *nested induction*, first on x and then the proofs $\mathcal{D}$ and $\mathcal{E}$. This means we can appeal to the induction hypothesis when

1. either the cut formula x becomes smaller,
2. or x remains the same, and
   (a) $\mathcal{D}$ becomes smaller and $\mathcal{E}$ stays the same,
   (b) or $\mathcal{D}$ stays the same and $\mathcal{E}$ becomes smaller.

This is also called *lexicographic induction* since it is an induction over a lexicographic order, first considering x and then $\mathcal{D}$ and $\mathcal{E}$.

The idea for this kind of induction can be synthesized from the proof if we observe what constructions take place in each case. We will see that the ideas of the cut reductions in the last lecture will be embodied in the proof cases. We distinguish three kinds of cases based on $\mathcal{D}$ and $\mathcal{E}$.

Identity cases. When one premise or the other is an instance of the identity rule we can eliminate the cut outright. This should be expected since identity ("*if we can use x we may prove x*") and cut ("*if we can prove x we may use x*") are direct inverses of each other.

Principal cases. When the cut formula x is introduced by the last inference in both premises we can reduce the cut to (potentially several) cuts on strict subformulas of A. We have demonstrated this by cut reductions in the last lecture.

Commutative cases. When the cut formula is a side formula of the last inference in either premise, we can appeal to the induction hypothesis on this premise and then re-apply the last inference. These constitute valid appeals to the induction hypothesis because the cut formula and one of the deductions in the premises remain the same while the other becomes smaller.

We now go through representative samples of these cases. First, the two identity cases.

Case: $\mathsf{id} \mathbin{\#} \mathcal{E}$

$$\mathcal{D} = \frac{}{x \Vdash x}\,\mathsf{id}_x \quad \text{and} \quad \begin{array}{c}\mathcal{E}\\ \Omega_L\ x\ \Omega_R \Vdash z\end{array} \quad \text{arbitrary}$$

We have to construct a proof of $\Omega_L\ \Omega\ \Omega_R \Vdash z$, but $\Omega = x$, so we can let $\mathcal{F} = \mathcal{E}$.

Case: $\mathcal{D} \mathbin{\#} \mathsf{id}$

$$\begin{array}{c}\mathcal{D}\\ \Omega \Vdash x\end{array} \quad \text{arbitrary, and} \quad \mathcal{E} = \frac{}{x \Vdash x}\,\mathsf{id}_x$$

We have to construct a proof of $\Omega_L\ \Omega\ \Omega_R \Vdash z$, but $\Omega_L = \Omega_R = (\cdot)$ and $z = x$, so we can let $\mathcal{F} = \mathcal{D}$.

Next we look at a principal case, where the cut proposition x (here $x_1 \,/\, x_2$) was introduced in the last inference in both premises, in which case we say x is the *principal proposition* of the inference.

Case: $/R \mathbin{\#} /L$

$$\mathcal{D} = \frac{\begin{array}{c}\mathcal{D}_1\\ \Omega\ x_2 \Vdash x_1\end{array}}{\Omega \Vdash x_1 \,/\, x_2}\,/R \quad \text{and} \quad \mathcal{E} = \frac{\begin{array}{cc}\mathcal{E}_2 & \mathcal{E}_1\\ \Omega'_R \Vdash x_2 & \Omega_L\ x_1\ \Omega''_R \Vdash z\end{array}}{\Omega_L\ (x_1 \,/\, x_2)\ \Omega'_R\ \Omega''_R \Vdash z}\,/L$$

Using the intuition gained from cut reduction, we can apply the induction hypothesis on x_2, $\mathcal{E}_2$, and $\mathcal{D}_1$ and we obtain

$$\begin{array}{c}\mathcal{D}'_1\\ \Omega\ \Omega'_R \Vdash x_1\end{array} \quad \text{by i.h. on } x_2, \mathcal{E}_2, \mathcal{D}_1$$

We can once again apply the induction hypothesis, this time on x_1, $\mathcal{D}_1'$, and $\mathcal{E}_1$:

$$\begin{array}{c}\mathcal{E}_1' \\ \Omega_L\ \Omega\ \Omega_R'\ \Omega_R'' \Vdash z\end{array} \quad \text{by i.h. on } x_1, \mathcal{D}_1', \mathcal{E}_1$$

Note that $\mathcal{D}_1'$ is the result of the previous appeal to the induction hypothesis and therefore not known to be smaller than $\mathcal{D}_1$, but the appeal to the induction hypothesis is justified since x_1 is a subformula of $x_1 \,/\, x_2$.

Now we can let $\mathcal{F} = \mathcal{E}_1'$ since $\Omega_R = \Omega_R'\ \Omega_R''$ in this case, so we already have the right endsequent.

A more concise way to write down the same argument is in the form of a tree, where rules that are admissible (by induction hypothesis!) are justified in this manner.

Given

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Omega\ x_2 \Vdash x_1\end{array}}{\Omega \Vdash x_1 \,/\, x_2}\,/R \quad \dfrac{\begin{array}{cc}\mathcal{E}_2 & \mathcal{E}_1 \\ \Omega_R' \Vdash x_2 & \Omega_L\ x_1\ \Omega_R'' \Vdash z\end{array}}{\Omega_L\ x_1 \,/\, x_2\ \Omega_R'\ \Omega_R'' \Vdash z}\,/L}{\Omega_L\ \Omega\ \Omega_R'\ \Omega_R'' \Vdash z}\,(\mathsf{cut}?)$$

construct

$$\dfrac{\dfrac{\begin{array}{cc}\mathcal{E}_2 & \mathcal{D}_1 \\ \Omega_R' \Vdash x_2 & \Omega\ x_2 \Vdash x_1\end{array}}{\Omega\ \Omega_R' \Vdash x_1}\,(\text{i.h. on } x_2) \quad \begin{array}{c}\mathcal{E}_1 \\ \Omega_L\ x_1\ \Omega_R'' \Vdash z\end{array}}{\Omega_L\ \Omega\ \Omega_R'\ \Omega_R'' \Vdash z}\,(\text{i.h. on } x_1)$$

This is of course the local reduction, revisited as part of an inductive proof.

Finally we look at a *commutative case*, where the last inference rule applied in the first or second premise of the cut must have been different from the cut formula. We call this a *side formula*. We organize the cases around which rule was applied to which premise. Fortunately, they all go the same way: we "push" up the cut past the inference that was applied to the side formula. We show only one example.

Case: $\mathcal{D} \mathbin{\#} \bullet R$

$$\overset{\mathcal{D}}{\Omega \Vdash x} \quad \text{arbitrary} \quad \text{and} \quad \mathcal{E} = \dfrac{\overset{\mathcal{E}_1}{\Omega_L' \Vdash z_1} \quad \overset{\mathcal{E}_2}{\Omega_L'' \; x \; \Omega_R \Vdash z}}{\Omega_L' \; \Omega_L'' \; x \; \Omega_R \Vdash z_1 \bullet z_2} \bullet R$$

In this case we have the situation

$$\dfrac{\overset{\mathcal{D}}{\Omega \Vdash x} \quad \dfrac{\overset{\mathcal{E}_1}{\Omega_L' \Vdash z_1} \quad \overset{\mathcal{E}_2}{\Omega_L'' \; x \; \Omega_R \Vdash z}}{\Omega_L' \; \Omega_L'' \; x \; \Omega_R \Vdash z_1 \bullet z_2} \bullet R}{\Omega_L' \; \Omega_L'' \; \Omega \; \Omega_R \Vdash z_1 \bullet z_2} \; (\mathsf{cut?})$$

and construct

$$\dfrac{\overset{\mathcal{E}_1}{\Omega_L' \Vdash z_1} \quad \dfrac{\overset{\mathcal{D}}{\Omega \Vdash x} \quad \overset{\mathcal{E}_2}{\Omega_L'' \; x \; \Omega_R \Vdash z}}{\Omega_L'' \; \Omega \; \Omega_R \Vdash z_2} \; \text{i.h. on } x, \mathcal{D}, \mathcal{E}_2}{\Omega_L' \; \Omega_L'' \; \Omega \; \Omega_R \Vdash z_1 \bullet z_2} \bullet R$$

Effectively, we have commuted the cut upward, past the $\bullet R$ inference.

□

Our proof was *constructive*: it presents an effective method for constructing a cut-free proof of the conclusion, given cut-free proofs of the premises. The algorithm that can be extracted from the proof is nondeterministic, since some of the commuting cases overlap when the principal formula is a side formula in both premises. For most logics (although usually classical logic) the result is unique up to further permuting conversions between inference rules, a characterization we will have occasion to discuss later.

4 Cut Elimination from Cut Admissibility

Because of its fundamental importance, there have been many different kinds of proofs of cut elimination for different logics. The first one, which also introduced the sequent calculus, was by Gentzen [Gen35]. We will develop a proof by *structural induction*, by far the most important method of

proof in the study of proofs. This technique was developed in [Pfe94] for classical linear logic, adapted to linear logic by Chang et al. [CCP03]. A key insight is to use the admissibility of cut on cut-free proofs as a lemma.

Theorem 2 (Cut Elimination) *If* $\Omega \vdash x$ *then* $\Omega \Vdash x$.

Proof: We proceed by induction on the structure of

$$\begin{array}{c} \mathcal{D} \\ \Omega \vdash x \end{array}$$

Except for cut, all cases are straightforward. We show one such case.

Case: $\backslash L$

$$\mathcal{D} = \dfrac{\begin{array}{cc} \mathcal{D}_1 & \mathcal{D}_2 \\ \Omega' \vdash y & \Omega_L\ z\ \Omega_R \vdash x \end{array}}{\Omega_L\ \Omega'\ (y \setminus z)\ \Omega_R \vdash x}\ \backslash L$$

Then construct

$$\mathcal{D}' = \dfrac{\begin{array}{cc} \text{i.h.}(\mathcal{D}_1) & \text{i.h.}(\mathcal{D}_2) \\ \Omega' \Vdash y & \Omega_L\ z\ \Omega_R \Vdash x \end{array}}{\Omega_L\ \Omega'\ (y \setminus z)\ \Omega_R \Vdash x}\ \backslash L$$

$\mathcal{D}'$ is cut free since i.h.$(\mathcal{D}_1)$ and i.h.$(\mathcal{D}_2)$ are.

The remaining case is that of cut. Luckily, we can call on admissibility of cut to obtain a cut-free proof of the conclusion!

Case:

$$\mathcal{D} = \dfrac{\begin{array}{cc} \mathcal{D}_1 & \mathcal{D}_2 \\ \Omega' \vdash y & \Omega_L\ y\ \Omega_2 \vdash x \end{array}}{\Omega_L\ \Omega'\ \Omega_R \vdash x}\ \mathsf{cut}_y$$

Then

$$\mathcal{D}' = \dfrac{\begin{array}{cc} \text{i.h.}(\mathcal{D}_1) & \text{i.h.}(\mathcal{D}_2) \\ \Omega' \Vdash y & \Omega_L\ y\ \Omega_2 \Vdash x \end{array}}{\Omega_L\ \Omega'\ \Omega_R \Vdash x}\ (\mathsf{cut}_y)$$

□

5 Identity Elimination

In this section[3] we show identity elimination. Fortunately, it is much easier than cut elimination. It is also not quite as important, since it's relationship to computation is less direct.

Theorem 3 (Admissibility of Identity) *In the sequent calculus where* id_a *is restricted to variables* a*, there are cut-free proofs of* $x \Vdash x$ *for any proposition* x*.*

Proof: By induction on the structure of x. We show only one case; all other cases are similar.

Case: $x = x_1 \,/\, x_2$ Then

$$\dfrac{\dfrac{\overset{\text{i.h.}(x_2)}{\cdots\cdots}\; x_2 \Vdash x_2 \qquad \overset{\text{i.h.}(x_1)}{\cdots\cdots}\; x_1 \Vdash x_1}{x_1 \,/\, x_2 \quad x_2 \Vdash x_1}\,/L}{x_1 \,/\, x_2 \Vdash x_1 \,/\, x_2}\,/R$$

□

The keys to this proof are the identity expansions, just like the cut reductions were the keys to the admissibility of cut.

Theorem 4 (Identity Elimination) *Whenever* $\Omega \vdash x$ *in the sequent calculus, then there exists a proof where identity is applied only to variables. If the given proof is cut-free, so will be the resulting one.*

Proof: By induction on the structure of the given proof, appealing to the admissibility of identity in the case of id_x. □

6 Consequences of Cut Elimination

There are many important consequences of cut elimination. One class of theorems are so-called *refutations*, showing that certain conjectures can not be proven. Here are a few.

Corollary 5 (Consistency) *It is not the case that* $\vdash x$ *for a variable* x*.*

[3] not covered in lecture

Proof: Assume $\vdash x$. By cut elimination, there must be a cut-free proof of x. But no rule could have this conclusion for a variable x. □

Without cut elimination the above proof would not work, because the sequent in question might have been inferred by the cut rule.

Exercises

Exercise 1 The binary counter in Lecture 2, Section 2, is almost in the form of a subsequential finite-state transducer. If we think of eps as the end-marker \$, the only fly in the ointment is the early termination, before all the input is read.

1. Represent increment of a binary string as an FST that correctly reads all of its input. How many states do you need?
2. Use the construction in Section 1 to present this new version as ordered inference rules.
3. Represent the functions $2 * n$ and $2 * n + 1$ for input n both as FSTs and as ordered inference rules.
4. Any examples, conjectures, or theorems how fast the output of an FST may grow as a function of the input on the binary representation of natural numbers?

Exercise 2 Consider defining a new unary connective $\$\, x$ with the following left rule:

$$\frac{x\ \Omega \vdash z}{(\$\, x)\ \Omega \vdash z}\ \$L$$

which is intended provide us with x, but only if $\$\, x$ is at the left end of the context. Define matching right rule(s) and test identity expansion and cut reduction, or explain why it dos not seem to be possible.

Exercise 3 Proceed as in Exercise 2 for new unary connective $x\, \$$ (written in postfix form) defined by

$$\frac{\Omega\ x \vdash z}{\Omega\ (x\, \$) \vdash z}\ \$L$$

which is intended to provide us with x, but only if $x\, \$$ is at the right end of the context.

Exercise 4 Consider defining a new connective $y \Rightarrow x$ with the right rule

$$\frac{\Omega_L\ y\ \Omega_R \vdash x}{\Omega_L\ \Omega_R \vdash y \Rightarrow x}\ \Rightarrow R$$

which is intended to express that y implies x if we can prove x under the assumption y *somewhere* in the antecedent. Define matching left rule(s) and test identity expansion and cut reduction, or explain why it does not seem to be possible.

Exercise 5 Write out the following cases in the proof of cut admissibility.

1. Show the principal case for $\bullet R$ matched against $\bullet L$.
2. Show the principal case for $\& R$ matched against $\& L_2$.
3. Show the principal case for $\mathbf{1}R$ matched against $\mathbf{1}L$.
4. Show all commutative cases for arbitrary $\mathcal{D}$ and $\mathcal{E}$ being $/L$ applied to a side formula.
5. Show all commutative cases for $\mathcal{D}$ being $\backslash L$ and $\mathcal{E}$ being arbitrary.

Exercise 6 Reconsider the alternative rule $\backslash L^*$.

$$\frac{\Omega_L\ x\ \Omega_R \vdash z}{\Omega_L\ y\ (y \setminus x)\ \Omega_R \vdash z}\ \backslash L^*$$

from Lecture 2.

1. Show which cases in the proof of cut admissibility go awry.
2. Prove that cut elimination does *not* hold if $\backslash L$ is replaced by $\backslash L^*$.

Exercise 7 Among the following prove those are true and refute those that are not by taking advantage of cut elimination. We write $x \dashv\vdash y$ for $x \vdash y$ and $y \vdash x$.

1. $x \dashv\vdash x \,\&\, x$
2. $x \dashv\vdash x \bullet x$
3. $\mathbf{1} \dashv\vdash x \setminus x$

References

[CCP03] 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.

[Gen35] 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.

[Lam58] Joachim Lambek. The mathematics of sentence structure. *The American Mathematical Monthly*, 65(3):154–170, 1958.

[Pfe94] Frank Pfenning. Structural cut elimination in linear logic. Technical Report CMU-CS-94-222, Department of Computer Science, Carnegie Mellon University, December 1994.

[Sch77] Marcel Paul Schützenberger. Sur une variante des fonctions sequentielles. *Theoretical Computer Science*, 4(1):47–57, 1977.

Lecture Notes on
Ordered Proofs as Concurrent Programs

15-816: Substructural Logics
Frank Pfenning

Lecture 5
September 13, 2016

In this lecture we will first define *subsingleton logic* which is the subset of ordered logic where each judgment has at most one antecedent. Then we provide an *operational interpretation* of subsingleton logic in which *cut reduction* drives computation. This is in contrast to what we have been doing so far, where *logical inference*, that is, *proof construction* models computation.

The correspondence from this lecture is summarized in the following table.

Logic	**Programming**
Propositions	Session types
Ordered proofs	Concurrent programs
Cut reduction	Communication

This is an instance of a very general connection between proofs and programs studied in type theory. We can vary the logic and the computational interpration. The big upside of this form of correspondence is that it helps us design programming languages in concert with the logic for reasoning about its programs.

This analysis was pioneered by Curry [Cur34] who related proofs in axiomatic form with combinatory logic. Late, Howard [How69] made the discovery that the Church's simply typed λ-calculus was in bijective correspondence with intuitionistic natural deduction. The particular instance of this correspondence for subsingleton logic is a recent discovery by DeYoung and yours truly [DP16].

1 Subsingleton Logic

We can examine the rules for each of the connectives to see which of them are still meaningful if we restrict ourselves to at most one antecedent. For example,

$$\frac{\Omega\ y \vdash x}{\Omega \vdash x \mathbin{/} y}\ /R$$

would have a premise with two antecedents if the conclusion has only one. Restricting the conclusion to just one[1] would not work because identity expansion would fail: in subsingleton logic, we wouldn't be able to prove $x \mathbin{/} y \vdash x \mathbin{/} y$.

What remains from the connectives we have introduced so far is only $x \mathbin{\&} y$ and $\mathbf{1}$. But we haven't had a notion of *disjunction* yet, which is written as $x \oplus y$. It is a disjunction, which means that $x \oplus y$ is true if either x or y is true. So we have two right rules:

$$\frac{\Omega \vdash x}{\Omega \vdash x \oplus y}\ \oplus R_1 \qquad \frac{\Omega \vdash y}{\Omega \vdash x \oplus y}\ \oplus R_2$$

Knowing that x or y is true, but not which one, means that the left rule proceeds by cases.

$$\frac{\Omega_L\ x\ \Omega_R \vdash z \quad \Omega_L\ y\ \Omega_R \vdash z}{\Omega_L\ (x \oplus y)\ \Omega_R \vdash z}\ \oplus L$$

It is straightforward to check the identity expansion and cut reduction, as well as extend the proof of cut elimination accordingly. Disjunction, just like the alternative conjunction, make sense in subsingleton logic.

We summarize the rules of subsingleton logic, with two small notational changes: we write ω for zero or one antecedent, and we use letters A, B, C to denote propositions (rather than x, y, z).

[1] in response to a question in lecture

$$\frac{}{A \vdash A}\ \mathsf{id}_A \qquad \frac{\omega \vdash A \quad A \vdash C}{\omega \vdash C}\ \mathsf{cut}_A$$

$$\frac{\omega \vdash A}{\omega \vdash A \oplus B}\ \oplus_1 \quad \frac{\omega \vdash B}{\omega \vdash A \oplus B}\ \oplus_2 \qquad \frac{A \vdash C \quad B \vdash C}{A \oplus B \vdash C}\ \oplus L$$

$$\frac{\omega \vdash A \quad \omega \vdash B}{\omega \vdash A \,\&\, B}\ \&R \qquad \frac{A \vdash C}{A \,\&\, B \vdash C}\ \&L_1 \quad \frac{B \vdash C}{A \,\&\, B \vdash C}\ \&L_2$$

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

We can generalize this slightly if we are prepared to accept an empty right-hand side. It will not be particularly useful for our purposes today, but the rules would be

$$\frac{A \vdash \cdot}{A \vdash \bot}\ \bot R \qquad \frac{}{\bot \vdash \cdot}\ \bot L$$

Important[2] is that we also have to generalize all the other left rules to permit an empty succedent. So in the presence of $\bot$, sequents would have the form $\omega \vdash \gamma$, with both sides either empty or a singleton.

2 Proofs as Programs

We write $\omega \vdash P : A$ with two alternative interpretations:

1. P is a proof of A with antecedent ω.
2. P is a process providing A and using ω.

Two processes are composed by cut, so that if

$$\omega \vdash P : A \qquad A \vdash Q : C$$

then P and Q can run next to each other and exchange messages. Which messages can be exchanged is dictated by the type (= proposition) A.

As an example, consider

$$\omega \vdash P : A \oplus B \qquad A \oplus B \vdash Q : C$$

[2] which I neglected to mention in lecture

The proof P of $A \oplus B$ will contain critical information, namely if A is true or if B is true. Since the proof Q of C must account for both possibilities, we see that P will eventually *send* some information (inl if A is true, or inr if B is true) and Q will *receive* it. In a synchronous communication model, the message exchange can ony take place if both sides are ready, which correspond to a principal case of cut where $A \oplus B$ is the principal formula in both inferences. Using this intuition to fill in proof terms for one of the cases we arrive at:

$$\frac{\omega \vdash P : A}{\omega \vdash (\mathsf{R}.\mathsf{inl}\ ;\ P) : A \oplus B}\ \oplus R_1 \qquad \frac{A \vdash Q_1 : C \quad B \vdash Q_2 : C}{A \oplus B \vdash (\mathsf{caseL}\ (\mathsf{inl} \Rightarrow Q_1 \mid \mathsf{inr} \Rightarrow Q_2)) : C}\ \oplus L$$

This reduces to the new cut at type A

$$\omega \vdash P : A \qquad A \vdash Q_1 : C$$

So we think of $(\mathsf{R}.\mathsf{inl}\ ;\ P)$ as sending the label inl to the right and then continuing as P, while $\mathsf{caseL}\ (\mathsf{inl} \Rightarrow Q_1 \mid \mathsf{inr} \Rightarrow Q_2)$ receives either inl or inr from the left and continues as Q_1 or Q_2, respectively. The types A that type the interface between two processes are called *session types* (see [HHN+14] for a survey). A strong logical foundation for session types in *linear logic* was discovered by Caires and your lecturer [CP10] and later extended by others [Wad12, CPT13, Ton15].

If $\oplus R_2$ was used in the first proof, then the new cut would be at type B. In either case, the communication corresponds exactly to a principal case in cut reduction.

Looking at computation more globally, processes are configured into a linear chain

$$P_1\ P_2\ P_3\ \cdots\ P_n$$

which we write as ordered propositions

$$\mathsf{proc}(P_1) \quad \mathsf{proc}(P_2) \quad \mathsf{proc}(P_3) \quad \mathsf{proc}(P_n)$$

since we would like to specify the rules of computation for this programming language using ordered inference. Such a configuration is well-typed if we have

$$(\omega_0 \vdash P_1 : A_1) \quad (A_1 \vdash P_2 : A_2) \quad (A_2 \vdash P_3 : A_3) \quad \ldots \quad (A_n \vdash P_n : A_n)$$

where for any two adjacent processes, the type A_i provided by P_i has to be the same as the one used by P_{i+1}.

We now go through the rules and connective of ordered logic and develop the operational interpretation of proofs.

Cut as Composition. Cut is straightforward, since it just corresponds to parallel composition.

$$\frac{\omega \vdash P : A \quad A \vdash Q : C}{\omega \vdash (P \mid Q) : C}\ \mathsf{cut}_A$$

with the computation rule

$$\frac{\mathsf{proc}(P \mid Q)}{\mathsf{proc}(P) \quad \mathsf{proc}(Q)}$$

Clearly, this rule preserves the typing invariant for a configuration if $(P \mid Q)$ is typed by the cut rule, since the type A provided by P is exactly the same as the one as used by Q, and the left and right interfaces ω and C, respectively, are preserved.

Identity as Forwarding. Cut creates two processes from one, while identity removes one process from the configuration, acting like a "forwarding" between the processes to its sides.

$$\frac{}{A \vdash \leftrightarrow : A}\ \mathsf{id}_A$$

The computation rule simply removes the process from the configuration.

$$\frac{\mathsf{proc}(\leftrightarrow)}{\cdot}$$

Again, this preserves the typing invariant for configurations since the processes to the left and right of $\mathsf{proc}(\leftrightarrow)$ have the same type A on the right and left sides, respectively.

Now we come to the logical connectives. We already foreshadowed the case for disjunction, but we first generalize it to be more amenable for programming without changing its logical meaning.

Disjunction as Internal Choice. We generalize disjunction to be an n-ary connective by writen $\oplus\{l_i : A_i\}_{i \in I}$ for some finite index set I and *labels* l_i. Now the binary disjunction is defined as $A \oplus B = \oplus\{\mathsf{inl} : A, \mathsf{inr} : B\}$. Disjunction is also called *internal choice* since the proof itself determines which of the alternatives is chosen.

The right rule will send the apropriate label while the left rule will receive it and branch on it.

$$\frac{\omega \vdash P : A_k \quad (k \in I)}{\omega \vdash (\mathsf{R}.l_k\ ;\ P) : \oplus\{l_i : A_i\}_{i\in I}}\ \oplus R_k \qquad \frac{A_i \vdash Q_i : C \quad (\text{for all } i \in I)}{\oplus\{l_i : A_i\}_{i\in I} \vdash \mathsf{caseL}(l_i \Rightarrow Q_i)_{i\in I} : C}\ \oplus L$$

Again, the computation rule just mirrors the cut reduction and therefore is easily seen to preserve configuration typing.

$$\frac{\mathsf{proc}(\mathsf{R}.l_k\ ;\ P) \quad \mathsf{proc}(\mathsf{caseL}(l_i \Rightarrow Q_i)_{i\in I})}{\mathsf{proc}(P) \quad \mathsf{proc}(Q_k)}$$

The interface type between the two adjacent processes transitions from $\oplus\{l_i : A_i\}_{i\in I}$ to A_k for some $k \in I$.

Alternative Conjunction as External Choice. Again, we generalize from $A\&B$ to $\&\{l_i : A_i\}_{i\in I}$ and defined $A\&B = \&\{\mathsf{inl} : A, \mathsf{inr} : B\}$. $A\&B$ is sometimes called *external choice* since its proof must account for both possibilities and the clients selects between them. Otherwise, it is the straightforward dual of $\oplus$, sending to the left and receiving from the right.

$$\frac{\omega \vdash P_i : A_i \quad (\text{for all } i \in I)}{\omega \vdash \mathsf{caseR}(l_i \Rightarrow P_i)_{i\in I} : \&\{l_i : A_i\}_{i\in I}}\ \&R \qquad \frac{A_k \vdash Q : C \quad (k \in I)}{\&\{l_i : A_i\}_{i\in I} \vdash (\mathsf{L}.l_k\ ;\ Q) : C}\ \&L_k$$

Again, the computation rule just mirrors the cut reduction and therefore is easily seen to preserve configuration typing.

$$\frac{\mathsf{proc}(\mathsf{caseR}(l_i \Rightarrow P_i)_{i\in I}) \quad \mathsf{proc}(\mathsf{L}.l_k\ ;\ Q)}{\mathsf{proc}(P_k) \quad \mathsf{proc}(Q)}$$

Unit as Termination. The unit $\mathbf{1}$ just corresponds to termination. Since communication is synchronous, the paired process to the right just waits for the termination to occur.

$$\frac{}{\cdot \vdash \mathsf{closeR} : \mathbf{1}}\ \mathbf{1}R \qquad \frac{\cdot \vdash Q : C}{\mathbf{1} \vdash \mathsf{waitL}\ ;\ Q : C}\ \mathbf{1}L$$

The computation rule lets the waiting process proceed while the closing one disappears.

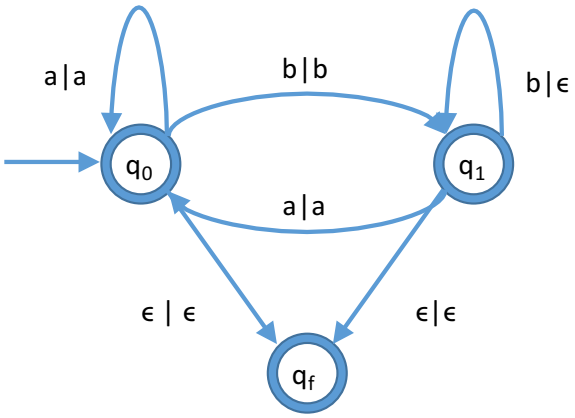
$$\frac{\mathsf{proc}(\mathsf{closeR}) \quad \mathsf{proc}(\mathsf{waitL}\ ;\ Q)}{\mathsf{proc}(Q)}$$

This preserves types since there is no left interface to the configuration before and after the step, and the right interface C is preserved. Again, this can simply be read off from the cut reduction.

This completes the introduction of the computational interpretation of subsingleton logic. The computational metatheory, namely the *progress* and *preservation* theorems will be discussed in the next lecture. We move on to write some programs!

3 Example: Subsequential Finite State Transducers

We have already introduced FSTs in Lecture 4, where we provided an implementation using the computation-as-ordered-inference paradigm. Here, we will use the computation-as-ordered-proof-reduction paradigm instead. We begin with the example of an FST that compresses runs of b into a single b.



We would like to represent the transducer as a process T that receives the input string, symbol by symbol, from the left and sends the output string, again symbol by symbol, to the right.

$$\mathit{string} \vdash T : \mathit{string}$$

The first problem is how to represent the type *string*. It is easy to represent symbols as *labels* and a choice between symbols a, b, and the endmarker \$ as an internal choice

$$\oplus\{a : A_a, b : A_b, \$: A_\$\} \vdash T : \mathit{string}$$

Clearly, T can proceed with a $\oplus L$ rule, which means it can branch on whether it receives an a, b, or a \$. After receiving an a, for example, what should the type on the left be? We can receive further symbols, so it should be again string. This leads us to

$$\mathit{string} = \oplus\{a : \mathit{string}, b : \mathit{string}, \$: A_\$\}$$

which is an example of a *recursive type* since *string* is defined in terms of itself. What should the remaining unspecified type $A_\$$ be? Once we receive the endmarker we can receive no further symbols (or anything else) from the left, we can only wait for the process to our left (that produced the string) to terminate. So $A_\$ = \mathbf{1}$ and we have

$$\mathit{string} = \oplus\{a : \mathit{string}, b : \mathit{string}, \$: \mathbf{1}\}$$

It is convenient in this setting to think of the *string* as being *equal* to the type on the right. If every type definition is *contractive* [GH05] in the sense that it starts with a type constructor ($\oplus, \&, \mathbf{1}$) then we do not need any explicit right or left rules since we can "silently" replace a type by its definition and apply the appropriate rule. This is the idea behind *equirecursive* treatment of recursive types. One should be worried that they could destroy all the good properties of the logic, but with some care this does not have to be the case. We will come back to this point in a future lecture.

Here is a simple program that produces the string $babb$:

$$\mathbf{1} \vdash \ulcorner babb \urcorner : \mathit{string}$$
$$\ulcorner babb \urcorner = \mathsf{R}.b\ ;\ \mathsf{R}.a\ ;\ \mathsf{R}.b\ ;\ \mathsf{R}.b\ ;\ \mathsf{R}.\$\ ;\ \leftrightarrow$$

To deal with recursive types, the program will have to be similarly recursive. At the level of proofs, this can be analyzed as circular proofs [FS13], fixed points [Bae12], or corecursive proofs [TCP14]. Again, we may come back to this point in a future lecture and just freely use recursion for now. Each state of the FST becames a process definition that captures how the FST behaves with the corresponding input. Output is handled simply by sending the appropriate label to the right, and the new state is handled by

invoking this state.

$$\begin{array}{lcl} Q_0 & = & \mathsf{caseL}\ (a \Rightarrow \mathsf{R}.a\ ;\ Q_0 \\ & & \quad \mid b \Rightarrow \mathsf{R}.b\ ;\ Q_1 \\ & & \quad \mid \$ \Rightarrow \mathsf{R}.\$\ ;\ Q_f) \\ \\ Q_1 & = & \mathsf{caseL}\ (a \Rightarrow \mathsf{R}.a\ ;\ Q_0 \\ & & \quad \mid b \Rightarrow Q_1 \\ & & \quad \mid \$ \Rightarrow \mathsf{R}.\$\ ;\ Q_f) \\ \\ Q_f & = & \leftrightarrow \end{array}$$

The type of the final state Q_f is a bit different, since we know input and output have completed by the time this state is reached. We have

$$\begin{array}{lcll} \mathit{string} & \vdash & Q_0 & : \ \mathit{string} \\ \mathit{string} & \vdash & Q_1 & : \ \mathit{string} \\ \mathbf{1} & \vdash & Q_f & : \ \mathbf{1} \end{array}$$

We also note an alternative definition for Q_f

$$Q_f \quad = \quad \mathsf{waitL}\ ;\ \mathsf{closeR}$$

These two definitions are equivalent in the sense that $(\mathsf{waitL}\ ;\ \mathsf{closeR})$ is the identity expansion of $\leftrightarrow\ :\ \mathbf{1}$. We will not go into detail, but this means that those two processes are *observationally equivalent* and can be used interchangeably [PCPT14].

At this point we could almost formulate a conjecture such as

$$\begin{array}{cc} \mathsf{proc}(\ulcorner w \urcorner) & \mathsf{proc}(Q_0) \\ \\ \vdots & \\ \\ \mathsf{proc}(\ulcorner v \urcorner) & \end{array}$$

where Q_0 is the process representing the initial state of the machine that transforms input w to output v. Before reading on, consider why this may not hold.

Yes: the problem is that when T attempts to send an output symbol to the right, there is no consumer so the process will actually block and the computation will come to a halt. There are at least two ways to solve this problem. One is to make communication *asynchronous* so that output (sending a label to the left of right) can always take place. This has two advantages: (1) it is more realistic from the implementation perspective, and (2) it increases the available parallelism. We will return to this option in a future lecture.

Another solution is to create a client that will accept the expected output string. This client is written in the form of a finite automaton, which we discuss in the next section.

4 Finite-State Automata

A (deterministic) finite-state automaton works almost exactly like a subsequential transducer, but it will output only either *acc* or *rej*, not a whole string. This is easy to model:

$$\mathit{answer} = \oplus\{\mathsf{acc} : \mathbf{1}, \mathsf{rej} : \mathbf{1}\}$$

It generalizes the grammar for strings by allowing two different endmarkers (instead of just \$), and has otherwise no symbols.

The we would write

$$\begin{array}{l}
\mathit{string} \vdash \mathit{reject} : \mathit{answer} \\
\mathit{reject} = \mathsf{caseL}\ (a \Rightarrow \mathit{reject} \mid b \Rightarrow \mathit{reject} \mid \$ \Rightarrow \mathsf{R.rej}\ ;\ \leftrightarrow) \\
\\
\mathit{string} \vdash \mathit{accept} : \mathit{answer} \\
\mathit{accept} = \mathsf{caseL}\ (a \Rightarrow \mathit{accept} \mid b \Rightarrow \mathit{accept} \mid \$ \Rightarrow \mathsf{R.acc}\ ;\ \leftrightarrow) \\
\\
\mathit{string} \vdash \llcorner \mathit{bab} \lrcorner : \mathit{answer} \\
\llcorner \mathit{bab} \lrcorner = \mathsf{caseL}\ (a \Rightarrow \mathit{reject} \\
\qquad\qquad\qquad \mid \$ \Rightarrow \mathsf{R.rej}\ ;\ \leftrightarrow \\
\qquad\qquad\qquad \mid b \Rightarrow \mathsf{caseL}\ (b \Rightarrow \mathit{reject} \\
\qquad\qquad\qquad\qquad\qquad \mid \$ \Rightarrow \mathsf{R.rej}\ ;\ \leftrightarrow \\
\qquad\qquad\qquad\qquad\qquad \mid a \Rightarrow \mathsf{caseL}\ (a \Rightarrow \mathit{reject} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \$ \Rightarrow \mathsf{R.rej}\ ;\ \leftrightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid b \Rightarrow \mathit{accept})))
\end{array}$$

We can then test our machine with

$$\begin{array}{c} \mathsf{proc}(\ulcorner bbab \urcorner) \quad \mathsf{proc}(Q_0) \quad \mathsf{proc}(\llcorner bab \lrcorner) \\ \vdots \\ \mathsf{proc}(\mathsf{R}.\mathit{acc} \; ; \leftrightarrow) \end{array}$$

Now we can state more generally that

$$\begin{array}{c} \mathsf{proc}(\ulcorner w \urcorner) \quad \mathsf{proc}(Q_0) \quad \mathsf{proc}(\llcorner v \lrcorner) \\ \vdots \\ \mathsf{proc}(\mathsf{R}.\mathit{acc} \; ; \leftrightarrow) \end{array}$$

if and only of the FST with initial state Q_0 transduces input w to output v. The proof of essentially this theorem is sketched in a recent paper [DP16].

The transitions of the transducer here are not exactly in one-to-one correspondence with the steps of proof construction, since the sequence of reading the input and writing the output are usually seen as a single step. Except for this potential minor difference regarding what counts as a step (depending on the precise formulation of the finite-state transducer), automata transitions are modeled precisely a logical inference steps.

In fact, the opposite is also true. If we have a *cut-free proof*

$$\mathit{string} \Vdash P : \mathit{string}$$

then P will behave like a finite state transducer. The proof is essentially by inversion: Since the proof is cut-free, P can proceed only by forwarding, receiving from the left, sending to the right, or recursing. From this we can easily construct an FST with the same behavior, again allowing for some minor discrepancies in how steps are counted.

Taken together, this means we have an isomormophism between proofs in subsingleton logic containing only $\oplus$ and $\mathbf{1}$ and inductively defined types and subsequential finite state transducers. A recent paper [DP16] slightly generalized FSTs so that they encompass finite-state automata as well by allowing multiple distinct endmarkers, as we have done for the representation of string acceptors.

As a last remark, we notice that composition of transducers is logically trivial, namely just cut. If we have

$$\mathit{string} \vdash T_1 : \mathit{string} \quad \text{and} \quad \mathit{string} \vdash T_2 : \mathit{string}$$

then

$$\mathit{string} \vdash (T_1 \mid T_2) : \mathit{string}$$

Here, the two transducers will run in parallel, similarly to our earlier modeling of transducers via ordered inference. T_1 will pass its output to T_2, which will in turn pass its output to a consumer on the right. We can also just perform cut elimination to obtain a cut-free T' equivalent to $(T_1 \mid T_2)$, but a word of caution: in the presence of corecursive (circular) proofs, the usual cut elimination algorithms has to work somewhat differently [FS13]. Nevertheless, it is an illustration how logical tools such as cut elimination can be used in programming languages, this time in program transformation.

Exercises

Exercise 1 Write a transducer over the alphabet a, b which produces ab for every occurrence of ab in the input and erases all other symbols.

1. Present it in the form of ordered inference rules.
2. Present it in the form of a well-typed program.

Exercise 2 Rewrite your parity-computing inference rules from Exercise L2.2 as a transducer, replacing eps with the endmarker $\$$.

1. Present the transducer in the form of ordered inference rules, for reference. You may freely change your solution of Exercise L2.2 in order to prepare it for part 2.
2. Rewrite it in the form of a well-typed ordered concurrent program.

Exercise 3 Rewrite the program below as a finite state transducer, expressed as a set of ordered inference rules. Describe the function on strings that Q_0 computes.

$$\begin{aligned} Q_0 = \mathsf{caseL}\ (&a \Rightarrow Q_1 \\ \mid\ &b \Rightarrow Q_2 \\ \mid\ &\$ \Rightarrow \mathsf{R}.\$\ ;\ \leftrightarrow) \end{aligned}$$

$$\begin{aligned} Q_1 = \mathsf{caseL}\ (&a \Rightarrow Q_1 \\ \mid\ &b \Rightarrow \mathsf{R}.b\ ;\ Q_2 \\ \mid\ &\$ \Rightarrow \mathsf{R}.\$\ ;\ \leftrightarrow) \end{aligned}$$

$$\begin{aligned} Q_2 = \mathsf{caseL}\ (&a \Rightarrow \mathsf{R}.a\ ;\ Q_1 \\ \mid\ &b \Rightarrow Q_2 \\ \mid\ &\$ \Rightarrow \mathsf{R}.\$\ ;\ \leftrightarrow) \end{aligned}$$

Exercise 4 Reconsider the transducers for compressing runs of b's, given here as a set of ordered inference rules. We present here the version without an explicit final state.

$$\frac{a\ q_0}{q_0\ a} \qquad \frac{b\ q_0}{q_1\ b} \qquad \frac{\$\ q_0}{\$}$$

$$\frac{a\ q_1}{q_0\ a} \qquad \frac{b\ q_1}{q_1} \qquad \frac{\$\ q_1}{\$}$$

In our encoding as a program Q_0 of type *string* $\vdash Q_0$: *string* we treated letters as messages and states as processes. No explicit representation of the final state is necessary with the rules above.

Define a dual encoding where symbols of the alphabet and endmarkers are represented processes and states as messages.

1. Define an appropriate type *state* so that *state* $\vdash P_a$: *state* where P_a is the process representation for the alphabet symbol a.

2. For each symbol a of the transducer alphabet, define the process P_a.

3. Give the type of the process $P_\$$ representing the endmarker \$. You may choose whether to represent a final state as an explicit message of some form or not.

4. Define the process $P_\$$ for the endmarker.

5. Define the initial configuration for the string $babb$ and initial state q_0. Then describe it in general for the machine under consideration here.

6. Define the final configuration for the given example string and initial state. Then describe it in general for the machine under consideration here.

7. Do you foresee any difficulties for encoding subsequential finite state transducers in general in this style? Note that FSTs read one symbol at a time but may output any number of symbols (including none) in one transition. Describe how this could be handled, or explain why a dual construction may only work for a restricted class of FSTs.

8. Consider how to compose transducers and compare to the composition in the original encoding given in lecture.

References

[Bae12] David Baelde. Least and greatest fixed points in linear logic. *ACM Transactions on Computational Logic*, 13(1), 2012.

[CP10] 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.

[CPT13] Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 2013. To appear. Special Issue on Behavioural Types.

[Cur34] H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences, U.S.A.*, 20:584–590, 1934.

[DP16] Henry DeYoung and Frank Pfenning. Substructural proofs as automata. In *14th Asian Symposium on Programming Languages and Systems*, Hanoi, Vietnam, November 2016. Springer LNCS. To appear.

[FS13] Jérôme Fortier and Luigi Santocanale. Cuts for circular proofs: Semantics and cut elimination. In *22nd Conference on Computer Science Logic*, volume 23 of *LIPIcs*, pages 248–262, 2013.

[GH05] Simon J. Gay and Malcolm Hole. Subtyping for session types in the π-calculus. *Acta Informatica*, 42(2–3):191–225, 2005.

[HHN+14] Kohei Honda, Raymond Hu, Rumyana Neykova, Tzu-Chun Chen, Romain Demangeon, Pierre-Malo Deniélou, and Nobuko Yoshida. Structuring communcation with session types. In *Concurrent Objects and Beyond (COB 2014)*, pages 105–127. Springer LNCS 8665, 2014.

[How69] 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.

[PCPT14] Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logical relations and observational equivalences

for session-based concurrency. *Information and Computation*, 239:254–302, 2014.

[TCP14] Bernardo Toninho, Luís Caires, and Frank Pfenning. Corecursion and non-divergence in session-typed processes. In *Proceedings of the 9th International Symposium on Trustworthy Global Computing (TGC 2014)*, Rome, Italy, September 2014. To appear.

[Ton15] Bernardo Toninho. *A Logical Foundation for Session-based Concurrent Computation*. PhD thesis, Carnegie Mellon University and Universidade Nova de Lisboa, May 2015. Available as Technical Report CMU-CS-15-109.

[Wad12] 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 Reasoning about Computation

15-816: Substructural Logics
Frank Pfenning

Lecture 6
September 15, 2016

In this lecture we will begin with a summary of the correspondence between proofs and programs from last lectures and then establish some key properties of the programming language that we derived from subsingleton logic. As usual, we will go back and forth between computational and logical interpretations. The properties we show are the usual preservation and progress, where the first shows that types are preserved during computation, and the second shows that computation can make progress unless it attempts to communicate with the "outside world". Both of these properties emerge very naturally from the cut reduction properties we checked for subsingleton logic.

1 Concurrent Subsingleton Programs

Types	A	$::=$	$\oplus\{l_i : A_i\}_{i\in I}$	internal choice	
		$\mid$	$\&\{l_i : A_i\}_{i\in I}$	external choice	
		$\mid$	$\mathbf{1}$	termination	
Processes	P, Q	$::=$	$\leftrightarrow$	forward	id
		$\mid$	$(P \mid Q)$	compose	cut
		$\mid$	$\mathsf{R}.l_k\ ;\ P$	send label right	$\oplus R_k$
		$\mid$	$\mathsf{caseL}(l_i \Rightarrow Q_i)_{i\in I}$	receive label left	$\oplus L$
		$\mid$	$\mathsf{caseR}(l_i \Rightarrow P_i)_{i\in I}$	receive label right	$\&R$
		$\mid$	$\mathsf{L}.l_k\ ;\ Q$	send label left	$\&L_k$
		$\mid$	closeR	close and notify right	$\mathbf{1}R$
		$\mid$	$\mathsf{waitL}\ ;\ Q$	wait on close left	$\mathbf{1}L$

We also allow mutually recursive type definitions $\alpha = A$ which must be *contractive*, that is, A must be of the form $\oplus\{\ldots\}$, $\&\{\ldots\}$, or $\mathbf{1}$. We treat a type name as equal to its definition and will therefore silently replace it. The usual manner of making this more explicit is to use types of the form $\mu\alpha.A$, but we forego this exercise here.

Similarly, we allow mutually recursive process definitions of variables X as processes P in the form $\omega \vdash X = P : A$. Collectively, these constitute the program $\mathcal{P}$. We fix a global program $\mathcal{P}$ so that the typing judgment, formally, is $\omega \vdash_{\mathcal{P}} P : A$ where we assume that $\omega \vdash_{\mathcal{P}} Q : A$ for every definition $\omega \vdash X = Q : A$ in $\mathcal{P}$. Since $\mathcal{P}$ does not change in any typing

derivation, we omit this subscript in the rules.

$$\frac{}{A \vdash \leftrightarrow : A}\ \mathsf{id}_A \qquad \frac{\omega \vdash P : A \quad A \vdash Q : C}{\omega \vdash (P \mid Q) : C}\ \mathsf{cut}_A$$

$$\frac{\omega \vdash P : A_k \quad (k \in I)}{\omega \vdash (\mathsf{R}.l_k\ ;\ P) : \oplus\{l_i : A_i\}_{i \in I}}\ \oplus R_k \qquad \frac{A_i \vdash Q_i : C \quad (\text{for all } i \in I)}{\oplus\{l_i : A_i\}_{i \in I} \vdash \mathsf{caseL}(l_i \Rightarrow Q_i)_{i \in I} : C}\ \oplus L$$

$$\frac{\omega \vdash P_i : A_i \quad (\text{for all } i \in I)}{\omega \vdash \mathsf{caseR}(l_i \Rightarrow P_i)_{i \in I} : \&\{l_i : A_i\}_{i \in I}}\ \& R \qquad \frac{A_k \vdash Q : C \quad (k \in I)}{\&\{l_i : A_i\}_{i \in I} \vdash (\mathsf{L}.l_k\ ;\ Q) : C}\ \& L_k$$

$$\frac{}{\cdot \vdash \mathsf{closeR} : \mathbf{1}}\ \mathbf{1}R \qquad \frac{\cdot \vdash Q : C}{\mathbf{1} \vdash \mathsf{waitL}\ ;\ Q : C}\ \mathbf{1}L$$

$$\frac{(\omega \vdash X = P : A) \in \mathcal{P}}{\omega \vdash X : A}\ X$$

For the synchronous operational semantics presented via ordered inference, we use ephemeral propositions $\mathsf{proc}(P)$ which expresses the current state of an executing process P. We also import the process definitions $X = P$ as persistent propositions $\underline{\mathsf{def}}(X, P)$.

$$\frac{\mathsf{proc}(\leftrightarrow)}{\cdot}\ \mathsf{fwd} \qquad \frac{\mathsf{proc}(P \mid Q)}{\mathsf{proc}(P) \quad \mathsf{proc}(Q)}\ \mathsf{cmp}$$

$$\frac{\mathsf{proc}(\mathsf{R}.l_k\ ;\ P) \quad \mathsf{proc}(\mathsf{caseL}(l_i \Rightarrow Q_i)_{i \in I})}{\mathsf{proc}(P) \quad \mathsf{proc}(Q_k)}\ \oplus C$$

$$\frac{\mathsf{proc}(\mathsf{caseR}(l_i \Rightarrow P_i)_{i \in I}) \quad \mathsf{proc}(\mathsf{L}.l_k\ ;\ Q)}{\mathsf{proc}(P_k) \quad \mathsf{proc}(Q)}\ \& C$$

$$\frac{\mathsf{proc}(\mathsf{closeR}) \quad \mathsf{proc}(\mathsf{waitL}\ ;\ Q)}{\mathsf{proc}(Q)}\ \mathbf{1}C$$

$$\frac{\mathsf{proc}(X) \quad \underline{\mathsf{def}}(X, P)}{\mathsf{proc}(P)}\ \mathsf{def}$$

A process configuration $\mathcal{C}$ consists of an ordered collection of $\mathsf{proc}(P)$ propositions. The typechecking judgment for configurations, $\omega \vdash \mathcal{C} : \omega'$ is

defined by the following rules that work through $\mathcal{C}$ from right to left.

$$\frac{}{\omega \vdash (\cdot) : \omega} \qquad \frac{\omega \vdash \mathcal{C} : \omega' \quad \omega' \vdash P : A}{\omega \vdash (\mathcal{C}\ \mathsf{proc}(P)) : A}$$

2 Type Preservation

We have already indicated the deeper reason type preservation holds in the last lecture: cut reduction (which is how computation mostly proceeds) preserves the endsequent of the proof and thereby the type of the process. Here, we go through the proof more rigorously.

Even though typing was defined from left to right, if we have a well-typed configuration any subconfiguration is also well-typed.

Lemma 1 (Configuration Typing)

1. *(*Split*) If* $\omega_L \vdash \mathcal{C}_L\ \mathcal{C}_R : \omega_R$ *then* $\omega_L \vdash \mathcal{C}_L : \omega_M$ *and* $\omega_M \vdash \mathcal{C}_R : \omega_R$ *for some* ω_M.
2. *(*Concatenation*) If* $\omega_L \vdash \mathcal{C}_L : \omega_M$ *and* $\omega_M \vdash \mathcal{C}_R : \omega_R$*, then* $\omega_L \vdash \mathcal{C}_L\ \mathcal{C}_R : \omega_R$.
3. *(*Singleton*)* $\omega \vdash \mathsf{proc}(P) : A$ *iff* $\omega \vdash P : A$

Proof: Split follows by induction on the structure of $\mathcal{C}_R$, concatenation by induction on the typing of $\mathcal{C}_R$, and singleton follows by inversion in one direction and by constructing the derivation in the other direction. □

Theorem 2 (Type Preservation) *If* $\omega \vdash \mathcal{C} : A$ *and* $\mathcal{C} \longrightarrow \mathcal{C}'$ *by one step of ordered inference, then* $\omega \vdash \mathcal{C}' : A$.

Proof: By split (Lemma 1), we have $\mathcal{C} = (\mathcal{C}_L\ \mathcal{C}_M\ \mathcal{C}_R)$ where $\omega_L \vdash \mathcal{C}_M : A_M$ is the premise of one of the computation rules. By concatenation (Lemma 1), we have preservation if we can show that the conclusion $\mathcal{C}'_M$ of the computation rule again has type $\omega_L \vdash \mathcal{C}'_M : A_M$.

We show only one case of this proof, since all others proceed analogously.

Case: $\mathcal{C}_M = (\mathsf{proc}(\mathsf{R}.l_k\ ;\ P)\ \mathsf{proc}(\mathsf{caseL}(l_i \Rightarrow Q_i)_{i \in I}))$.

$\omega_L \vdash \mathsf{proc}(\mathsf{R}.l_k\ ;\ P) : B$
and $B \vdash \mathsf{proc}(\mathsf{caseL}(l_i \Rightarrow Q_i)_{i \in I}) : A_M$ by inversion on typing of $\mathcal{C}_M$
$B = \oplus\{l_i : B_i\}_{i \in I}$ and $B_i \vdash Q_i : A_M$ for all $i \in I$
by inversion on typing of caseL
$k \in I$ and $\omega_L \vdash P : B_k$ by inversion on typing of $\mathsf{R}.l_k$
$B_k \vdash Q_k : A_M$ using $i = k$
$\omega_L \vdash (\mathsf{proc}(P)\ \mathsf{proc}(Q_k)) : A_M$ by concatenation

□

3 Progress

Progress means that a configuration will either try to communicate with the "outside world" at its endpoints, or it will be able to make a transition. It does not get stuck in some unexpected way. Just as preservation came down to the fact that cut reduction preserves the endsequent, progress comes down to the fact that every right rule for a connective matched with any left rule will be able to reduce. This was essentially our test whether the interpretation of the logical connectives is meaningful. Again, this intuitive argument is couched in an inductive proof.

Theorem 3 (Progress) *If $\omega \vdash \mathcal{C} : A$ then*

1. *either $\mathcal{C}$ can make a transition (by ordered inference),*
2. *or $\mathcal{C} = (\cdot)$ is empty,*
3. *or $\mathcal{C}$ attempts to communicate to the left (*caseL, $\mathsf{L}.l_k$*, or* waitL*),*
4. *or $\mathcal{C}$ attempts to communicate to the right (*caseR, $\mathsf{R}.l_k$*, or* closeR*).*

Proof: Perhaps surprisingly, the proof is by a simply structural induction on the typing of $\mathcal{C}$. Note that the four cases are not mutually exclusive. For example, the rightmost process in a configuration may want to communicate to the right while another part of the configuration transitions. This is the nature of concurrent computation.

Case:

$$\frac{}{A \vdash (\cdot) : A}$$

The $\mathcal{C} = (\cdot)$ and part 2 applies.

Case:

$$\frac{\omega \vdash \mathcal{C}' : \omega' \quad \omega' \vdash \mathsf{proc}(P) : A}{\omega \vdash (\mathcal{C}'\ \mathsf{proc}(P)) : A}$$

where $\mathcal{C} = (\mathcal{C}'\ \mathsf{proc}(P))$. First, if P is a forward ($\leftrightarrow$), composition ($P_1 \mid P_2$), or a defined name (X), it can make a transition and therefore also $\mathcal{C}$. Moreover, if P communicates to the right, then so does $\mathcal{C}$ and part 4 applies. So we can exclude these cases from consideration below and we may assume that P attempts to communicate to the left (caseL, L.l_k, or waitL).

From the induction hypothesis on the first premise, we know we can distinguish the following subcases.

Subcase: $\mathcal{C}'$ can make a transition. Then so can $\mathcal{C}$.

Subcase: $\mathcal{C}' = (\cdot)$. Then $\mathcal{C} = \mathsf{proc}(P)$. Since P communicates to the left, so does $\mathcal{C}$ and part 3 applies.

Subcase: $\mathcal{C}'$ attempts to communicate to the left. Then so does $\mathcal{C} = (\mathcal{C}'\ \mathsf{proc}(P))$.

Subcase: $\mathcal{C}'$ attempts to communicate to the right, that is, its rightmost process P' has the form caseR, R.l_k or closeR. We already know that P communicates to the left, which is one of caseL, L.l_k, or waitL. Now we apply inversion on the typing of P' and P taking advantage of the fact that the mediating type $\omega' = B$ on the right of P' and left of P must be the same. It emerges from this analysis that one of the remaining two-premise rules ($\oplus C$, $\& C$, or $\mathbf{1}C$) must be applicable. This is because once we fix one side and therefore the interface type B, the rule on the other side must be one of the cases we considered when checking cut reduction.

□

4 Example: Bit Strings Revisited

In this section we revisit the implementation of bit strings and increment from Lecture 2. Recall the ordered logic program for incrementing a bit string, now using \$ as the terminator. Recall that numbers are written with

propositions b0 for 0 and b1 for 1 and are shown with the least significant bit to the right.

$$\frac{\mathsf{b0\ inc}}{\mathsf{b1}}\ \mathsf{inc0} \qquad \frac{\mathsf{b1\ inc}}{\mathsf{inc\ b0}}\ \mathsf{inc1} \qquad \frac{\$\ \mathsf{inc}}{\$\ \mathsf{b1}}\ \mathsf{inc}\$$$

This is not quite a finite state transducer, since we may not read the whole input, but it is still straightforward to represent this as a proof in subsingleton logic. We recommend you try before reading on.

$$
\begin{array}{l}
\mathit{bits} = \oplus\{\mathsf{b0} : \mathit{bits}, \mathsf{b1} : \mathit{bits}, \$: \mathbf{1}\} \\
\mathit{bits} \vdash \mathsf{inc} : \mathit{bits} \\
\mathsf{inc} = \mathsf{caseL}\ (\mathsf{b0} \Rightarrow \mathsf{R.b1}\ ;\ \leftrightarrow \\
\qquad\qquad\quad \mid \mathsf{b1} \Rightarrow \mathsf{R.b0}\ ;\ \mathsf{inc} \\
\qquad\qquad\quad \mid \$ \Rightarrow \mathsf{R.b1}\ ;\ \mathsf{R.\$}\ ;\ \leftrightarrow)
\end{array}
$$

We now experiment with ways we can use logical tools to reason about this program. First, how can we define a type of *std* which corresponds to a bit string in *standard form*, that is, without leading 0 bits? Again, it is an excellent way to test your understanding to construct such a definition before moving on.

The key idea is that we have two mutually recursive types, *std* for any standard bit string and *pos* for a positive one, that is, not consisting only of $\mathsf{b0}$ and \$.

$$\begin{aligned} \mathit{std} = \oplus\{ & \mathsf{b0} : \mathit{pos}, \\ & \mathsf{b1} : \mathit{std}, \\ & \$: \mathbf{1}\} \end{aligned}$$

$$\begin{aligned} \mathit{pos} = \oplus\{ & \mathsf{b0} : \mathit{pos}, \\ & \mathsf{b1} : \mathit{std}\} \end{aligned}$$

Now the increment process should have type

$$\mathit{std} \vdash \mathsf{inc} : \mathit{pos}$$

which expresses that if we receive any number in standard form from the left we will send a positive number in standard form to the right.

Before we proceed with checking the definition, one observation: if $\omega \vdash P : \mathit{std}$ then also $\omega \vdash P : \mathit{bits}$ no matter what ω and P are. This is easy to see, since *std* represents a sequence of 0's and 1's followed by \$ while *std* is just a restricted form of such a sequence. We say *std* is a *subtype* of *bits*, written $\mathit{std} \leq \mathit{bits}$. It turns out that subtyping is decidable [GH05] with an interesting algorithm we intend to return to in a later lecture. Here we just note that

$$\mathit{pos} \leq \mathit{std} \quad \text{and} \quad \mathit{std} \leq \mathit{bits}$$

We further note that in the presence of subtyping we can relax the identity rule to

$$\frac{A \leq A'}{A \vdash \leftrightarrow : A'}\ \mathsf{id}^{\leq}$$

Again, I strongly recommend writing out the typing derivation of the program yourself to check your understanding.

$$std = \oplus\{\mathsf{b0} : pos, \mathsf{b1} : std, \$: \mathbf{1}\}$$
$$pos = \oplus\{\mathsf{b0} : pos, \mathsf{b1} : std\}$$

$$std \vdash \mathsf{inc} : pos$$
$$\begin{array}{l}\mathsf{inc} = \mathsf{caseL}\ (\mathsf{b0} \Rightarrow \mathsf{R.b1}\ ;\ \leftrightarrow \\ \qquad\qquad\quad \mid \mathsf{b1} \Rightarrow \mathsf{R.b0}\ ;\ \mathsf{inc} \\ \qquad\qquad\quad \mid \$ \Rightarrow \mathsf{R.b1}\ ;\ \mathsf{R.\$}\ ;\ \leftrightarrow)\end{array}$$

$$\dfrac{\dfrac{\dfrac{pos \leq std}{pos \vdash \leftrightarrow : std}\ \mathsf{id}^{\leq}}{pos \vdash \mathsf{R.b1}\ ;\ \leftrightarrow : pos}\ \oplus R \quad \dfrac{std \vdash \mathsf{inc} : pos}{std \vdash \mathsf{R.b0}\ ;\ \mathsf{inc} : pos}\ \oplus R \quad \dfrac{\dfrac{\dfrac{}{\mathbf{1} \vdash \leftrightarrow : \mathbf{1}}\ \mathsf{id}}{\mathbf{1} \vdash \mathsf{R.\$}\ ;\ \leftrightarrow : std}\ \oplus R}{\mathbf{1} \vdash \mathsf{R.b1}\ ;\ \mathsf{R.\$}\ ;\ \leftrightarrow : pos}\ \oplus R}{std \vdash \mathsf{caseL}(\mathsf{b0} \Rightarrow \mathsf{R.b1}\ ;\ \leftrightarrow \mid \mathsf{b1} \Rightarrow \mathsf{R.b0}\ ;\ \mathsf{inc} \mid \$ \Rightarrow \mathsf{R.b1}\ ;\ \mathsf{R.\$}\ ;\ \leftrightarrow) : pos}\ \oplus L$$

When we arrive at process names X such as inc, we accept a globally asserted type. This is fine, as long as we make sure that we check all definitions and mirrors the same approach for recursively defined functions in functional languages.

5 Implementing Turing Machines

We have already established that Turing machines can be easily implemented using ordered inference, and that finite state transducers can be implemented using ordered inference as well as ordered proofs. Can we also implement Turing machines, following the same ideas? The answer is yes; our development roughly follows [DP16]. Here is a summary of the representation of Turing machines using ordered inference. The initial configuration is represented by the ordered context

$$\$\ q_0\ \triangleright\ a_1\ \ldots\ a_n\ \$$$

and the final configuration as

$$\$\ b_1\ \ldots\ b_k\ \$$$

and we go from the first to the last by a process of ordered inference using the following rules:

For $\delta(q,a) = (q', a', R)$:
$$\frac{q\ \triangleright\ a}{a'\ q'\ \triangleright}\ \mathsf{LRMR} \qquad \frac{a\ \triangleleft\ q}{a'\ q'\ \triangleright}\ \mathsf{LLMR}$$

For $\delta(q,a) = (q', a', L)$:
$$\frac{q\ \triangleright\ a}{\triangleleft\ q'\ a'}\ \mathsf{LRML} \qquad \frac{a\ \triangleleft\ q}{\triangleleft\ q'\ a'}\ \mathsf{LLML}$$

To extend the tape:
$$\frac{\triangleright\ \$}{\triangleright\ \sqcup\ \$}\ \mathsf{ER} \qquad \frac{\$\ \triangleleft}{\$\ \sqcup\ \triangleleft}\ \mathsf{EL}$$

To halt in final state:
$$\frac{\triangleleft\ q_f}{\cdot}\ \mathsf{FL} \qquad \frac{q_f\ \triangleright}{\cdot}\ \mathsf{FR}$$

We will represent the tape head together with the direction symbol as a process, so that for every state q_k in the machine we have two definitions, ${}^{<}Q_k$ and $Q_k{}^{>}$. What should their types be? A process ${}^{<}Q_k$ will have to receive a tape symbol or endmarker from the left, so its type should be

$$\begin{array}{l} \mathit{tape} = \oplus_{i\in\Sigma}\{a_i : \mathit{tape}, \$: \mathbf{1}\} \\ \mathit{tape} \vdash {}^{<}Q_i : ? \end{array}$$

On the other hand, a right-looking process $Q_k{}^{>}$ will have to receive a tape symbol or endmarker from the right, so its type should be

$$\begin{array}{l} \mathit{epat} = \&_{i\in\Sigma}\{a_i : \mathit{epat}, \$: \mathbf{1}\} \\ ? \vdash Q_i{}^{>} : \mathit{epat} \end{array}$$

It seems advisable for both kinds of processes to have the same type so we can transition easily between them, which gives us

$$\begin{array}{l} \mathit{tape} \vdash {}^{<}Q_i : \mathit{epat} \\ \mathit{tape} \vdash Q_i{}^{>} : \mathit{epat} \end{array}$$

Now consider one case, looking left and moving left:

$$\frac{a \ \lhd \ q}{\lhd \ q' \ a'} \ \mathsf{LLML}$$

The code to effect this change should recognize an a from the left, then output an a' to the right, and then transition to ${}^{<}Q'$.

$$\begin{array}{l} \mathit{tape} \vdash {}^{<}Q : \mathit{epat} \\ {}^{<}Q = \mathsf{caseL}(a \Rightarrow \mathsf{R}.a' \ ; \ {}^{<}Q' \mid \ldots) \end{array}$$

However, there is a serious problem with this code. Can you spot it?

The problem is that it does not type-check! The type that governs Q's communication on the right is $\mathit{epat} = \&\{\ldots\}$ which prescribes *receiving* a symbol from the right rather than sending one! At first, this looks like it is very difficult to repair, but by employing cut we can overcome the problem.

The idea is to spawn a new process on the right whose sole job is to send the symbol a' to the left! Recalling the definition of *epat*, we have

$$\begin{array}{l}\mathit{epat} = \&_{i\in\Sigma}\{a_i : \mathit{epat}, \$: \mathbf{1}\} \\ \mathit{epat} \vdash \mathsf{L}.a' \; ; \leftrightarrow : \mathit{epat}\end{array}$$

We then rewrite the code snippet above as

$$\begin{array}{ll}\mathit{tape} \vdash {}^{<}Q : \mathit{epat} & \\ {}^{<}Q = \mathsf{caseL}(a \Rightarrow \mathsf{R}.a' \; ; {}^{<}Q' \mid \ldots) & \text{ill-typed!} \\ {}^{<}Q = \mathsf{caseL}(a \Rightarrow ({}^{<}Q' \mid (\mathsf{L}.a' \; ; \leftrightarrow)) \mid \ldots) & \text{well-typed!}\end{array}$$

The new cut is well-typed:

$$\dfrac{\mathit{tape} \vdash {}^{<}Q' : \mathit{epat} \quad \dfrac{\dfrac{}{\mathit{epat} \vdash \leftrightarrow : \mathit{epat}}\,\mathsf{id}}{\mathit{epat} \vdash (\mathsf{L}.a' \; ; \leftrightarrow) : \mathit{epat}}\,\&L_{a'}}{\mathit{tape} \vdash ({}^{<}Q' \mid (\mathsf{L}.a' \; ; \leftrightarrow)) : \mathit{epat}}\,\mathsf{cut}$$

With this idea we can easily fill in the four symmetric cases:

$$\dfrac{q \;\triangleright\; a}{a' \; q' \;\triangleright}\,\mathsf{LRMR} \qquad \dfrac{a \;\triangleleft\; q}{a' \; q' \;\triangleright}\,\mathsf{LLMR}$$

$$\dfrac{q \;\triangleright\; a}{\triangleleft \; q' \; a'}\,\mathsf{LRML} \qquad \dfrac{a \;\triangleleft\; q}{\triangleleft \; q' \; a'}\,\mathsf{LLML}$$

with the following snippets

$$\begin{array}{ll}\mathsf{LRMR} & Q^{>} = \mathsf{caseR}(\ldots \mid a \Rightarrow ((\mathsf{R}.a' \; ; \leftrightarrow) \mid Q'^{>}) \mid \ldots) \\ \mathsf{LLMR} & {}^{<}Q = \mathsf{caseL}(\ldots \mid a \Rightarrow ((\mathsf{R}.a' \; ; \leftrightarrow) \mid Q'^{>}) \mid \ldots) \\ \mathsf{LRML} & Q^{>} = \mathsf{caseR}(\ldots \mid a \Rightarrow ({}^{<}Q' \mid (\mathsf{L}.a' \; ; \leftrightarrow)) \mid \ldots) \\ \mathsf{LLML} & {}^{<}Q = \mathsf{caseL}(\ldots \mid a \Rightarrow ({}^{<}Q' \mid (\mathsf{L}.a' \; ; \leftrightarrow)) \mid \ldots)\end{array}$$

The same idea can be used to implement extension of the tape on the left and right. We have slightly rewritten the rules to account for the state q,

because the directional markers $\triangleleft$ and $\triangleright$ are not represented as processes.

$$\frac{q \triangleright \$}{q \triangleright \sqcup \$}\ \mathsf{ER} \qquad \frac{\$ \triangleleft q}{\$ \sqcup \triangleleft q}\ \mathsf{EL}$$

The rule snippets for this are part of Exercise 4

Termination of the machine can now no longer be forwarding (the way it was for the transducer), since the types on the left and right, *tape* and *epat*, respectively, are different. Instead, we could finish with an idling process, or we could traverse the tape to the left end or right end. Going to the right end makes sense if we want to pass on the result of the computation as a string. Going to the left end makes sense if we want to compose Turing machines and start the next machine once the current one has finished (see Exercises 4 and 5)

Exercises

Exercise 1 Show the following cases in the proof of preservation (Theorem 2)

1. Termination (rule $\mathbf{1}C$)
2. Composition (rule cmp)
3. Forwarding (rule fwd)

Exercise 2 Rewrite the code for inc in Section 4 so that forwarding ($\leftrightarrow$) is only used at type $\mathbf{1}$ by adding a second process copy that represents the identity function and calling it in the right place. Then

1. Show the typing derivation for $\mathit{std} \vdash \mathsf{inc} : \mathit{pos}$. Which type(s) do you need for copy?
2. Show the typing derivation(s) for copy.

In the above, forwarding is used only at type $\mathbf{1}$, so we should not need subtyping.

Exercise 3 Define type *even* and *odd* for bit strings (not necessarily in standard form) that represent even and odd binary numbers, respectively. Where they exist, provide the typing derivations for or explain why typing might fail. You may use subtyping if it turns out to be convenient.

$$\mathit{even} \vdash \mathsf{inc} : \mathit{odd}$$
$$\mathit{odd} \vdash \mathsf{inc} : \mathit{even}$$

Exercise 4 Complete the encoding of Turing machines in Section 5.

1. Give the code snippets for the ER and EL rules that extend the tape on the right and left end, respectively.
2. Provide the correct types and code for a final state Q_f that traverses the tape to reach the right endmarker and then terminates so that the final configuration Z behaves like the string

   $$\$\ a_1\ \cdots\ a_n$$

   and has type $\mathbf{1} \vdash Z : \mathsf{tape}$. For this I believe it may be necessary to change the type *epat* in a way that does not affect the remainder of the program but allows us to reach Z.

Exercise 5 In order to compose Turing machines so that the second one runs on the result of the first one, we need the final state of the first machine to traverse the tape to its left end and then transition to the initial state of the next machine.

Develop this as an alternative to Exercise 4.2.

References

[DP16] Henry DeYoung and Frank Pfenning. Substructural proofs as automata. In *14th Asian Symposium on Programming Languages and Systems*, Hanoi, Vietnam, November 2016. Springer LNCS. To appear.

[GH05] Simon J. Gay and Malcolm Hole. Subtyping for session types in the π-calculus. *Acta Informatica*, 42(2–3):191–225, 2005.

Lecture Notes on
From Subsingleton to Ordered Logic

15-816: Substructural Logics
Frank Pfenning

Lecture 7
September 20, 2016

In this lecture we first discuss[1] an *asynchronous* model of communication, where messages are always sent and buffered, even if the recipient is not yet ready to receive. Asynchronous communication increases parallelism in a concurrent program, because it let's the sender continue earlier. It is also more realistic with respect to actual implementation models. We will see the *logical origins* of asynchronous communication [DCPT12], and later that synchronous session-typed communication is actually a special case of asynchronous communication [PG15], which is decidedly not the case in the untyped setting of the π-calculus [Pal03].

Then we generalize our operational interpretation of proofs as processes from the subsingleton fragments to all of ordered logic (to the extent we have introduced it so far).

1 Asynchronous Communication

So far, communication has been *synchronous*: the matched processes that are sending and receiving a message continue with their remaining programs together. *Asynchronous communication* in our case will mean that the sending process will not have to wait for the receiving process, but can continue with the remainder of its computation right away. In contrast, attempting to receive a message will *block* until a message arrives. Asynchronous communication requires a *message buffer*, which turns out to be naturally

[1] Actually, we first discussed some homework solutions, which I will omit here to the delight of future generations of students.

representable in our operational framework. Perhaps more suprisingly, it also has a clear logical interpretation [DCPT12].

As an example, let's take a look at internal choice. Its operational semantics is specified by the following ordered rule of inference.

$$\frac{\mathsf{proc}(\mathsf{R}.l_k\ ;\ P) \quad \mathsf{proc}(\mathsf{caseL}(l_i \Rightarrow Q_i)_{i\in I})}{\mathsf{proc}(P) \quad \mathsf{proc}(Q_k)}\ \oplus C$$

We now decompose this into two rules, one ($\oplus C_s$) to send a message and one ($\oplus C_r$) to receive a message. In order to express this, we use a new proposition $\mathsf{msg}(m)$.

$$\frac{\mathsf{proc}(\mathsf{R}.l_k\ ;\ P)}{\mathsf{proc}(P) \quad \mathsf{msg}(\mathsf{R}.l_k)}\ \oplus C_s \qquad \frac{\mathsf{msg}(\mathsf{R}.l_k) \quad \mathsf{proc}(\mathsf{caseL}(l_i \Rightarrow Q_i)_{i\in I})}{\mathsf{proc}(Q_k)}\ \oplus C_r$$

Note that if a process sends multiple messages, the fact that we have an ordered context will neatly preserve their relative order. In this manner, the messages taken together form a buffer between the two processes. For example:

$$\frac{\dfrac{\mathsf{proc}(\mathsf{R}.l_k\ ;\ \mathsf{R}.l_j\ ;\ P) \quad \mathsf{proc}(Q)}{\mathsf{proc}(\mathsf{R}.l_j\ ;\ P) \quad \mathsf{msg}(\mathsf{R}.l_k) \quad \mathsf{proc}(Q)}\ \oplus C_s}{\mathsf{proc}(P) \quad \mathsf{msg}(\mathsf{R}.l_j) \quad \mathsf{msg}(\mathsf{R}.l_k) \quad \mathsf{proc}(Q)}\ \oplus C_s$$

2 Typing Messages

What should the types of messages be? We have collected enough invariants on the computation state in the last lecture in order to derive what needs to happen here. Let's look at the sending rule first, where we have written the interface types on the right below.

$$\frac{\mathsf{proc}(\mathsf{R}.l_k\ ;\ P)}{\mathsf{proc}(P) \quad \mathsf{msg}(\mathsf{R}.l_k)}\ \oplus C_s \qquad \frac{\omega \vdash (\mathsf{R}.l_k\ ;\ P) : \oplus\{l_i : A_i\}_{i\in I}}{\omega \vdash P : A_k \qquad ??}$$

It is clear from the interface type that we must have

$$A_k \vdash \mathsf{msg}(\mathsf{R}.l_k) : \oplus\{l_i : A_i\}_{i\in I}$$

because it matches the type of P to its left and the type of the interface to the right.

Now comes the central insight of asynchronous communication:

$$\mathsf{msg}(\mathsf{R}.l_k) \simeq \mathsf{proc}(\mathsf{R}.l_k\ ;\leftrightarrow)$$

From the typing perspective, this is easily verified:

$$\dfrac{\dfrac{}{A_k \vdash \leftrightarrow : A_k}\ \mathsf{id}}{A_k \vdash (\mathsf{R}.l_k\ ;\leftrightarrow) : \oplus\{l_i : A_i\}_{i\in I}}\ \oplus R_k$$

We can also see that message receipt

$$\dfrac{\mathsf{msg}(\mathsf{R}.l_k) \quad \mathsf{proc}(\mathsf{caseL}(l_i \Rightarrow Q_i)_{i\in I})}{\mathsf{proc}(Q_k)}\ \oplus C_r$$

works like synchronous communication if we replace the message by a process:

$$\dfrac{\mathsf{proc}(\mathsf{R}.l_k\ ;\leftrightarrow) \quad \mathsf{proc}(\mathsf{caseL}(l_i \Rightarrow Q_i)_{i\in I})}{\dfrac{\mathsf{proc}(\leftrightarrow) \quad \mathsf{proc}(Q_k)}{\mathsf{proc}(Q_k)}\ \mathsf{fwd}}\ \oplus C$$

On the other hand, we really need to treat a message differently from a process, because the *process* $\mathsf{proc}(\mathsf{R}.l_k\ ;\leftrightarrow)$ would immediately spawn another message qua process, which would spawn another message, and so on. So in the operational semantics we use $\mathsf{proc}(P)$ and $\mathsf{msg}(m)$, where our underlying intuition for the meaning of a message is

$$\mathsf{msg}(m) \simeq \mathsf{proc}(m\ ;\leftrightarrow)$$

Rewriting the remaining rules via messages is now straightforward, giving us a complete asynchronous operational semantics. The rules for forwarding, composition, and definitions do not change since they do not involve

communication.

$$\frac{\mathsf{proc}(\leftrightarrow)}{\cdot}\ \mathsf{fwd} \qquad \frac{\mathsf{proc}(P \mid Q)}{\mathsf{proc}(P) \quad \mathsf{proc}(Q)}\ \mathsf{cmp}$$

$$\frac{\mathsf{proc}(\mathsf{R}.l_k\ ;\ P)}{\mathsf{proc}(P) \quad \mathsf{msg}(\mathsf{R}.l_k)}\ \oplus C_s \qquad \frac{\mathsf{msg}(\mathsf{R}.l_k) \quad \mathsf{proc}(\mathsf{caseL}(l_i \Rightarrow Q_i)_{i\in I})}{\mathsf{proc}(Q_k)}\ \oplus C_r$$

$$\frac{\mathsf{proc}(\mathsf{L}.l_k\ ;\ Q)}{\mathsf{msg}(\mathsf{L}.l_k) \quad \mathsf{proc}(Q)}\ \& C_s \qquad \frac{\mathsf{proc}(\mathsf{caseR}(l_i \Rightarrow P_i)_{i\in I}) \quad \mathsf{msg}(\mathsf{L}.l_k)}{\mathsf{proc}(P_k)}\ \& C_r$$

$$\frac{\mathsf{proc}(\mathsf{closeR})}{\mathsf{msg}(\mathsf{closeR})}\ \mathbf{1} C_s \qquad \frac{\mathsf{msg}(\mathsf{closeR}) \quad \mathsf{proc}(\mathsf{waitL}\ ;\ Q)}{\mathsf{proc}(Q)}\ \mathbf{1} C_r$$

$$\frac{\mathsf{proc}(X) \quad \underline{\mathsf{def}}(X, P)}{\mathsf{proc}(P)}\ \mathsf{def}$$

We see that closeR is somewhat of a special case since it requires the antecedents to be empty, so we cannot forward and $\mathsf{msg}(\mathsf{closeR}) \simeq \mathsf{proc}(\mathsf{closeR})$.

3 Asynchronous Communication as Commuting Cut Reduction

The intuition

$$\mathsf{msg}(m) \simeq \mathsf{proc}(m\ ;\ \leftrightarrow)$$

is the key providing a proof-theoretic understanding of asynchronous sending of messages. Consider once again

$$\frac{\mathsf{proc}(\mathsf{R}.l_k\ ;\ P)}{\mathsf{proc}(P) \quad \mathsf{msg}(\mathsf{R}.l_k)}\ \oplus C_s \qquad \frac{\mathsf{proc}(\mathsf{R}.l_k\ ;\ P)}{\mathsf{proc}(P) \quad \mathsf{proc}(\mathsf{R}.l_k\ ;\ \leftrightarrow)}$$

where the second form uses our definition. But the second form produces two processes, which is the exact behavior of a cut:

$$\frac{\mathsf{proc}(\mathsf{R}.l_k\ ;\ P)}{\mathsf{proc}(P) \quad \mathsf{proc}(\mathsf{R}.l_k\ ;\ \leftrightarrow)} \qquad \frac{\mathsf{proc}(P \mid (\mathsf{R}.l_k\ ;\ \leftrightarrow))}{\mathsf{proc}(P) \quad \mathsf{proc}(\mathsf{R}.l_k\ ;\ \leftrightarrow)}\ \mathsf{cmp}$$

But what is the relationship between

$$(\mathsf{R}.l_k\ ;\ P) \quad \text{and} \quad (P \mid (\mathsf{R}.l_k\ ;\ \leftrightarrow)) \quad ?$$

Writing out the proofs:

$$\dfrac{\dfrac{\omega \vdash P : A_k}{\omega \vdash \mathsf{R}.l_k \;;\; P : \oplus\{l_i : A_i\}_{i\in I}}\,\oplus R_k}{\omega \vdash P \mid (\mathsf{R}.l_k \;;\; \leftrightarrow) : \oplus\{l_i : A_i\}_{i\in I}}\,\mathsf{cut} \qquad \dfrac{\omega \vdash P : A_k \quad \dfrac{\dfrac{}{A_k \vdash \leftrightarrow : A_k}\,\mathsf{id}}{A_k \vdash \mathsf{R}.l_k \;;\; \leftrightarrow : \oplus\{l_i : A_i\}_{i\in I}}\,\oplus R_k}{\omega \vdash P \mid (\mathsf{R}.l_k \;;\; \leftrightarrow) : \oplus\{l_i : A_i\}_{i\in I}}\,\mathsf{cut}$$

we see that eliminating the cut in the second proof above will lead to the first one: we push the cut up but the $\oplus R_k$ rule with a commuting cut reduction and then eliminate the resulting cut with the identity.

This means from left to right we *introduce* a cut that can be eliminated by a commuting conversion. Introducing a cut will lead to more processes, but at the same time it will also lead to more parallelism because processes can execute independently.

This also tells us that if we stick with synchronous communication we can easily rewrite our programs to behave asynchronously (at least in this case) simply by rewriting $(\mathsf{R}.l_k \;;\; P)$ as $(P \mid (\mathsf{R}.l_k \;;\; \leftrightarrow))$. In other words, in the synchronous calculus we already had the expressive power of asynchronous communication simply by introducing a pair of cut (= spawn) and identity (= forwarding). For the special case of closeR this analysis is not needed, since closeR has no continuation and we cannot logically distinguish between proc(closeR) and msg(closeR).

From a programming point of view, however, it is much more convenient to stick with the standard construction $\mathsf{R}.l_k \;;\; P$ and interpret all such sends as asynchronous. Interestingly, we show later (or you can read in [PG15]) that we can also go the other way: if we assume all communication is asynchronous we can also recover synchronous communication based on logical principles.

4 From Subsingleton to Ordered Logic

A big next step in this course is investigate how our ideas so far generalize from subsingleton to ordered logic. The difference is only that we allow multiple antecedents. This is a big change since we immediately obtain four new connectives: over ($A \;/\; B$), under ($A \setminus B$), fuse ($A \bullet B$), and twist ($A \circ B$).

Before we get to their operational meaning, let's reconsider the basic judgment. The first attempt is to generalize from

$$A \vdash P : B$$

to

$$A_1 \dots A_n \vdash P : B$$

The problem now is: How can P address A_i if it wants to send or receive a message from it? For example, several of these types might be internal choice, and P could receive a label from any of them. In subsingleton logic, there was only (at most) a single process to the left, so this was unambiguous.

We could address this by saying, for example, that P received from the ith process, essentially numbering the antecedents. This quickly becomes unwieldy, both in practice and in theory. Or we might say that P can only communicate with, say, A_n or A_1, the extremal processes in the antecedents. However, this appears too restrictive (see Exercise 1). Instead, we uniquely label each antecedent as well as the succedent[2] with a *channel name*.

$$(x_1{:}A_1) \dots (x_n{:}A_n) \vdash P :: (y{:}B)$$

We read this as

> *Process P provides a service of type B along channel y and uses channels x_i of type A_i.*

Since we are still in ordered logic, the order of the antecedents matter, and we will see later in which way. We abbreviate it as $\Omega \vdash P :: (y{:}B)$, overloading Ω to stand either for just an ordered sequence of antecedent or one where each antecedent is labeled.

We now generalize each of the rules from before.

Cut. Instead of simply writing $P \mid Q$, the two processes P and Q share a private channel.

$$\frac{\Omega \vdash P_x :: (x{:}A) \quad \Omega_L\ (x{:}A)\ \Omega_R \vdash Q_x :: (z{:}C)}{\Omega_L\ \Omega\ \Omega_R \vdash (x \leftarrow P_x\ ;\ Q_x) :: (z{:}C)}\ \mathsf{cut}$$

As a point of notation, we subscript processes variables such as P or Q with *bound variables* if they are allowed to occur in them. In the process expression $x \leftarrow P_x\ ;\ Q_x$, the variable x is bound and occurs on both side, because it is a channel connecting the two processes. We almost maintain the invariant that all channel names in the antecedent and succedent are distinct, possibly renaming bound variable silently to maintain that.

[2] Not strictly necessary, since the conclusion remains a singleton, but convenient to correlate providers with their clients through a private shared channel.

Operationally, the process executing $(x \leftarrow P_x \ ; \ Q_x)$ continues as Q_x while spawning a new process P_x. This interpretation is meaningful since both $(x \leftarrow P_x \ ; \ Q_x)$ and Q_x offer service C along z. This asymmetry in the operational interpretation comes from the asymmetry of ordered logic (and intuitionistic logic in general) with multiple antecedents but at most one succedent.

In order to define the operational semantics, we write write $\mathsf{proc}(x, P)$ if the process P provides along channel x, which is to say it is typed as $\Omega \vdash P :: (x{:}A)$ for some Ω and A. This is useful to track communications. Then for cut we have the generalized rule of composition

$$\frac{\mathsf{proc}(z, x \leftarrow P_x \ ; Q_x)}{\mathsf{proc}(w, P_w) \quad \mathsf{proc}(z, Q_w)} \ \mathsf{cmp}^w$$

We write cmp^w to remind ourselves that the channel w must be globally "fresh": it is not allowed to occur anywhere else in the process configuration.

Identity. The identity rule could just be

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

based on the idea that x and y are known at this point in a proof so they don't need to be mentioned. Experience dictates that easily irecognizing whenever channels are used makes programs much more readable, so we write

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

and read is as x *is implemented by* y or x *forwards to* y.

There are various levels of detail in the operational semantics for describing identity in the presence of channel names. We cannot simply terminate the process, but we need to actively connect x with y. One way to do this is to globally identify them, which we can do in ordered inference by using equality (which we have not introduced yet).

$$\frac{\mathsf{proc}(x, x \leftarrow y)}{x = y} \ \mathsf{fwd}$$

Internal Choice. This should be straightforward: instead of sending a label "to the right", we send it along the channel the process provides.

$$\frac{\Omega \vdash P :: (x{:}A_k) \quad (k \in I)}{\Omega \vdash (x.l_k\ ;\ P) :: (x : \oplus\{l_i{:}A_i\}_{i\in I})}\ \oplus R_k$$

Conversely, for the left rule we just receive along a channel of the right type, rather than receiving from the right.

$$\frac{\Omega_L\ (x{:}A_i)\ \Omega_R \vdash Q_i :: (z{:}C) \quad (\forall i \in I)}{\Omega_L\ (x{:}\oplus\{l_i{:}A_i\}_{i\in I})\ \Omega_R \vdash \mathsf{case}\ x\ (l_i \Rightarrow Q_i)_{i\in I} :: (z{:}C)}\ \oplus L$$

Communication of the label goes through a channel. We only show the synchronous version:

$$\frac{\mathsf{proc}(x, x.l_k\ ;\ P) \quad \mathsf{proc}(z, \mathsf{case}\ x\ (l_i \Rightarrow Q_i)_{i\in I})}{\mathsf{proc}(x, P) \quad \mathsf{proc}(z, Q_k)}\ \oplus C$$

The fly in the ointment here is that these two processes may actually not be next to each other, because a client can not be next to all of its providers now that there is more than one.

One possible solution is to send messages (asynchronously) and allow them to be move past other messages and processes (see Exercise 2). This, however, does not seem a faithful representation of channel behavior, and a single communication could take many steps of exchange. A simpler solution is to retreat to *linear inference* where the order of the propositions no longer matters. We have used this, for example, to describe the spanning tree construction, Hamiltonian cycles, blocks world, etc. Now we reuse it for the operational semantics. Our earlier rules for cut and identity should also be reinterpreted in linear and not ordered inference.

External Choice. This is symmetric to internal choice and therefore boring (see Lecture 8 for the rules).

Unit. The previous pattern generalizes nicely: instead of closeR and waitL we close and wait on a channel.

$$\frac{}{\cdot \vdash \mathsf{close}\ x :: (x{:}\mathbf{1})}\ \mathbf{1}R \qquad \frac{\Omega_L\ \Omega_R \vdash Q :: (z{:}C)}{\Omega_L\ (x{:}\mathbf{1})\ \Omega_R \vdash (\mathsf{wait}\ x\ ;\ Q) :: (z{:}C)}\ \mathbf{1}L$$

$$\frac{\mathsf{proc}(x, \mathsf{close}\ x) \quad \mathsf{proc}(z, \mathsf{wait}\ x\ ;\ Q)}{\mathsf{proc}(z, Q)}\ \mathbf{1}C$$

Over. As the final connective in this lecture we consider $A \mathbin{/} B$. First, we review the purely logical rules.

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

It turns out that the fact the $/L$ rule has two premises complicates the operational reading, so we use the simplified version $/L^*$. With this rule, cut elimination no longer holds (see Exercise L4.6), but cut reduction and identity expansion still do. Since in the setting of proofs as processes we are less concerned about full cut elimination, this is acceptable here. Eventually, we can arrange that all rules except cut have at most one premise, which means that only cut spawns new processes. Recall also that with $/L^*$ and cut, $/L$ is derivable. We return to this issue in Lecture 9.

$$\frac{\Omega\ B \vdash A}{\Omega \vdash A \mathbin{/} B}\ /R \qquad \frac{\Omega_L\ A\ \Omega_R \vdash C}{\Omega_L\ (A \mathbin{/} B)\ B\ \Omega_R \vdash C}\ /L^*$$

Which of these sends and which receives? In general, there is information in the rule which cannot always be applied (and therefore is not *invertible*), which in this case is $/L^*$. Therefore the process assigned to this rule sends, while the $/R$ receives. We can mechanically fill in channel names and notice that the channel x in $/R$ transitions from type $A \mathbin{/} B$ to type A, so the same transition has to take place in $/L^*$.

$$\frac{\Omega\ (y{:}B) \vdash P_y :: (x{:}A)}{\Omega \vdash ? :: (x{:}A \mathbin{/} B)}\ /R \qquad \frac{\Omega_L\ (x{:}A)\ \Omega_R \vdash Q :: (z{:}C)}{\Omega_L\ (x{:}A \mathbin{/} B)\ (w{:}B)\ \Omega_R \vdash ? :: (z{:}C)}\ /L^*$$

Staring at this rule for a while, we can see *which* information must be transmitted. We reveal the answer on the next page, but you should try to find the answer first.

It is the channel w! Essentially, it is first owned (that is, used) by the process executing the left rule and afterwards it is owned by the process executing the right rule.

$$\frac{\Omega\ (y{:}B) \vdash P_y :: (x{:}A)}{\Omega \vdash (y \leftarrow \mathsf{recv}\ x\ ;\ P_y) :: (x{:}A\ /\ B)}\ /R \qquad \frac{\Omega_L\ (x{:}A)\ \Omega_R \vdash Q :: (z{:}C)}{\Omega_L\ (x{:}A\ /\ B)\ (w{:}B)\ \Omega_R \vdash (\mathsf{send}\ x\ w\ ;\ Q) :: (z{:}C)}\ /L^*$$

The following computation rule implements the cut reduction of $/R$ and $/L^*$.

$$\frac{\mathsf{proc}(x, y \leftarrow \mathsf{recv}\ x\ ;\ P_y) \quad \mathsf{proc}(z, \mathsf{send}\ x\ w\ ;\ Q)}{\mathsf{proc}(x, [w/y]P_y) \quad \mathsf{proc}(z, Q)}\ /C$$

We see that we substitute the received channel w for the bound channel name y in P_y. We will usually write $[w/y]P_y$ as P_w. Since w cannot appear in Q but will appear in P_w, this amounts to an *ownership transfer* for the channel w from one process to another.

In the next lecture we will complete the logical connectives of ordered logic and their operational reading and then summarize them.

Exercises

Exercise 1 Explore whether or not we obtain a logic (and whether this logic has a reasonable operational interpretation) if we restrict the ordered judgment $\Omega\ A \vdash P : B$ so that P can only communicate with the process providing A on the left and B on the right.

Exercise 2 We gave up on the ordered operational semantics because a process needs to communicate with another process that is not an immediate neighbor. Specify an *asynchronous* operational semantics that proceeds via ordered inference and let's messages flow through the configuration. Assess positives and negatives of this semantics.

References

[DCPT12] Henry DeYoung, Luís Caires, Frank Pfenning, and Bernardo Toninho. Cut reduction in linear logic as asynchronous session-typed communication. In P. Cégielski and A. Durand, editors, *Proceedings of the 21st Conference on Computer Science Logic*, CSL 2012, pages 228–242, Fontainebleau, France, September 2012. Leibniz International Proceedings in Informatics.

[Pal03] Catuscia Palamidessi. Comparing the expressive power of the synchronous and the asynchronous π-calculus. *Mathematical Structures in Computer Science*, 13(5):685–719, 2003.

[PG15] 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 Law and Order

15-816: Substructural Logics
Frank Pfenning

Lecture 8
September 22, 2016

In this lecture we first complete a set of connectives for ordered logic and their operational interpretation. Then we will write some programs exploiting the gained expressive power.

1 Multiplicative Connectives

We call $A \,/\, B$, $B \setminus A$, $A \bullet B$ and $A \circ B$ the *multiplicative connectives*, following Girard's nomenclature for linear logic [Gir87]. This class of connectives also includes $\mathbf{1}$, which have already defined.

Under. This is entirely symmetric to / from the last lecture, so we just state the rules here.

$$\frac{(y{:}B)\ \Omega \vdash P_y :: (x{:}A)}{\Omega \vdash (y \leftarrow \mathsf{recv}\ x\ ;\ P_y) :: (x{:}B \setminus A)}\ \setminus R \qquad \frac{\Omega_L\ (x{:}A)\ \Omega_R \vdash Q :: (z{:}C)}{\Omega_L\ (w{:}B)\ (x{:}B \setminus A)\ \Omega_R \vdash (\mathsf{send}\ x\ w\ ;\ Q) :: (z{:}C)}\ \setminus L^*$$

The following computation rule implements the cut reduction of $\setminus R$ and $\setminus L^*$.

$$\frac{\mathsf{proc}(x, y \leftarrow \mathsf{recv}\ x\ ;\ P_y) \quad \mathsf{proc}(z, \mathsf{send}\ x\ w\ ;\ Q)}{\mathsf{proc}(x, P_w) \quad \mathsf{proc}(z, Q)}\ \setminus C$$

Note that the operational reading here is *identical* for $A \,/\, B$; the difference is entirely in the restrictions about where $w{:}B$ or $y{:}B$ are to be found. This indicates that order in this formulation of the operational semantics is not essential from the computational point of view, but imposes some restrictions

on the programs one can write. These restrictions should express some intuitively understandable program properties which will hopefully emerge when we start to use these connectives.

Fuse. The natural right rule for fuse has two premises.

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

In order to avoid spawning a new process in this rule, we have to find an equivalent one-premise version somehow analogous to the restricted left rules for $A \mathbin{/} B$ and $B \setminus A$.

Actually, we could decide that either of the two premises should be the identity, just as the specialized form forms of $/L$ and $/R$ arise by forcing one of the premises to be an identity. So either $\Omega_1 = A$ or $\Omega_2 = B$. These considerations yield:

$$\frac{\Omega \vdash B}{A\ \Omega \vdash A \bullet B}\ \bullet R^* \qquad \frac{\Omega \vdash A}{\Omega\ B \vdash A \bullet B}\ \bullet R^\dagger$$

Rather arbitrarily we pick the first, which yields the following pair of right and left rules for $A \bullet B$

$$\frac{\Omega \vdash B}{A\ \Omega \vdash A \bullet B}\ \bullet R^* \qquad \frac{\Omega_L\ A\ B\ \Omega_R \vdash C}{\Omega_L\ (A \bullet B)\ \Omega_R \vdash C}\ \bullet L$$

Again, we can ask which of the rules carries information, and here it is $\bullet R^*$ which sends. Filling in channel names, we see that once again a channel is sent and received.

$$\frac{\Omega \vdash P :: (x{:}B)}{(w{:}A)\ \Omega \vdash (\mathsf{send}\ x\ w\ ;\ P) :: (x{:}A \bullet B)}\ \bullet R^* \qquad \frac{\Omega_L\ (y{:}A)\ (x{:}B)\ \Omega_R \vdash Q_y :: (z{:}C)}{\Omega_L\ (x{:}A \bullet B)\ \Omega_R \vdash (y \leftarrow \mathsf{recv}\ x\ ;\ Q_y) :: (z{:}C)}\ \bullet L$$

Interestingly, we don't need any new program constructs, since $A \bullet B$, just like $A \mathbin{/} B$ and $B \setminus A$, just send and receive channels. This is reflected in this simple computation rule where we now write Q_w for $[w/y]Q_y$:

$$\frac{\mathsf{proc}(x, \mathsf{send}\ x\ w\ ;\ P) \quad \mathsf{proc}(z, y \leftarrow \mathsf{recv}\ x\ ;\ Q_y)}{\mathsf{proc}(P) \quad \mathsf{proc}(Q_w)}\ \bullet C$$

Twist. $A \circ B$ is entirely symmetric to $A \bullet B$, so we just state the rules.

$$\frac{\Omega \vdash P :: (x{:}B)}{\Omega\ (w{:}A) \vdash (\mathsf{send}\ x\ w\ ;\ P) :: (x{:}A \circ B)}\ \circ R^* \qquad \frac{\Omega_L\ (x{:}B)\ (y{:}A)\ \Omega_R \vdash Q_y :: (z{:}C)}{\Omega_L\ (x{:}A \circ B)\ \Omega_R \vdash (y \leftarrow \mathsf{recv}\ x\ ;\ Q_y) :: (z{:}C)}\ \circ L$$

$$\frac{\mathsf{proc}(x, \mathsf{send}\ x\ w\ ;\ P) \quad \mathsf{proc}(z, y \leftarrow \mathsf{recv}\ x\ ;\ Q_y)}{\mathsf{proc}(P) \quad \mathsf{proc}(Q_w)}\ \circ C$$

2 Rule Summary

Here is a summary of the propositions and proofs, which double as types and programs, together with their rules.

Types	A, B, C	$::=$	$\oplus\{l_i : A_i\}_{i \in I} \mid \&\{l_i : A_i\})_{i \in I} \mid \mathbf{1}$	
		$\mid$	$A \,/\, B \mid B \setminus A \mid A \bullet B \mid A \circ B$	
Processes	P, Q	$::=$	$x \leftarrow y$	identity
		$\mid$	$x \leftarrow P_x\ ;\ Q_x$	cut
		$\mid$	$x.l_k\ ;\ P \mid \mathsf{case}\ x\ (l_i \Rightarrow Q_i)_{i \in I}$	$\oplus, \&$
		$\mid$	$\mathsf{close}\ x \mid \mathsf{wait}\ x\ ;\ Q$	$\mathbf{1}$
		$\mid$	$\mathsf{send}\ x\ y\ ;\ P \mid y \leftarrow \mathsf{recv}\ x\ ;\ Q_x$	$/, \setminus, \bullet, \circ$

We can see that despite some complexity in the language of types, the process language is relatively parsimonious. This syntactic overloading of several constructs is acceptable because during type checking of processes we always track the types of all channels exactly. For this to be always possible, we need to annotate cut with the type of freshly introduced channel, as in $x{:}A \leftarrow P_x\ ;\ Q_x$. Because we are in the sequent calculus, the types of channels in the premises are always strict components of the types in the conclusion.

Judgmental Rules

$$\frac{\Omega \vdash P_x :: (x{:}A) \quad \Omega_L\ (x{:}A)\ \Omega_R \vdash Q_x :: (z{:}C)}{\Omega_L\ \Omega\ \Omega_R \vdash (x \leftarrow P_x\ ;\ Q_x) :: (z{:}C)}\ \mathsf{cut} \qquad \frac{}{y{:}A \vdash x \leftarrow y :: (x{:}A)}\ \mathsf{id}$$

Additive Connectives

$$\frac{\Omega \vdash P :: (x{:}A_k) \quad (k \in I)}{\Omega \vdash (x.l_k\ ;\ P) :: (x : \oplus\{l_i{:}A_i\}_{i \in I})}\ \oplus R_k \qquad \frac{\Omega_L\ (x{:}A_i)\ \Omega_R \vdash Q_i :: (z{:}C) \quad (\forall i \in I)}{\Omega_L\ (x{:}\oplus\{l_i{:}A_i\}_{i \in I})\ \Omega_R \vdash \mathsf{case}\ x\ (l_i \Rightarrow Q_i)_{i \in I} :: (z{:}C)}\ \oplus L$$

$$\frac{\Omega \vdash P_i :: (x{:}A_i) \quad (\forall i \in I)}{\Omega \vdash \mathsf{case}\ x\ (l_i \Rightarrow P_i)_{i \in I} :: (x{:}\&\{l_i{:}A_i\}_{i \in I}))}\ \&R \qquad \frac{\Omega_L\ (x{:}A_k)\ \Omega_R \vdash P :: (z{:}C) \quad (k \in I)}{\Omega_L\ (x : \&\{l_i{:}A_i\}_{i \in I})\ \Omega_R \vdash (x.l_k\ ;\ Q) :: (z{:}C)}\ \&L_k$$

Multiplicative Connectives

$$\frac{}{\cdot \vdash \mathsf{close}\ x :: (x{:}\mathbf{1})}\ \mathbf{1}R \qquad \frac{\Omega_L\ \Omega_R \vdash Q :: (z{:}C)}{\Omega_L\ (x{:}\mathbf{1})\ \Omega_R \vdash (\mathsf{wait}\ x\ ;\ Q) :: (z{:}C)}\ \mathbf{1}L$$

$$\frac{\Omega\ (y{:}B) \vdash P_y :: (x{:}A)}{\Omega \vdash (y \leftarrow \mathsf{recv}\ x\ ;\ P_y) :: (x{:}A\ /\ B)}\ /R \qquad \frac{\Omega_L\ (x{:}A)\ \Omega_R \vdash Q :: (z{:}C)}{\Omega_L\ (x{:}A\ /\ B)\ (w{:}B)\ \Omega_R \vdash (\mathsf{send}\ x\ w\ ;\ Q) :: (z{:}C)}\ /L^*$$

$$\frac{(y{:}B)\ \Omega \vdash P_y :: (x{:}A)}{\Omega \vdash (y \leftarrow \mathsf{recv}\ x\ ;\ P_y) :: (x{:}B \setminus A)}\ \setminus R \qquad \frac{\Omega_L\ (x{:}A)\ \Omega_R \vdash Q :: (z{:}C)}{\Omega_L\ (w{:}B)\ (x{:}B \setminus A)\ \Omega_R \vdash (\mathsf{send}\ x\ w\ ;\ Q) :: (z{:}C)}\ \setminus L^*$$

$$\frac{\Omega \vdash P :: (x{:}B)}{(w{:}A)\ \Omega \vdash (\mathsf{send}\ x\ w\ ;\ P) :: (x{:}A \bullet B)}\ \bullet R^* \qquad \frac{\Omega_L\ (y{:}A)\ (x{:}B)\ \Omega_R \vdash Q_y :: (z{:}C)}{\Omega_L\ (x{:}A \bullet B)\ \Omega_R \vdash (y \leftarrow \mathsf{recv}\ x\ ;\ Q_y) :: (z{:}C)}\ \bullet L$$

$$\frac{\Omega \vdash P :: (x{:}B)}{\Omega\ (w{:}A) \vdash (\mathsf{send}\ x\ w\ ;\ P) :: (x{:}A \circ B)}\ \circ R^* \qquad \frac{\Omega_L\ (x{:}B)\ (y{:}A)\ \Omega_R \vdash Q_y :: (z{:}C)}{\Omega_L\ (x{:}A \bullet B)\ \Omega_R \vdash (y \leftarrow \mathsf{recv}\ x\ ;\ Q_y) :: (z{:}C)}\ \circ L$$

For the operational semantics we have to remember that we are using *linear inference*, not ordered inference.

$$\frac{\mathsf{proc}(z, x \leftarrow P_x \;;\; Q_x)}{\mathsf{proc}(w, P_w) \quad \mathsf{proc}(z, Q_w)}\ \mathsf{cmp}^w \qquad \frac{\mathsf{proc}(x, x \leftarrow y)}{x = y}\ \mathsf{fwd}$$

$$\frac{\mathsf{proc}(x, x.l_k \;;\; P) \quad \mathsf{proc}(z, \mathsf{case}\ x\ (l_i \Rightarrow Q_i)_{i \in I})}{\mathsf{proc}(x, P) \quad \mathsf{proc}(z, Q_k)}\ \oplus C$$

$$\frac{\mathsf{proc}(x, \mathsf{case}\ x\ (l_i \Rightarrow P_i)_{i \in I}) \quad \mathsf{proc}(z, x.l_k \;;\; Q)}{\mathsf{proc}(x, Q) \quad \mathsf{proc}(z, P_k)}\ \& C$$

$$\frac{\mathsf{proc}(x, \mathsf{close}\ x) \quad \mathsf{proc}(z, \mathsf{wait}\ x \;;\; Q)}{\mathsf{proc}(z, Q)}\ \mathbf{1}C$$

$$\frac{\mathsf{proc}(x, y \leftarrow \mathsf{recv}\ x \;;\; P_y) \quad \mathsf{proc}(z, \mathsf{send}\ x\ w \;;\; Q)}{\mathsf{proc}(x, P_w) \quad \mathsf{proc}(z, Q)}\ /C, \backslash C$$

$$\frac{\mathsf{proc}(x, \mathsf{send}\ x\ w \;;\; P) \quad \mathsf{proc}(z, y \leftarrow \mathsf{recv}\ x \;;\; Q_y)}{\mathsf{proc}(P) \quad \mathsf{proc}(Q_w)}\ \bullet C, \circ C$$

3 Example: Lists

We are used to lists being a data structure; here it describes the behavior of a process which (essentially) sends a sequence of elements. Looking back at the rules, we see that this requires either fuse or twist, and we choose fuse. It therefore goes beyond the subsingleton fragment.

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

Lists are polymorphic in the sense the type of all elements in a list must be the same, but arbitrary, session type A. We indicate this with a subscript A on the list type, which therefore represents really a whole collection of types. We might decide to formally introduce first class type constructors and explicit polymorphism at a later time.

Our first program will be to append two lists. We will write it in stages.

$(l_1{:}\mathsf{list}_A)\ (l_2{:}\mathsf{list}_A) \vdash \mathit{append} : (l{:}\mathsf{list}_A)$
$\mathit{append} = \mathsf{case}\ l_1\ (\mathsf{cons} \Rightarrow \ldots \quad \% \quad (l_1{:}A \bullet \mathsf{list}_A)\ (l_2{:}\mathsf{list}_A) \vdash l{:}\mathsf{list}_A$
$\qquad\qquad\qquad\qquad\quad \mid \mathsf{nil} \Rightarrow \ldots)$

We have written in the type of the first hole, indicated by the first ellipsis, that we are working to fill. From the type we can see that l_1 will send us a channel x of type A, and since it is $A \bullet \mathsf{list}_A$ we are required to add this to the left of l_1, which yields:

$$
\begin{array}{l}
(l_1{:}\mathsf{list}_A)\ (l_2{:}\mathsf{list}_A) \vdash \mathit{append} : (l{:}\mathsf{list}_A) \\
\mathit{append} = \mathsf{case}\ l_1\ \ (\mathsf{cons} \Rightarrow x \leftarrow \mathsf{recv}\ l_1\ ; \\
\qquad\qquad\qquad\qquad \ldots \qquad \% \quad (x{:}A)\ (l_1{:}\mathsf{list}_A)\ (l_2{:}\mathsf{list}_A) \vdash l{:}\mathsf{list}_A \\
\qquad\qquad\quad \mid \mathsf{nil} \Rightarrow \ldots)
\end{array}
$$

At this point we know the result list will start with x, and fortunately we can find x at the left end of the context. So we can send it along x, after we indicate the result starts with an element by sending the label cons.

$$
\begin{array}{l}
(l_1{:}\mathsf{list}_A)\ (l_2{:}\mathsf{list}_A) \vdash \mathit{append} : (l{:}\mathsf{list}_A) \\
\mathit{append} = \mathsf{case}\ l_1\ \ (\mathsf{cons} \Rightarrow x \leftarrow \mathsf{recv}\ l_1\ ; \\
\qquad\qquad\qquad\qquad l.\mathsf{cons}\ ;\ \mathsf{send}\ l\ x\ ; \\
\qquad\qquad\qquad\qquad \ldots \qquad \% \quad (l_1{:}\mathsf{list}_A)\ (l_2{:}\mathsf{list}_A) \vdash l{:}\mathsf{list}_A \\
\qquad\qquad\quad \mid \mathsf{nil} \Rightarrow \ldots)
\end{array}
$$

Now we are back to the original type and we can make a recursive call in this branch of the case expression.

$$
\begin{array}{l}
(l_1{:}\mathsf{list}_A)\ (l_2{:}\mathsf{list}_A) \vdash \mathit{append} : (l{:}\mathsf{list}_A) \\
\mathit{append} = \mathsf{case}\ l_1\ \ (\mathsf{cons} \Rightarrow x \leftarrow \mathsf{recv}\ l_1\ ; \\
\qquad\qquad\qquad\qquad l.\mathsf{cons}\ ;\ \mathsf{send}\ l\ x\ ; \\
\qquad\qquad\qquad\qquad \mathit{append} \\
\qquad\qquad\quad \mid \mathsf{nil} \Rightarrow \ldots \qquad \% \quad (l_1{:}\mathbf{1})\ (l_2{:}\mathsf{list}_A) \vdash l{:}\mathsf{list}_A \\
\qquad\qquad\quad)
\end{array}
$$

We have filled in the type of l_1 in the second branch of the case expression. The code is now easy: we wait for l_1 to terminate and then implement l as l_2 by forwarding. This is possible since their types match.

$$
\begin{array}{l}
(l_1{:}\mathsf{list}_A)\ (l_2{:}\mathsf{list}_A) \vdash \mathit{append} : (l{:}\mathsf{list}_A) \\
\mathit{append} = \mathsf{case}\ l_1\ \ (\mathsf{cons} \Rightarrow x \leftarrow \mathsf{recv}\ l_1\ ; \\
\qquad\qquad\qquad\qquad l.\mathsf{cons}\ ;\ \mathsf{send}\ l\ x\ ; \\
\qquad\qquad\qquad\qquad \mathit{append} \\
\qquad\qquad\quad \mid \mathsf{nil} \Rightarrow \mathsf{wait}\ l_1\ ;\ l \leftarrow l_2)
\end{array}
$$

It is very easy to imagine some syntactic sugar, where consecutive sends and receives of labels and channels are combined. For example:

$(l_1{:}\mathsf{list}_A)\ (l_2{:}\mathsf{list}_A) \vdash \mathit{append} : (l{:}\mathsf{list}_A)$
$\mathit{append} = \mathsf{case}\ l_1\ \ (\mathsf{cons}(x) \Rightarrow l.\mathsf{cons}(x)\ ;\ \mathit{append}$
$\qquad\qquad\qquad\qquad\quad \mid \mathsf{nil}() \Rightarrow l \leftarrow l_2)$

We refrain from such niceties because it obscures the structure of communication.

However, we will make one change. There is an oddity in our code in that the type declaration for *append* declares channel variables l_1, l_2 and l which appear free in the definition. Really, they should be somehow *bound* so that the definition is self-contained. Our notation for a process name X with type declaration

$$(x_1{:}A_1) \ldots (x_n{:}A_n) \vdash X :: (x{:}A)$$

is

$$x \leftarrow X \leftarrow x_1 \ldots x_n = P(x, x_1, \ldots, x_n)$$

thereby writing the provided channel x like a return value and the used channels channel and $x_1, \ldots, x_n$ like arguments to X. Note that $x, x_1, \ldots x_n$ are *bound* channel names occuring in the body P and can be renamed as convenient. In this form, which we will use from now on, we have

$(l_1{:}\mathsf{list}_A)\ (l_2{:}\mathsf{list}_A) \vdash \mathit{append} : (l{:}\mathsf{list}_A)$
$l \leftarrow \mathit{append} \leftarrow l_1\ l_2 =$
$\quad \mathsf{case}\ l_1\ \ (\mathsf{cons} \Rightarrow x \leftarrow \mathsf{recv}\ l_1\ ;$
$\qquad\qquad\qquad\qquad\quad l.\mathsf{cons}\ ;\ \mathsf{send}\ l\ x\ ;$
$\qquad\qquad\qquad\qquad\quad l \leftarrow \mathit{append} \leftarrow l_1\ l_2$
$\qquad\qquad\quad \mid \mathsf{nil} \Rightarrow \mathsf{wait}\ l_1\ ;\ l \leftarrow l_2)$

Note that the recursive calls also now specifies the offered channels l and the used channels l_1 and l_2.

In practice, is it convenient to fold in an appeal to a defined process name with a cut. More formally, we generalize definitions and the def rules for defined process names to be as follows:

$$\frac{\mathsf{proc}(y \leftarrow X \leftarrow y_1 \ldots y_n\ ;\ Q_y) \quad \underline{\mathsf{def}}(x \leftarrow X \leftarrow x_1 \ldots x_n = P_{x,x_1,\ldots,x_n})}{\mathsf{proc}(P_{w,y_1,\ldots,y_n}) \quad \mathsf{proc}(Q_w)}\ \mathsf{def}^w$$

At first it might look like there could be many processes of the type

$(l_1{:}\mathsf{list}_A)\ (l_2{:}\mathsf{list}_A) \vdash X : (l{:}\mathsf{list}_A)$

for example, taking alternating elements from l_1 and l_2, or appending l_2 to l_1. I believe that none of these would be well-typed, and essentially the only possible behavior is to append l_1 and l_2 or diverge at some point. This is pure conjecture, since at this point we have not developed the necessary theories of observational equivalence and parametricity that might allow us to prove such a result. Similarly, I would conjecture that $(t{:}\mathsf{list}_A) \vdash X :: (l{:}\mathsf{list}_A)$ would force X to be observationally equivalent to the identity and could not, for example, reverse t. Occasionally we will want to reverse lists, which means eventually we will have to leave the confines of ordered concurrency. But let's first see what we can write here.

Next we think about writing *processes nil* and *cons* that behave like the empty list, or add an element to a given list. We will write these in one installment.

$\cdot \vdash \mathit{nil} : (l{:}\mathsf{list}_A)$
$l \leftarrow \mathit{nil} =$
  $l.\mathsf{nil}$; $\mathsf{close}\ l$

$(x{:}A)\ (t : \mathsf{list}_A) \vdash \mathit{cons} : (l{:}\mathsf{list}_A)$
$l \leftarrow \mathit{cons} \leftarrow x\ t =$
  $l.\mathsf{cons}$; $\mathsf{send}\ l\ x$; $l \leftarrow t$

Note that there seems to be no possible implementation if we *reverse* the arguments to cons.

$(t : \mathsf{list}_A)\ (x{:}A) \vdash \mathit{cons}' : (l{:}\mathsf{list}_A)$
$l \leftarrow \mathit{cons}' \leftarrow x\ t =$
  $l.\mathsf{cons}$; $\mathsf{send}\ l\ x$??

The send is ill-typed since x is at the wrong end of the context for the type $l : A \bullet \mathsf{list}_A$. The ordering constraints impose a tight discpline on the use of channels. See also Exercise 2.

In the next lecture we will write some more programs along these lines.

Exercises

Exercise 1 Reconsider the proposition $\bot$ from Lecture 5 and write out logical rules as well as an operational semantics via rules of linear inference. Implicitly this means that all the other rules with a parametric succedent (usuall denoted by C or $z{:}C$) should be generalized to also allow an empty succedent (which you do not need to rewrite). Operationally, this corresponds to a detached process that has no client, but may use other processes.

Exercise 2 Can you write an intuititively meaningful process *cons*′ with

$$(t : \mathsf{list}_A)\ (x{:}A) \vdash \mathit{cons}' : (l{:}\mathsf{list}_A)$$

If so, show the definition and explain what it does. If not, explain why you think it might be impossible to write a process of this type.

Exercise 3 Define

$$\mathsf{tsil}_A = \oplus\{\mathsf{snoc} : A \circ \mathsf{tsil}_A, \mathsf{lin} : \mathbf{1}\}$$

Define processes *dneppa*, *lin*, and *snoc* that mirror *append*, *nil* and *cons*.

Exercise 4 Define

$$\mathsf{dlist}_A = \oplus\{\mathsf{cons} : A \bullet \mathsf{dlist}_A, \mathsf{snoc} : A \circ \mathsf{dlist}_A, \mathsf{nil} : \mathbf{1}\}$$

Explore the behavior of this type, and which kinds of operations can be defined over this type. Form some conjectures about which operations may be impossible.

Exercise 5 We would like to define a type $\%A$ which behaves exactly like A once it has synchronized with a client at the same type. No information except that the client has arrived at a matching point will be exchanged.

1. Give logical $\%R$ and $\%L$ rules.
2. Assign process expressions to these rules.
3. Provide the operational $\%C$ rule using linear inference.
4. Can you define $\%A$ in terms of other connectives in ordered logic with a corresponding operational behavior?

References

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

Lecture Notes on Queues and Stacks

15-816: Substructural Logics
Frank Pfenning

Lecture 9
Tuesday, September 27, 2016

In the last lecture we introduced lists with arbitrary elements and wrote ordered programs for *nil* (the empty list), *cons* (adding an element to the head of a list) and *append* to append two lists. The representation was in the form of an *internal choice*

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

We might think of this as the usual functional data structure of lists, but we should keep in mind that it is really just an interface specification for processes. It does not imply any particular representation.

Today, we will look at a data structure in which we can insert and delete channels of arbitrary type. The interface is different because it is in the form of an *external choice*, more in the style of object-oriented programming or signatures in module systems for functional languages.

1 Storing Channels

Here is our simple interface to a storage service for channels:

$$\begin{array}{l}\mathsf{store}_A = \&\{\ \mathsf{ins} : A \setminus \mathsf{store}_A, \\ \qquad\qquad \mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : A \bullet \mathsf{store}_A\}\}\end{array}$$

Using our operational interpretation, we can read this as follows:

> *A* store *for channels of type A offers a client a choice between insertion (label* ins*) and deletion (label* del*).*
> *When inserting, the clients sends a channel of type A which is added*

> *to the store.*
> *When deleting, the store responds with the label* none *if there are no elements in the store and terminates, or with the label* some*, followed by an element.*
> *When an element is actually inserted or deleted the provider of the storage service then waits for the next input (again, either an insertion or deletion).*

In this reading we have focused on the operations, and intentionally ignored the restrictions order might place on the use of the storage service. Hopefully, this will emerge as we write the code and analyze what the restrictions might mean.

First, we have to be able to create an empty store. We will write the code in stages, because I believe it is much harder to understand the final program than it is to follow its construction.

$$\begin{array}{l}\mathsf{store}_A = \&\{\ \mathsf{ins} : A \setminus \mathsf{store}_A, \\ \qquad\qquad \mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : A \bullet \mathsf{store}_A\}\}\end{array}$$

First, the header of the process definition.

$$\begin{array}{l}\cdot \vdash \mathit{empty} :: (s : \mathsf{store}_A) \\ s \leftarrow \mathit{empty} = \ldots\end{array}$$

Because a store_A is an external choice, we begin with a case construct, branching on the received label.

$$\begin{array}{lll}\cdot \vdash \mathit{empty} :: (s : \mathsf{store}_A) & & \\ s \leftarrow \mathit{empty} = \mathsf{case}\ s\ (\mathsf{ins} \Rightarrow \ldots & \% & \cdot \vdash s : A \setminus \mathsf{store}_A \\ \qquad\qquad\qquad\quad \mid \mathsf{del} \Rightarrow \ldots & \% & \cdot \vdash s : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : A \bullet \mathsf{store}_A\} \\ \qquad\qquad\qquad\quad)\end{array}$$

The case of deletion is actually easier: since this process represents an empty store, we send the label none and terminate.

$$\begin{array}{lll}\cdot \vdash \mathit{empty} :: (s : \mathsf{store}_A) & & \\ s \leftarrow \mathit{empty} = \mathsf{case}\ s\ (\mathsf{ins} \Rightarrow \ldots & \% & \cdot \vdash s : A \setminus \mathsf{store}_A \\ \qquad\qquad\qquad\quad \mid \mathsf{del} \Rightarrow s.\mathsf{none}\ ;\ \mathsf{close}\ s) & &\end{array}$$

In the case of an insertion, the type dictates that we receive a channel of type A which we call x. It is added at the left end of the antecedents. Since they are actually none, both $A \setminus \mathsf{store}_A$ and $\mathsf{store}_A \,/\, A$ would behave the same way here.

$$
\begin{array}{l}
\cdot \vdash \mathit{empty} :: (s : \mathsf{store}_A) \\
s \leftarrow \mathit{empty} = \mathsf{case}\ s\ (\mathsf{ins} \Rightarrow \qquad\qquad\qquad\qquad \% \quad \cdot \vdash s : A \setminus \mathsf{store}_A \\
\qquad\qquad\qquad\qquad x \leftarrow \mathsf{recv}\ s\ ; \qquad \% \quad x{:}A \vdash s : \mathsf{store}_A \\
\qquad\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad \mid \mathsf{del} \Rightarrow s.\mathsf{none}\ ;\ \mathsf{close}\ s)
\end{array}
$$

At this point it seems like we are stuck. We need to start a process implementing a store with *one* element, but so far we just writing the code for an empty store. We need to define a process *elem*

$$(x{:}A)\ (t{:}\mathsf{store}_A)\ \vdash \mathit{elem} :: (s : \mathsf{store}_A)$$

which holds an element $x{:}A$ and also another store $t{:}\mathsf{store}_A$ with further elements. In the singleton case, t will then be the empty store. Therefore, we first make a recursive call to create another empty store, calling it n for *none*.

$$
\begin{array}{l}
\cdot \vdash \mathit{empty} :: (s : \mathsf{store}_A) \\
s \leftarrow \mathit{empty} = \mathsf{case}\ s\ (\mathsf{ins} \Rightarrow x \leftarrow \mathsf{recv}\ s\ ; \qquad \% \quad x{:}A \vdash s : \mathsf{store}_A \\
\qquad\qquad\qquad\qquad\ n \leftarrow \mathit{empty}\ ; \qquad \% \quad (x{:}A)\ (n{:}\mathsf{store}_A) \vdash s : \mathsf{store}_A \\
\qquad\qquad\qquad\qquad\ \ldots \\
\qquad\qquad\qquad \mid \mathsf{del} \Rightarrow s.\mathsf{none}\ ;\ \mathsf{close}\ s) \\
\\
(x{:}A)\ (t{:}\mathsf{store}_A)\ \vdash \mathit{elem} :: (s : \mathsf{store}_A) \\
s \leftarrow \mathit{elem} \leftarrow x\ t = \ldots
\end{array}
$$

Postponing the definition of *elem* for now, we can invoke *elem* to create a singleton store with just x, calling the resulting channel e. This call will consume x and n, leaving e as the only antecedent.

$$
\begin{array}{l}
\cdot \vdash \mathit{empty} :: (s : \mathsf{store}_A) \\
s \leftarrow \mathit{empty} = \mathsf{case}\ s\ (\mathsf{ins} \Rightarrow x \leftarrow \mathsf{recv}\ s\ ; \qquad \% \quad x{:}A \vdash s : \mathsf{store}_A \\
\qquad\qquad\qquad\qquad\ n \leftarrow \mathit{empty}\ ; \qquad \% \quad (x{:}A)\ (n{:}\mathsf{store}_A) \vdash s : \mathsf{store}_A \\
\qquad\qquad\qquad\qquad\ e \leftarrow \mathit{elem} \leftarrow x\ n\ ; \quad \% \quad e{:}\mathsf{store}_A \vdash s : \mathsf{store}_A \\
\qquad\qquad\qquad\qquad\ \ldots \\
\qquad\qquad\qquad \mid \mathsf{del} \Rightarrow s.\mathsf{none}\ ;\ \mathsf{close}\ s) \\
\\
(x{:}A)\ (t{:}\mathsf{store}_A)\ \vdash \mathit{elem} :: (s : \mathsf{store}_A) \\
s \leftarrow \mathit{elem} \leftarrow x\ t = \ldots
\end{array}
$$

At this point we can implement s by e (the singleton store), which is just an application of the identity rule.

$\cdot \vdash \mathit{empty} :: (s : \mathsf{store}_A)$
$s \leftarrow \mathit{empty} = \mathsf{case}\ s\ (\mathsf{ins} \Rightarrow x \leftarrow \mathsf{recv}\ s\ ;$ $\quad\%\quad (x{:}A) \vdash s : \mathsf{store}_A$
$\qquad\qquad\qquad\qquad\qquad\quad n \leftarrow \mathit{empty}\ ;$ $\quad\%\quad (x{:}A)\ (n{:}\mathsf{store}_A) \vdash s : \mathsf{store}_A$
$\qquad\qquad\qquad\qquad\qquad\quad e \leftarrow \mathit{elem} \leftarrow x\ n$ $\quad\%\quad e{:}\mathsf{store}_A \vdash s : \mathsf{store}_A$
$\qquad\qquad\qquad\qquad\qquad\quad s \leftarrow e$
$\qquad\qquad\qquad\qquad\quad \mid \mathsf{del} \Rightarrow s.\mathsf{none}\ ;\ \mathsf{close}\ s)$

$(x{:}A)\ (t{:}\mathsf{store}_A) \vdash \mathit{elem} :: (s : \mathsf{store}_A)$
$s \leftarrow \mathit{elem} \leftarrow x\ t = \ldots$

It remains to write the code for the process holding an element of the store. We suggest you reconstruct or at least read it line by line the way we developed the definition of *empty*, but we will not break it out explicitly into multiple steps. However, we will still give the types after each interaction. For easy reference, we repeat the type definition for store_A.

$$\begin{array}{l}\mathsf{store}_A = \&\{\ \mathsf{ins} : A \setminus \mathsf{store}_A, \\ \qquad\qquad \mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : A \bullet \mathsf{store}_A\}\}\end{array}$$

$$\begin{array}{lll}
(x{:}A)\ (t{:}\mathsf{store}_A)\ \vdash \mathit{elem} :: (s : \mathsf{store}_A) & & \\
1 \quad s \leftarrow \mathit{elem} \leftarrow x\ t = & & \\
2 \qquad \mathsf{case}\ s\ (\mathsf{ins} \Rightarrow y \leftarrow \mathsf{recv}\ s\ ; & \% & (y{:}A)\ (x{:}A)\ (t{:}\mathsf{store}_A) \vdash s : \mathsf{store}_A \\
3 \qquad\qquad\qquad\quad t.\mathsf{ins}\ ; & \% & (y{:}A)\ (x{:}A)\ (t{:}A \setminus \mathsf{store}_A) \vdash s : \mathsf{store}_A \\
4 \qquad\qquad\qquad\quad \mathsf{send}\ t\ x\ ; & \% & (y{:}A)\ (t{:}\mathsf{store}_A) \vdash s : \mathsf{store}_A \\
5 \qquad\qquad\qquad\quad r \leftarrow \mathit{elem} \leftarrow y\ t\ ; & \% & r{:}\mathsf{store}_A \vdash s : \mathsf{store}_A \\
6 \qquad\qquad\qquad\quad s \leftarrow r & & \\
7 \qquad\qquad \mid \mathsf{del} \Rightarrow s.\mathsf{some}\ ; & \% & (x{:}A)\ (t{:}\mathsf{store}_A) \vdash s : A \bullet \mathsf{store}_A \\
8 \qquad\qquad\qquad\quad \mathsf{send}\ s\ x\ ; & \% & t{:}\mathsf{store}_A \vdash s : \mathsf{store}_A \\
9 \qquad\qquad\qquad\quad s \leftarrow t) & &
\end{array}$$

A few notes on this code. Look at the type at the end of the *previous* line to understand the next line.

- In line 2, we add $y{:}A$ at the left end of the context since $s : A \setminus \mathsf{store}_A$.
- In line 4, we can only pass x to t but not y, due restrictions of $\setminus L^*$.
- In line 5, y and t are in the correct order to call elem recursively.
- In line 8, we can pass x along s since it is at the left end of the context.

How does this code behave? Assume we have a store s holding elements x_1 and x_2 it would look like

$$\mathsf{proc}(s, s \leftarrow \mathit{elem} \leftarrow x_1\ t_1) \quad \mathsf{proc}(t_1, t_1 \leftarrow \mathit{elem} \leftarrow x_2\ t_2) \quad \mathsf{proc}(t_2, t_2 \leftarrow \mathit{empty})$$

where we have indicated the code executing in each process without unfolding the definition. If we insert an element along s (by sending ins and then a new y) then the process $s \leftarrow \mathit{elem} \leftarrow x_1\ t_1$ will insert x_1 along t_1 and then, in two steps, become $s \leftarrow \mathit{elem} \leftarrow y\ t_1$. Now the next process will pass x_2 along t_2 and hold on to x_1, and finally the process holding no element will spawn a new one (t_3) and itself hold on to x_2.

$$\begin{array}{l}\mathsf{proc}(s, s \leftarrow \mathit{elem} \leftarrow y\ t_1) \quad \mathsf{proc}(t_1, t_1 \leftarrow \mathit{elem} \leftarrow x_1\ t_2) \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{proc}(t_2, t_2 \leftarrow \mathit{elem} \leftarrow x_2\ t_3) \quad \mathsf{proc}(t_3, t_3 \leftarrow \mathit{empty})\end{array}$$

If we next delete an element, we will get y back and the store will effectively revert to its original state, with some (internal) renaming.

$\mathsf{proc}(s, s \leftarrow \mathit{elem} \leftarrow x_1\ t_2) \quad \mathsf{proc}(t_2, t_2 \leftarrow \mathit{elem} \leftarrow x_2\ t_3) \quad \mathsf{proc}(t_3, t_3 \leftarrow \mathit{empty})$

In essence, the store behaves like a *stack*: the most recent element we have inserted will be the first one deleted. If you carefully look through the intermediate types in the *elem* process, it seems that this behavior is forced. We conjecture that any implementation of the store interface we have given will behave like a stack or might at some point not respond to further messages. We do not yet have the means to carry out such a proof. Some related prior work might provide hints on how this might be proved using parametricity [Rey83, CPPT13].[1]

[1] If I or someone else in the class can prove or refute this conjecture, we may return to it in a future lecture.

2 Tail Calls

Let's look again at the two pieces of code we have written.

$$\mathsf{store}_A = \&\{\, \mathsf{ins} : A \setminus \mathsf{store}_A,\ \mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : A \bullet \mathsf{store}_A\}\}$$

$\cdot \vdash \mathit{empty} :: (s : \mathsf{store}_A)$

```
1   s ← empty =
2     case s (ins ⇒ x ← recv s ;          %  (x:A) ⊢ s : store_A
3                   n ← empty ;           %  (x:A) (n:store_A) ⊢ s : store_A
4                   e ← elem ← x n        %  e:store_A ⊢ s : store_A
5                   s ← e
6             | del ⇒ s.none ; close s)
```

$(x{:}A)\ (t{:}\mathsf{store}_A) \vdash \mathit{elem} :: (s : \mathsf{store}_A)$

```
7   s ← elem ← x t =
8       case s (ins ⇒ y ← recv s ;          %  (y:A) (x:A) (t:store_A) ⊢ s : store_A
9                     t.ins ;               %  (y:A) (x:A) (t:A \ store_A) ⊢ s : store_A
10                    send t x ;            %  (y:A) (t:store_A) ⊢ s : store_A
11                    r ← elem ← y t ;      %  r:store_A ⊢ s : store_A
12                    s ← r
13              | del ⇒s.some ;             %  (x:A) (t:store_A) ⊢ s : A • store_A
14                    send s x ;            %  t:store_A ⊢ s : store_A
15                    s ← t)
```

empty starts two new processes, in lines 3 and 4 and then terminates in line 5 by forwarding. *elem* spawns only one new process, in line 11, and then terminates in line 12 by forwarding. Intuitively, spawning a new process and then immediately forwarding to this process is wasteful, especially if process creation is an expensive operation.

It would be nice if the process executing *empty* could effectively just continue by executing *elem*, and similarly, if *elem* could continue as the same process once x has been sent along t. This can be achieved if we treat *tail calls* specially. So instead of writing

```
4       e ← elem ← x n
5       s ← e
```

we write

```
4       s ← elem ← x n
```

and similarly in the definition of *elem*.

In general, we compress a cut in the form of a process invocation followed by an identity simply as a process invocation:

$y \leftarrow X \leftarrow y_1 \ldots y_n$
$x \leftarrow y$

becomes

$x \leftarrow X \leftarrow y_1 \ldots y_n$

This is analogous to the so-called *tail-call optimization* in functional languages where instead of f calling a function g and immediately returning its value, f just continues as g. This is often represented as saving stack space since it can be implemented as a jump instead of a call. Here, too, recursively defined processes executing a sequence of interactions can simply continue without spawning a new process and then forwarding the result immediately, thereby saving process invocations.

From now on, we will often silently use the compressed form. Of course, its purely logical meaning can be recovered by expanding it into a cut followed by an identity.

3 Analyzing Parallel Complexity

We can analyze various complexity measures of our implementations. For example, we can count the number of processes that execute. Any call (except for a tail call) will spawn a new process, and any forward and close will terminate a process. Looking at the code below we can see that inserting an element into a store will spawn exactly one new process, namely when we eventually insert the last element into the empty store. Deleting an element will terminate exactly one process: either the empty one, or the one holding the element we are returning. Therefore in a store with n elements there will be *exactly* $n + 1$ processes.

$$\mathsf{store}_A = \&\{\, \mathsf{ins} : A \setminus \mathsf{store}_A,\ \mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : A \bullet \mathsf{store}_A\}\}$$

$\cdot \vdash \mathit{empty} :: (s : \mathsf{store}_A)$

```
1   s ← empty =
2     case s (ins ⇒ x ← recv s ;          %  (x:A) ⊢ s : store_A
3                   n ← empty ;           %  (x:A) (n:store_A) ⊢ s : store_A
4                   s ← elem ← x n
5             | del ⇒ s.none ; close s)
```

$(x{:}A)\ (t{:}\mathsf{store}_A) \vdash \mathit{elem} :: (s : \mathsf{store}_A)$

```
6   s ← elem ← x t =
7       case s (ins ⇒ y ← recv s ;        %  (y:A) (x:A) (t:store_A) ⊢ s : store_A
8                     t.ins ;             %  (y:A) (x:A) (t:A \ store_A) ⊢ s : store_A
9                     send t x ;          %  (y:A) (t:store_A) ⊢ s : store_A
10                    s ← elem ← y t
11                | del ⇒s.some ;         %  (x:A) (t:store_A) ⊢ s : A • store_A
12                    send s x ;          %  t:store_A ⊢ s : store_A
13                    s ← t)
```

Another interesting measure is the *reaction time* which is analogous to the *span* complexity measure for parallel programs. If we try to carry out two consecutive operations, how many steps must elapse between them, assuming maximal parallelism? Here it is convenient to count every interaction as a step and no other costs.

Looking at the code for *elem* we see that there are only two interactions along channel t until the *elem* process can interact again along s after it has received ins and y. For *empty* there is only one spawn but no other

interactions. Moreover, there is no delay for a deletion, since the process will respond immedidately along s.

In aggregate, when we store n elements consecutively, the constant reaction time means that there will be n elements building up the internal data structure simultaneously. No matter how many insertions and deletions we carry out, the reaction time (measured in total system interactions assuming maximal parallelism) is always constant.

On the other hand, if we count the total number of interactions of the system taking place (ignoring any question of parallelism) we see that for n insertions it will be $O(n^2)$, since each new element initiates a chain reaction that reaches to the end of the chain of elements. This is usually called the *work* performed by the algorithm.

4 Queues

As notes, our implementation so far ended up behaving like a stack, and we conjectured that the type of the interface itself forced this behavior. Can we modify the type to allow (and perhaps force) the behavior of the store as a queue, where the first element we store is the first one we receive back? I encourage you to try to work this out before reading on ...

The key idea is to change the type

$$\begin{array}{l}\mathsf{store}_A = \&\{\ \mathsf{ins} : A \setminus \mathsf{store}_A, \\ \qquad\qquad \mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : A \bullet \mathsf{store}_A\}\}\end{array}$$

to

$$\begin{array}{l}\mathsf{queue}_A = \&\{\ \mathsf{ins} : \mathsf{queue}_A \,/\, A, \\ \qquad\qquad \mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : A \bullet \mathsf{queue}_A\}\}\end{array}$$

We will not go through this in detail, but reading the following code and the type after each interaction should give you a sense for what this change entails.

$\cdot \vdash \mathit{empty} :: (s : \mathsf{queue}_A)$

```
1   s ← empty =
2     case s (ins ⇒ x ← recv s ;          %  x:A ⊢ s : queue_A
3                 n ← empty ;             %  (x:A) (n:queue_A) ⊢ s : queue_A
4                 s ← elem ← x n
5             | del ⇒ s.none ; close s)
```

$(x{:}A)\ (t{:}\mathsf{queue}_A) \vdash \mathit{elem} :: (s : \mathsf{queue}_A)$

```
6   s ← elem ← x t =
7       case s (ins ⇒ y ← recv s ;          %  (x:A) (t:queue_A) (y:A) ⊢ s : queue_A
8                     t.ins ;               %  (x:A) (t:queue_A / A) (y:A) ⊢ s : queue_A
9                     send t y ;            %  (x:A) (t:queue_A) ⊢ s : queue_A
10                    s ← elem ← x t
11                | del ⇒ s.some ;          %  (x:A) (t:queue_A) ⊢ s : A • queue_A
12                    send s x ;            %  t:queue_A ⊢ s : queue_A
13                    s ← t)
```

The critical changes are in line 7 (where y is added to the *right end* of the antecedents instead of the left) and line 9 (where consequently y instead of x must be sent along t).

The complexity of all the operations remains the same, since the only difference is whether the current x or the new y is sent along t, but the implementation now behaves like a queue rather than a stack.

Exercises

Exercise 1 In this exercise we explore an alternative implementation of stacks. First, consider type of stacks (renamed from store_A in this lecture)

$$\begin{aligned}\mathsf{stack}_A = \&\{\ &\mathsf{ins} : A \setminus \mathsf{stack}_A,\\ &\mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : A \bullet \mathsf{stack}_A\}\}\end{aligned}$$

1. Provide definitions for

   $$\cdot \vdash \mathit{stack\_new} :: (s : \mathsf{stack}_A)$$
   $$l{:}\mathsf{list}_A \vdash \mathit{stack} :: (s : \mathsf{stack}_A)$$

   which represents the elements of the stack in a list. If you need auxiliary process definitions for lists, please state them clearly, including their type.

2. Repeat the analysis of Section 3:

   (a) How many processes execute for a stack with n elements?

   (b) What is the reaction time for an insertion or deletion given a stack with n elements?

   (c) What is the total work for each insertion or deletion given a stack with n elements?

Exercise 2 In this exercise we explore an alternative implementation of queues. First, recall the type of queues from Section 4.

$$\begin{aligned}\mathsf{queue}_A = \&\{\ &\mathsf{ins} : \mathsf{queue}_A \,/\, A,\\ &\mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : A \bullet \mathsf{queue}_A\}\}\end{aligned}$$

1. Provide definitions for

   $$\cdot \vdash \mathit{queue\_new} :: (s : \mathsf{queue}_A)$$
   $$l{:}\mathsf{list}_A \vdash \mathit{queue} :: (s : \mathsf{queue}_A)$$

   which represents the elements of the queue in a list. If you need auxiliary process definitions for lists, please state them clearly, including their type.

2. Repeat the analysis of Section 3:

(a) How many processes execute for a queue with n elements?

(b) What is the reaction time for an insertion or deletion given a queue with n elements?

(c) What is the total work for each insertion or deletiion given a queue with n elements?

Exercise 3 In this exercise we will "turn around" Exercise 1. Write a process definition

$$s{:}\mathsf{stack}_A \vdash \mathsf{to\_list} :: (l{:}\mathsf{list}_A)$$

which converts a stack into a list. As far as you can tell, is the order of the elements that are sent along l fixed?

Exercise 4 Consider the standard functional programming technique of implementing a queue with two lists. Just briefly, we have an *input list in* to which we add elements when they are enqueued and an *output list out* from which we take elements when they are dequeued. When the output list becomes empty, we reverse the input list, adding each element in turn onto the output list. Initially, both lists are empty.

Explore if you can write such an implementation against the queue interface from Section 4. The implementation should have one of the two types

$$(\mathit{in}{:}\mathsf{list}_A)\ (\mathit{out}{:}\mathsf{list}_A) \vdash \mathit{queue2} :: (s : \mathsf{queue}_A)$$
$$(\mathit{out}{:}\mathsf{list}_A)\ (\mathit{in}{:}\mathsf{list}_A) \vdash \mathit{queue2} :: (s : \mathsf{queue}_A)$$

References

[CPPT13] Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. Behavioral polymorphism and parametricity in session-based communication. In M.Felleisen and P.Gardner, editors, *Proceedings of the European Symposium on Programming (ESOP'13)*, pages 330–349, Rome, Italy, March 2013. Springer LNCS 7792.

[Rey83] John C. Reynolds. Types, abstraction, and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing 83*, pages 513–523. Elsevier, September 1983.

Lecture Notes on Ordered Programming

15-816: Substructural Logics
Frank Pfenning

Lecture 10
Thursday, September 29

We begin the lecture by continuing to program in ordered logic using the example of *list segments*, whose imperative versions have recently become popular in verification.

Then we return to the operational reading of the original, general rules for $\backslash L$, $/L$, $\bullet R$ and $\circ R$ in response to a question from an earlier lecture.

Finally we will sketch the proof of *progress* for ordered logic considered as a programming language.

1 List Segments

A list segment is the beginning of a list without its tail. It becomes a list once a tail is supplied. Functionally, it can be seen as function from a list to a list; here it will be a process that when given a list on its right will behave like a list.

$$\mathsf{seg}_A = \mathsf{list}_A \mathbin{/} \mathsf{list}_A$$

We discussed this in response to a question on how we might get direct access to the end of a queue, since our implementation of the queue_A interface actually had to pass any newly inserted element down a chain of processes. As we will see it doesn't exactly serve the same purpose, but first let's program.

We begin with the empty segment. If we append a tail t the empty segment becomes the list t. Our definition of seg_A is transparent, so that we will silently replace it by its definition as was our habit earlier in this course.

$\cdot \vdash \mathit{empty} :: (s : \mathsf{seg}_A)$
$s \leftarrow \mathit{empty} =$
  $t \leftarrow \mathsf{recv}\ s\ ;$ % $\cdot \vdash s : \mathsf{list}_A\ /\ \mathsf{list}_A$
  $s \leftarrow t$ % $t{:}\mathsf{list}_A \vdash s{:}\mathsf{list}_A$

Concatenating two lists is straightforward, and the code more or less writes itself if we heed the types.

$(s_1{:}\mathsf{seg}_A)\ (s_2{:}\mathsf{seg}_A) \vdash \mathit{concat} :: (s : \mathsf{seg}_A)$
$s \leftarrow \mathit{concat} \leftarrow s_1\ s_2 =$
  $t \leftarrow \mathsf{recv}\ s\ ;$ % $(s_1{:}\mathsf{list}_A\ /\ \mathsf{list}_A)\ (s_2{:}\mathsf{list}_A\ /\ \mathsf{list}_A)\ (t{:}\mathsf{list}_A) \vdash s : \mathsf{list}_A$
  $\mathsf{send}\ s_2\ t\ ;$ % $(s_1{:}\mathsf{list}_A\ /\ \mathsf{list}_A)\ (s_2{:}\mathsf{list}_A) \vdash s : \mathsf{list}_A$
  $\mathsf{send}\ s_1\ s_2\ ;$ % $s_1{:}\mathsf{list}_A \vdash s : \mathsf{list}_A$
  $s \leftarrow s_1$

Next, we can *prepend* an element to a segment to obtain another segment, which means we add it to the *front* of the given segment.

$(x{:}A)\ (s'{:}\mathsf{seg}_A) \vdash \mathit{prepend} :: (s : \mathsf{seg}_A)$
$s \leftarrow \mathit{prepend} \leftarrow x\ s' =$
  $t \leftarrow \mathsf{recv}\ s\ ;$ % $(x{:}A)\ (s'{:}\mathsf{list}_A\ /\ \mathsf{list}_A)\ (t{:}\mathsf{list}_A) \vdash s : \mathsf{list}_A$
  $\mathsf{send}\ s'\ t\ ;$ % $(x{:}A)\ (s'{:}\mathsf{list}_A) \vdash s : \mathsf{list}_A$
  $s'' \leftarrow \mathsf{cons} \leftarrow x\ s'$ % $s''{:}\mathsf{list}_A \vdash s : \mathsf{list}_A$
  $s \leftarrow s''$

For reasons of symmetry with the next case, we have not combined the last two lines into the simpler tail call $s \leftarrow \mathsf{cons} \leftarrow x\ s'$.

Appending an element to the end of a segment is similar. It will still come before (the absent) tail. This is right behavior, since a segment with x appended still accepts a tail to come *after* x. Note that we have to be careful to state the arguments to *postpend* in the right order.

$(s'{:}\mathsf{seg}_A)\ (x{:}A) \vdash \mathit{postpend} :: (s : \mathsf{seg}_A)$
$s \leftarrow \mathit{postpend} \leftarrow s'\ x =$
  $t \leftarrow \mathsf{recv}\ s\ ;$ % $(s'{:}\mathsf{list}_A\ /\ \mathsf{list}_A)\ (x{:}A)\ (t{:}\mathsf{list}_A) \vdash s : \mathsf{list}_A$
  $t' \leftarrow \mathsf{cons} \leftarrow x\ t$ % $(s'{:}\mathsf{list}_A\ /\ \mathsf{list}_A)\ (t'{:}\mathsf{list}_A) \vdash s : \mathsf{list}_A$
  $\mathsf{send}\ s'\ t'\ ;$ % $s'{:}\mathsf{list}_A \vdash s : \mathsf{list}_A$
  $s \leftarrow s'$

What can we do with a segment? We can create an empty segment and then add elements to the left and right ends. In that way, it is almost like a double-ended queue. However, we cannot remove elements. Instead, at

some point we need to convert the element to a list by appending an empty list, after which we can no longer (easily) access the tail (see Exercise 1).

$s{:}\mathsf{seg}_A \vdash \mathit{seg\_to\_list} :: (l : \mathsf{list}_A)$
$l \leftarrow \mathit{seg\_to\_list} \leftarrow s =$
  $n \leftarrow \mathit{nil}$; % $(s{:}\mathsf{list}_A \,/\, \mathsf{list}_A)\ (n{:}\mathsf{list}_A) \vdash l : \mathsf{list}_A$
  $\mathsf{send}\ s\ n$; % $s{:}\mathsf{list}_A \vdash l : \mathsf{list}_A$
  $l \leftarrow s$

2 An Operational Reading of $\bullet R$

Recall that we restricted the general form $\bullet R$ to $\bullet R^*$:

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

In the presence of cut and identity, we can interderive these rules so the same sequents remain provable.

$$\frac{\dfrac{}{A \vdash A}\ \mathsf{id}_A \quad \Omega_2 \vdash B}{A\ \Omega_2 \vdash A \bullet B}\ \bullet R \qquad \frac{\Omega_1 \vdash A \quad \dfrac{\Omega_2 \vdash B}{A\ \Omega_2 \vdash A \bullet B}\ \bullet R^*}{\Omega_1\ \Omega_2 \vdash A \bullet B}\ \mathsf{cut}_A$$

But what is the right operational reading of the fully general rule, and how does it relate to the more restricted one? Let's take a look at the operational interpretation of the proof on the right, which is available to us in ordered programming since it only uses $\bullet R^*$. From this, we may be able to glean the right interpretation of $\bullet R$.

$$\frac{\Omega_1 \vdash Q_y :: (y : A) \quad \dfrac{\Omega_2 \vdash P :: (x : B)}{(y{:}A)\ \Omega_2 \vdash (\mathsf{send}\ x\ y\ ;\ P) :: (x : A \bullet B)}\ \bullet R^*}{\Omega_1\ \Omega_2 \vdash (y \leftarrow Q_y\ ;\ \mathsf{send}\ x\ y\ ;\ P) :: (x : A \bullet B)}\ \mathsf{cut}_A$$

The program here will create a new channel w, spawn a new process Q_w, send w along x and continue as P. We invent a new notation for the general $\bullet R$ to accomplish the same interactions.

$$\frac{\Omega_1 \vdash Q :: (y : A) \quad \Omega_2 \vdash P :: (x : B)}{\Omega_1\ \Omega_2 \vdash (\mathsf{send}\ x\ (y \leftarrow Q_y)\ ;\ P) :: (x : A \bullet B)}\ \bullet R$$

In an *asynchronous* model of communication these two behave exactly the same. Under a *synchronous* interpretation, there is a small difference. Here is the generalized communication rule for $\bullet$ (which always creates a fresh channel in the conclusion):

$$\frac{\mathsf{proc}(x, \mathsf{send}\ x\ (y \leftarrow Q_y)\ ;\ P) \quad \mathsf{proc}(z, y \leftarrow \mathsf{recv}\ x\ ;\ R_y)}{\mathsf{proc}(x, P) \quad \mathsf{proc}(w, Q_w) \quad \mathsf{proc}(z, R_w)}\ \bullet C^w$$

We can see that Q_w does not start to execute until this synchronous communication actually takes place. However, when executing our derived form (on the top line)

$$\frac{\mathsf{proc}(x, y \leftarrow Q_y\ ;\ \mathsf{send}\ x\ y\ ;\ P)}{\mathsf{proc}(x, \mathsf{send}\ x\ w\ ;\ P) \quad \mathsf{proc}(w, Q_w)}\ \mathsf{cmp}^w$$

a new channel w will always be created and Q_w starts immediately, whether a client is ready to communicate along x or not. If there is a client ready it can then immediately step to the same configuration as the $\bullet C^w$ rule.

With appropriate small changes, the same construction can be used for $\circ R$, $/L$ and $\backslash L$ (see Exercise 4).

Why did we choose the simplified forms of these constructs? One reason is that all the logical rules just send or receive some information and continue, but do nothing else. The rules for internal ($\oplus$) and external ($\&$) choice send or receive a label, the rules for the unit (**1**) send or receive a end-of-communication token, the rules for $/$, $\backslash$, $\bullet$, and $\circ$ send or receive a channel. Composition (cut) is solely responsible for spawning new processes and forwarding (id) terminates a process (as does $\mathbf{1}R$, for a different reason).

With the general rules, the multiplicative connectives ($/, \backslash, \bullet, \circ$) also spawn new processes, which complicates reasoning about their operational behavior. The big advantage, however, is that the rules are directly derived from the sequent calculus and therefore satisfy full cut elimination, which, as we have seen, is not the case for the restricted rules. But since we view them operationally, from the perspective of a programming language, we are more interested in progress and preservation properties. And these still hold, because cut reduction and identity expansion still hold.

3 Progress

We now would like to generalize the proof of progress from the subsingleton to the ordered case. This means the scaffolding around the key insight,

namely that cut reduction holds for each connective, has to be generalized.

We begin by typing configurations. While configurations are *linear* rather than *ordered*, we still need type checking to proceed in some order. So we may need to "permute" the configuration so that the following rules apply. The typing derivation for a configuration then fixes some order. At the top level we use this for a configuration executing a single process offering along a single channel.

$$\models \mathsf{proc}(x, P) :: (x : A)$$

We need to generalize almost immediately if P spawns a new process. Then then have

$$\models \mathcal{C} :: (x : A)$$

that is, we have multiple running processes but, overall, we can interact with the configuration only along one channel. If P spans multiple processes, say $P_1, \ldots, P_n$, then we need to check all of them. While each of them offers along a single channel, collectively they offer along a whole sequence of channels, so we end up with the judgment

$$\models \mathcal{C} :: \Omega$$

allowing both multiple processes in the configuration and multiple channels that they provide, namely all the ones in Ω. This judgment is defined by the following two rules:

$$\frac{}{\models (\cdot) :: (\cdot)} \qquad \frac{\models \mathcal{C} :: \Omega\ \Omega' \quad \Omega' \vdash P :: (x : A)}{\models \mathcal{C}\ \mathsf{proc}(x, P) :: \Omega\ (x{:}A)}$$

This means that our particular arrangement of the configuration will have to list processes in dependency order, with a provider always preceding (looking left to right) its client. This corresponds to a pre-order traversal of the dependency tree where we traverse the subtrees from right to left.[1]

The key is now to come up with the correct induction hypothesis to prove progress. We introduce the in-line notation $\mathcal{C} \longrightarrow \mathcal{D}$ for the clumsier

$$\frac{\mathcal{C}}{\mathcal{D}}$$

We need one lemma, whose role will only become clear in the proof. It allows us to extract a typing derivation for the offering process for a channel $x{:}A$ that's provided by a configuration.

[1] In lecture, I had a slightly more general rule which also worked out, but was unnecessarily complicated (as suggested by several students).

Lemma 1 (Configuration Inversion)
If $\models \mathcal{C} :: \Omega_1\ (x{:}A)\ \Omega_2$ *then* $\mathcal{C} = (\mathcal{C}_1\ \mathsf{proc}(x, P)\ \mathcal{C}_2)$
with $\models \mathcal{C}_1 :: \Omega_1\ \Omega'$ *and* $\Omega' \vdash P :: (x : A)$ *for some* $\mathcal{C}_1$, P, $\mathcal{C}_2$, *and* Ω'.

Proof: By induction on the structure of the given typing derivation (see Exercise 5). □

Theorem 2 *If* $\models \mathcal{C} :: \Omega$ *then either*

(i) $\mathcal{C} \longrightarrow \mathcal{D}$ *for some* $\mathcal{D}$, *or*

(ii) all processes in $\mathcal{C}$ *are executing a right rule.*

Proof: By induction on the structure of the typing derivation $\models \mathcal{C} :: \Omega$.

Case: The empty configuration. Then *(i)* holds.

Case: $\mathcal{C} = \mathcal{C}'\ \mathsf{proc}(x, P)$ and

$$\models \mathcal{C}' :: \Omega'\ \Omega^* \qquad\qquad \Omega^* \vdash P :: (x{:}A)$$

We first consider some subcases for P.

Subcase: P ends in a cut. Then *(i)* holds because P can transition.

Subcase: P ends in an identity. Then *(i)* holds because P can transition.

Subcase: P is a defined name. Then *(i)* holds because P can transition.

In the remaining cases we can now assume that P is not a cut, identity, or name. In other words, it must end in a logical rule. We now appeal to the induction hypothesis and get two further subcases.

Subcase: $\mathcal{C}' \longrightarrow \mathcal{D}'$ for some $\mathcal{D}'$. Then also $\mathcal{C} \longrightarrow \mathcal{D}'\ \mathsf{proc}(x, P)$ so *(i)* holds.

Subcase: All processes in $\mathcal{C}'$ execute a right rule.

At this point we further descend in our tree of case distinctions: P executes either a right rule or a left rule.

Subcase: P executes a right rule. Then all processes in $\mathcal{C} = \mathcal{C}'\ \mathsf{proc}(x, P)$ execute a right rule and *(ii)* holds.

Subcase: P executes a left rule.

Sadly, we now have to make further distinctions as to which left rule P executes. We consider only one case; the others are analogous.

Subcase: P executes $\backslash L^*$. Then $P = (\mathsf{send}\ y\ w\ ;\ P')$,
$\Omega^* = \Omega^*_L\ (w{:}B)\ (y : B \setminus C)\ \Omega^*_R$,

$$\frac{\Omega^*_L\ (y{:}C)\ \Omega^*_R \vdash P' :: (x : A)}{\Omega^*_L\ (w{:}B)\ (y : B \setminus C)\ \Omega^*_R \vdash \mathsf{send}\ y\ w\ ;\ P' :: (x : A)}\ \backslash L^*$$

and

$$\models \mathcal{C}' :: \Omega'\ \Omega^*_L\ (w{:}B)\ (y : B \setminus C)\ \Omega^*_R$$

By configuration inversion (Lemma 1) the derivation of the latter contains a typing derivation for the provider of $y : B \setminus C$

$$\Omega'' \vdash Q :: (y : B \setminus C)$$

for some Ω'' and Q. We also know in this subcase that Q executes a right rule. By inversion, this must be $\backslash R$, and we get $Q = (z \leftarrow \mathsf{recv}\ y\ ;\ Q'_z)$ and

$$\frac{(z{:}B)\ \Omega'' \vdash Q'_z :: (y : C)}{\Omega'' \vdash (z \leftarrow \mathsf{recv}\ y\ ;\ Q'_z) :: (y : B \setminus C)}\ \backslash R$$

Unraveling this, the original configuration has the form

$$\mathcal{C} = \mathcal{C}'_1\ \mathsf{proc}(y, z \leftarrow \mathsf{recv}\ y\ ;\ Q'_z)\ \mathcal{C}'_2\ \mathsf{proc}(x, \mathsf{send}\ y\ w\ ;\ P')$$

where $\mathcal{C}'_1$ and $\mathcal{C}'_2$ come from the appeal to configuration inversion. Hence we can finally infer

$$\frac{\mathcal{C}'_1\ \mathsf{proc}(y, z \leftarrow \mathsf{recv}\ y\ ;\ Q'_z)\ \mathcal{C}'_2\ \mathsf{proc}(x, \mathsf{send}\ y\ w\ ;\ P')}{\mathcal{C}'_1\ \mathsf{proc}(y, Q'_w)\ \mathcal{C}'_2\ \mathsf{proc}(x, P')}\ \backslash C$$

and clause *(i)* holds.

□

Exercises

Exercise 1 Write an ordered program to convert a list back into a segment.

$$l{:}\mathsf{list}_A \vdash \mathit{list\_to\_seg} :: (s : \mathsf{seg}_A)$$

Characterize work and span (= reaction time under maximal parallelism) of this operation when provided with a list of length n?

Exercise 2 We cannot write a direct analogue for the functional *map* from a functional language, because the function f that is mapped over a data structure is used potentially many times, violating linearity and order. In order to circumvent this problem, we define a *mapper* to be a process that can transform inputs of type A to outputs of B arbitrarily often.

$$\mathsf{mapper}_{AB} = \&\{\mathsf{next} : (B \bullet \mathsf{mapper}_{AB}) \,/\, A, \mathsf{done} : \mathbf{1}\}$$

1. Define a process *map* with type

   $$(m{:}\mathsf{mapper}_{AB})\ (l{:}\mathsf{list}_A) \vdash \mathit{map} :: (k : \mathsf{list}_B)$$

   that applies mapper m to each element of list l to produce list k.

2. Define a mapper *map_id* such that

   $l{:}\mathsf{list}_A \vdash \mathit{identity} :: (k : \mathsf{list}_A)$
   $k \leftarrow \mathit{identity} \leftarrow l =$
   $\quad m \leftarrow \mathit{map\_id}\ ;$
   $\quad k \leftarrow \mathit{map} \leftarrow m\ l$

   does indeed behave like the identity. State both the type of *map_id* and its definition.

Exercise 3 We cannot write a direct process analogue of the function *fold* for the same reason as for *map* in Exercise 2.

1. Devise a type folder_{AB} that can reduce a list of type A to a result of type B. We suggest you study mapper_{AB} from Exercise 2 for hints on how to proceed.

2. Define a process *fold* with type

   $$(b{:}B)\ (m{:}\mathsf{folder}_{AB})\ (l{:}\mathsf{list}_A) \vdash \mathit{fold} :: (r : B)$$

   that folds m over the list l with initial value b to produce r. Feel free to change the order of the antecedents if another order turns out to be more convenient.

3. Define a folder *fold_id* and complete the following program such that

   l:list$_A \vdash$ *identity* :: $(k$: list$_A)$
   $k \leftarrow$ *identity* $\leftarrow l =$
   $f \leftarrow$ *fold_id* ;
   ...

   does indeed behave like the identity. State both the type of *fold_id* and its definition.

Exercise 4 Give a direct proof term assignment for the general $/L$ using the new form of process

$$\mathsf{send}\ x\ (y \leftarrow Q_y)\ ;\ P$$

from Section 2.

Exercise 5 Prove configuration inversion (Lemma 1).

Lecture Notes on Linear Logic

15-816: Substructural Logics
Frank Pfenning

Lecture 11
October 4, 2016

In this lecture we introduce the sequent calculus for the multiplicative-additive fragment of linear logic [**?**], often abbreviated as MALL. From intuitionistic linear logic it is only missing the exponential, $!A$, which, as an antecedent, allows A to be used arbitrarily many times. The exponential will be introduced next week, in Lecture 13.

I have to admit this is not actually what happened in lecture when in addition to introducing linear logic I also tried to talk about combining logics. I slightly revised my—ahem—somewhat cryptic approach and tried again in Lecture 12.

1 Exchange

The way we obtain linear logic from ordered logic is just to allow exchange among the antecedents of a sequent. This is importing into the sequent calculus the difference between ordered and linear inference from the beginning of the course. Allowing exchange can be formalized in two ways: with an explicit inference rule

$$\frac{\Omega_L\ B\ A\ \Omega_R \vdash C}{\Omega_L\ A\ B\ \Omega_R \vdash C}\ \mathsf{exchange}$$

or we can treat the context as a multiset rather than a sequence of propositions. We write

$$\Delta = (A_1, \ldots, A_n)$$

for antecedents that form multisets and for now reserve Ω for ordered antecedents. We will usually use the latter approach to reduce the bureaucracy of explicit inference rules. The comma separating antecedents and the use of the letter Δ will remind us that "*order doesn't matter*".

2 Collapsing Connectives

When we identify antecedents up to exchange, several previously distinct connectives become indistinguishable. For example, $A \setminus B = B \,/\, A$ become the single connective of *linear implication*, written $A \multimap B$. Similarly, $A \bullet B = A \circ B$ becomes *multiplicative conjunction* (or *simultaneous conjunction*), written $A \otimes B$. All the other connectives remain the same. This information can be gleaned from the inference rules. We only show one example, the emergent rules for linear implication.

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

In the latter rule we write $A \multimap B$ on the right end of the antecedents, but due to exchange we could have written it anywhere. Similarly, writing Δ, Δ' means (reading the rule bottom-up as we are used to) that we take a multiset of antecedents and split it into two. Thus there are 2^n ways to split a context of n antecedents, while in the ordered case there are only $n + 1$ ways.

3 Cut Reduction and Identity Expansion

The properties of cut reduction and identity expansion carry over. Identity expansion is automatic, since we only relax the applicability of rules. Cut reduction also works exactly as before, because the only difference is a relaxed order among antecedents. Similarly, the admissibility of cut in the cut-free sequent calculus works just as before, with the exact same induction measure and argument. In fact, we will unify the different proofs into a single one in a future lecture.

4 Operational Interpretation

The assignment of proof terms and processes remains *exactly* the same as for the ordered case. In other words, linear processes (as compared to or-

dered ones) only relax the requirements on typing of processes proofs while retaining their operational meaning from ordered logic. Therefore we do not need to introduce any new rules.

In principle, we would have to reprove progress and preservation, since the more relaxed form of typing a priori might be too weak to guarantee them. However, the proof follows the exact same patterns as before and contains no new insights.

As an example, we reconsider lists in their linear rather than ordered form and show that they can be reversed.

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

Linearity guarantees that elements of a list are preserved, although their order may not. To define *reverse* we use an auxiliary process *rev* which has an accumulator argument in which we construct the reversed list.

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

$(k{:}\mathsf{list}_A)\ (a{:}\mathsf{list}_A) \vdash \mathit{rev} :: (l{:}\mathsf{list}_A)$

$$
\begin{array}{lll}
l \leftarrow \mathit{rev} \leftarrow k\ a = & & \\
\quad \mathsf{case}\ k\ \ (\mathsf{cons} \Rightarrow x \leftarrow \mathsf{recv}\ k\ ; & \% & (x{:}A)\ (k{:}\mathsf{list}_A)\ (a{:}\mathsf{list}_A) \vdash (l{:}\mathsf{list}_A) \\
\qquad\qquad\qquad\quad a' \leftarrow \mathit{cons} \leftarrow x\ a\ ; & \% & (k{:}\mathsf{list}_A)\ (a'{:}\mathsf{list}_A) \vdash (l{:}\mathsf{list}_A) \\
\qquad\qquad\qquad\quad l \leftarrow \mathit{rev} \leftarrow k\ a' & & \\
\qquad\quad \mid \mathsf{nil} \Rightarrow \mathsf{wait}\ k\ ;\ l \leftarrow a' & \% & \text{accumulator becomes result}
\end{array}
$$

$k{:}\mathsf{list}_A \vdash \mathit{reverse} :: (l{:}\mathsf{list}_A)$

$$
\begin{array}{lll}
l \leftarrow \mathit{reverse} \leftarrow k & & \\
\quad n \leftarrow \mathit{nil}\ ; & & \\
\quad l \leftarrow \mathit{rev} \leftarrow k\ n & \% & \text{initialize accumulator with } \mathit{nil}
\end{array}
$$

Observe that the call to *cons* could not be typed in the case of ordered lists.

Exercises

Exercise 1 Show one principal and one commutative case among the $\multimap$ and $\otimes$ connectives in the proof of the admissibility of cut for linear logic.

Exercise 2 A linear proposition can be turned into an ordered proposition by deciding for each muliplicative conjunction ($\otimes$) if it should become fuse ($\bullet$) or twist ($\circ$) and for each linear implication ($\multimap$) if it should become under ($\backslash$) or over ($/$). For example, we can translate the provable $(A \otimes (A \multimap B)) \multimap B$ into the provable $(A \bullet (A \backslash B)) \,/\, B$.

1. If possible, construct a provable linear formula containing only $\otimes$, $\multimap$ and propositional variables such that all of its ordered translations are provable in ordered logic. If one exists, try to minimize the number of its connectives.

2. If possible, construct a provable linear formula such that none of its translations to ordered logic are provable. If one exists, try to minimize the number of its connectives.

Exercise 3 Analyze the parallel complexity of the *rev* and *reverse* processes in **??** in terms of latency and throughput. Given a list k of length n that produces its elements with a constant delay c between consecutive elements:

1. (*Latency*) How many steps (counting only communications, but not spawns or forwards) until the first element can be retrieved from l?

2. (*Throughput*) How many steps until the whole reversed lists can be retrieved from l, assuming a client that has no delay between successive interactions along l.

3. If we reverse twice

$$l \leftarrow \mathsf{reverse} \leftarrow k \; ; \; k' \leftarrow \mathsf{reverse} \leftarrow l$$

   the result k' should be observationally equivalent to the given k. Analyze latency and throughput for the pipeline from k to k'.

References

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

Lecture Notes on Combining Logics

15-816: Substructural Logics
Frank Pfenning

Lecture 12
October 6, 2016

In this lecture we will introduce a general approach to combining logics, using the special case of ordered and linear logic. Other examples of this construction are *monads* in the sense used in functional programming to integrate effects [Mog89] and LNL [Ben94] and the modal logic S4 which gives rise a type system for quotation [DP01, PD01]. A systematic logical study was initiated by Reed [Ree09] and subsequently extended and applied to concurrency [PG15].

1 Connecting Two Layers with Shifts

We start by writing down ordered and linear logic in a way that exhibits the correspondence between the propositions. We use a subscript O to indicate an *ordered mode* and L to indicate *linear mode*. In a later lecture we will introduce other modes. p_{O} and p_{L} stand for ordered and linear propositional variables, respectively.

$$\begin{array}{lll} \text{Linear} & A_{\mathsf{L}}, B_{\mathsf{L}} ::= & p_{\mathsf{L}} \mid A_{\mathsf{L}} \oplus B_{\mathsf{L}} \mid A_{\mathsf{L}} \mathbin{\&} B_{\mathsf{L}} \mid \mathbf{1} \mid A_{\mathsf{L}} \multimap B_{\mathsf{L}} \mid A_{\mathsf{L}} \otimes B_{\mathsf{L}} \\ \text{Ordered} & A_{\mathsf{O}}, B_{\mathsf{O}} ::= & p_{\mathsf{O}} \mid A_{\mathsf{O}} \oplus B_{\mathsf{O}} \mid A_{\mathsf{O}} \mathbin{\&} B_{\mathsf{O}} \mid \mathbf{1} \mid A_{\mathsf{O}} \setminus B_{\mathsf{O}} \mid B_{\mathsf{O}} / A_{\mathsf{O}} \mid A_{\mathsf{O}} \bullet B_{\mathsf{O}} \mid A_{\mathsf{O}} \circ B_{\mathsf{O}} \end{array}$$

Strictly speaking, there are also two versions of internal and external choice and the unit $\mathbf{1}$, but since their rules and behavior are independent of whether we have ordered or unordered antecedents, so we write them the same way. At this point there is no overlap, so the two logics are not connected in any way. To allow ordered proposition to mention linear ones and vice versa, we add two *shift* operators. Since we view linear propositions as

more powerful antecedents (they can move around, after all, while the ordered ones are locked into place) we think of L as stronger than O (written $\mathsf{L} > \mathsf{O}$) and we use an uparrow to go from ordered to linear and a downarrow to go from linear to ordered. Officially, this arrow is annotated with the two modes. We will omit them in today's lecture since O and L will be the only modes.

$$\begin{array}{lll} \text{Linear} & A_\mathsf{L}, B_\mathsf{L} ::= p_\mathsf{L} \mid A_\mathsf{L} \oplus B_\mathsf{L} \mid A_\mathsf{L} \mathbin{\&} B_\mathsf{L} \mid \mathbf{1} \mid A_\mathsf{L} \multimap B_\mathsf{L} \mid A_\mathsf{L} \otimes B_\mathsf{L} \mid \uparrow^\mathsf{L}_\mathsf{O} A_\mathsf{O} \\ \text{Ordered} & A_\mathsf{O}, B_\mathsf{O} ::= p_\mathsf{O} \mid A_\mathsf{O} \oplus B_\mathsf{O} \mid A_\mathsf{O} \mathbin{\&} B_\mathsf{O} \mid \mathbf{1} \mid A_\mathsf{O} \setminus B_\mathsf{O} \mid B_\mathsf{O} \mathbin{/} A_\mathsf{O} \mid A_\mathsf{O} \bullet B_\mathsf{O} \mid A_\mathsf{O} \circ B_\mathsf{O} \mid \downarrow^\mathsf{L}_\mathsf{O} A_\mathsf{L} \end{array}$$

We read $\uparrow^\mathsf{L}_\mathsf{O} A_\mathsf{O}$ as *from ordered* A_O *up to linear* or just *up* A_O, and $\downarrow^\mathsf{L}_\mathsf{O} A_\mathsf{L}$ as *from linear* A_L *down to ordered* or just *down* A_L.

2 Counterexample: Swap

We refer to unary logical connectives that mediate between different modes of truth as *modal operators* or *modalities*. In traditional philosophical logic such modes could be *knowledge*, *belief*, *necessity*, or *possibility*, among many others that have been investigated. Here the mode determines whether we are considering ordered or linear logic.

We need some principled understanding of the shift modalities before we can write correct left and right rules to define their meaning. I hesitate to write down wrong rules, lest they burn themselves into your memory, but it is worth playing the *what if* game for a little to understand the restrictions we will eventually impose. Consider the judgment

$$\Delta_\mathsf{L} \mathbin{;} \Omega_\mathsf{O} \vdash C_m$$

where Δ_L collects all *linear* antecedents, Ω_O collects all *ordered* antecedents, and C_m is the succedent with the mode m either O or L. The first attempt at the rules, **which will be shown to be wrong**, would be to just strip the modality in all four cases. We take care to place each proposition in its appropriate antecedent zone.

$$\frac{\Delta \mathbin{;} \Omega \vdash A_\mathsf{O}}{\Delta \mathbin{;} \Omega \vdash \uparrow A_\mathsf{O}} \uparrow R?? \qquad \frac{\Delta \mathbin{;} \Omega_1 \; A_\mathsf{O} \; \Omega_2 \vdash C_m}{\Delta, \uparrow A_\mathsf{O} \mathbin{;} \Omega_1 \; \Omega_2 \vdash C_m} \uparrow L??$$

$$\frac{\Delta \mathbin{;} \Omega \vdash A_\mathsf{L}}{\Delta \mathbin{;} \Omega \vdash \downarrow A_\mathsf{L}} \downarrow R?? \qquad \frac{\Delta, A_\mathsf{L} \mathbin{;} \Omega_1 \; \Omega_2 \vdash C_m}{\Delta \mathbin{;} \Omega_1 \; (\downarrow A_\mathsf{L}) \; \Omega_2 \vdash C_m} \downarrow L??$$

Next comes the counterexample which shows that this combined logic does not make sense in the way we intended.

$$\dfrac{\dfrac{\dfrac{}{\cdot\ ;\ A_o \vdash A_o}\ \mathsf{id}}{\cdot\ ;\ A_o \vdash \uparrow A_o}\ \uparrow R??\qquad \dfrac{\dfrac{}{\cdot\ ;\ B_o \vdash B_o}\ \mathsf{id}\quad \dfrac{}{\cdot\ ;\ A_o \vdash A_o}\ \mathsf{id}}{\cdot\ ;\ B_o\ A_o \vdash B_o \bullet A_o}\ \bullet R}{\uparrow A_o\ ;\ B_o \vdash B_o \bullet A_o}\ \uparrow L??}{\cdot\ ;\ A_o\ B_o \vdash B_o \bullet A_o}\ \mathsf{cut}??}{\cdot\ ;\ A_o \bullet B_o \vdash B_o \bullet A_o}\ \bullet L$$

Why is this troublesome? The conclusion proves that fuse is commutative, which means that the ordered connectives no longer have the meaning we expect. Moreover, something must fail in cut elimination: our conclusion contains only ordered propositions, and in ordered logic it is certainly *not* the case that fuse is commutative. There can be no cut-free proof of the endsequent in the combined logic, so cut elimination must fail.

The issue is that $\uparrow A_o$ is a *linear* proposition, which is therefore "mobile" in the sense that it can move around arbitrarily among the antecedents. If we could indeed prove $\cdot\ ;\ A_o \vdash \uparrow A_o$ this means we could make any ordered proposition mobile, place it wherever we like (the use of $\uparrow L??$ in the counterexample) when we make it ordered again. This "trick" was exploited to illegally move A_o from the left of B_o to its right.

Our analysis is that the proof of *linear* proposition $\uparrow A_o$ should not be allowed to depend on the *ordered* proposition A_o, because the linear succedent can be used in ways not justified by the ordered antecedent. This would rule out the very judgment $\cdot\ ;\ A_o \vdash \uparrow A_o$ so the cut rule cannot be correct, nor can the application of $\uparrow R??$. On the other hand, $\uparrow L??$ looks defensible.

We can also take a look at what the result of a hypothetical cut reduction would be. It is a principal case, since the cut formula $\uparrow A_o$ is the principal formula of both inferences. We obtain:

$$\dfrac{\dfrac{\dfrac{}{\cdot\ ;\ A_o \vdash A_o}\ \mathsf{id}\qquad \dfrac{\dfrac{}{\cdot\ ;\ B_o \vdash B_o}\ \mathsf{id}\quad \dfrac{}{\cdot\ ;\ A_o \vdash A_o}\ \mathsf{id}}{\cdot\ ;\ B_o\ A_o \vdash B_o \bullet A_o}\ \bullet R}{\cdot\ ;\ A_o\ B_o \vdash B_o \bullet A_o}\ \mathsf{cut}??}{\cdot\ ;\ A_o \bullet B_o \vdash B_o \bullet A_o}\ \bullet L$$

We see that the conclusion of the cut would have antecedents $B_o\ A_o$ when before the cut reduction it was $A_o\ B_o$. Thus, the cut reduction fails.

3 The Declaration of Independence

The considerations in the previous section lead us to the following *principle of independence*.

> **Independence Principle:** The proof of a linear succedent cannot depend on ordered antecedents.

The general form of this principle in later lectures will be: *The proof of a succedent with mode m can only depend on antecedents with modes equal to or stronger than m.* Here we consider a mode stronger if it supports more structural properties such as exchange, weakening, or contraction.

This means we really have only the following two judgment forms.

$$\Delta_l\ ;\ \Omega_o \vdash A_o$$
$$\Delta_l \vdash A_l$$

Ordered succedents A_o can depend on linear and ordered antecedents, while linear succedents A_l can only depend on linear antecedents. We will later unify these with others into a single judgment form with some presuppositions to ensure they are meaningful.

Now we can go through our rules and subject them to the independence principle to arrive at the following collection of right and left rules.

$$\frac{\Delta\ ;\ \cdot \vdash A_o}{\Delta \vdash \uparrow A_o}\ \uparrow R \qquad \frac{\Delta\ ;\ \Omega_1\ A_o\ \Omega_2 \vdash C_o}{\Delta, \uparrow A_o\ ;\ \Omega_1\ \Omega_2 \vdash C_o}\ \uparrow L$$

$$\frac{\Delta \vdash A_l}{\Delta\ ;\ \cdot \vdash \downarrow A_l}\ \downarrow R \qquad \frac{\Delta, A_l\ ;\ \Omega_1\ \Omega_2 \vdash C_o}{\Delta\ ;\ \Omega_1\ (\downarrow A_l)\ \Omega_2 \vdash C_o}\ \downarrow L$$

Please make sure you go through these and understand how they arise from the previous dubious rules simply by heeding the independence principle. As an example, with consider $\downarrow R$. We had proposed

$$\frac{\Delta\ ;\ \Omega \vdash A_l}{\Delta\ ;\ \Omega \vdash \downarrow A_l}\ \downarrow R??$$

In the premise, there cannot be any ordered antecedents, leading to a better approximation:

$$\frac{\Delta \vdash A_{\mathsf{L}}}{\Delta \,;\, \Omega \vdash \downarrow A_{\mathsf{L}}}\ \downarrow R??$$

But every antecedent in Ω must be used—we cannot simply drop them at this rule which would be tantamount to *weakening*. So we must require there to be no ordered antecedents and we arrive at the correct rule.

$$\frac{\Delta \vdash A_{\mathsf{L}}}{\Delta \,;\, \cdot \vdash \downarrow A_{\mathsf{L}}}\ \downarrow R$$

4 Cut and Identity

The rules for cut and identity can now be derived from the same considerations of independence. Identity is particularly straightforward.

$$\frac{}{\cdot \,;\, A_{\mathsf{O}} \vdash A_{\mathsf{O}}}\ \mathsf{id}_{\mathsf{O}} \qquad \frac{}{A_{\mathsf{L}} \vdash A_{\mathsf{L}}}\ \mathsf{id}_{\mathsf{L}}$$

For cut, it turns out there are three rules because independence rules out one of the four combinations of the judgment forms.

$$\frac{\Delta' \,;\, \Omega' \vdash A_{\mathsf{O}} \quad \Delta \,;\, \Omega_1\ A_{\mathsf{O}}\ \Omega_2 \vdash C_{\mathsf{O}}}{\Delta, \Delta' \,;\, \Omega_1\ \Omega'\ \Omega_2 \vdash C_{\mathsf{O}}}\ \mathsf{cut}_{\mathsf{OO}}$$

$$\frac{\Delta' \vdash A_{\mathsf{L}} \quad \Delta, A_{\mathsf{L}} \,;\, \Omega \vdash C_{\mathsf{O}}}{\Delta, \Delta' \,;\, \Omega \vdash C_{\mathsf{O}}}\ \mathsf{cut}_{\mathsf{LO}} \qquad \frac{\Delta' \vdash A_{\mathsf{L}} \quad \Delta, A_{\mathsf{L}} \vdash C_{\mathsf{L}}}{\Delta, \Delta' \vdash C_{\mathsf{L}}}\ \mathsf{cut}_{\mathsf{LL}}$$

5 Other Propositional Rules

The other propositional rules are carried over or straightforwardly adapted from ordered or linear logic. We show only two examples: the rules for $A_{\mathsf{O}} \setminus B_{\mathsf{O}}$ and $A_{\mathsf{L}} \multimap B_{\mathsf{L}}$. The $\multimap L$ rules split into two, for similar reasons why there are two version of the cut rule for linear propositions.

$$\frac{\Delta \,;\, A_{\mathsf{O}}\ \Omega \vdash B_{\mathsf{O}}}{\Delta \,;\, \Omega \vdash A_{\mathsf{O}} \setminus B_{\mathsf{O}}}\ \setminus R \qquad \frac{\Delta' \,;\, \Omega' \vdash A_{\mathsf{O}} \quad \Delta \,;\, \Omega_1\ B_{\mathsf{O}}\ \Omega_2 \vdash C_{\mathsf{O}}}{\Delta, \Delta' \,;\, \Omega_1\ \Omega'\ (A_{\mathsf{O}} \setminus B_{\mathsf{O}})\ \Omega_2 \vdash C_{\mathsf{O}}}\ \setminus L$$

$$\frac{\Delta, A_{\mathsf{L}} \vdash B_{\mathsf{L}}}{\Delta \vdash A_{\mathsf{L}} \multimap B_{\mathsf{L}}}\ \multimap R \qquad \frac{\Delta' \vdash A_{\mathsf{L}} \quad \Delta, B_{\mathsf{L}} \vdash C_{\mathsf{L}}}{\Delta, \Delta', A_{\mathsf{L}} \multimap B_{\mathsf{L}} \vdash C_{\mathsf{L}}}\ \multimap L_{\mathsf{L}} \qquad \frac{\Delta' \vdash A_{\mathsf{L}} \quad \Delta, B_{\mathsf{L}} \,;\, \Omega \vdash C_{\mathsf{O}}}{\Delta, \Delta', A_{\mathsf{L}} \multimap B_{\mathsf{L}} \,;\, \Omega \vdash C_{\mathsf{O}}}\ \multimap L_{\mathsf{O}}$$

6 Roundtrips

Including shifts in the language of propositions allows us to consider *round trips* from ordered to linear logic and back, and from ordered to linear and back. How does $\downarrow\uparrow A_o$ compare to A_o? Conversely, how does $\uparrow\downarrow A_l$ compare to A_l? We just try construct a cut-free derivation, bottom up, for each of these proposition to see which hold and which do not.

$$\text{no rule applicable} \qquad \dfrac{\dfrac{\dfrac{}{\cdot\ ;\ A_o \vdash A_o}\ \mathsf{id}_o}{\uparrow A_o\ ;\ \cdot \vdash A_o}\ \uparrow R}{\cdot\ ;\ \downarrow\uparrow A_o \vdash A_o}\ \downarrow L$$

$$\cdot\ ;\ A_o \vdash \downarrow\uparrow A_o$$

$$\dfrac{\dfrac{\dfrac{}{A_l \vdash A_l}\ \mathsf{id}_l}{A_l\ ;\ \cdot \vdash \downarrow A_l}\ \downarrow R}{A_l \vdash \uparrow\downarrow A_l}\ \uparrow R \qquad \text{no rule applicable}$$

$$\uparrow\downarrow A_l \vdash A_l$$

In two sequents we get stuck immediately due to the restrictions imposed by the independence principle.

So if we know A_l we can conclude $\uparrow\downarrow A_l$ via a roundtrip to an ordered proposition but not the other way around. Conversely, if we are trying to prove A_o it is sufficient to prove $\downarrow\uparrow A_o$.

The observations

$$A_o \not\vdash \downarrow\uparrow A_o$$
$$\downarrow\uparrow A_o \vdash A_o$$

$$A_l \vdash \downarrow\uparrow A_l$$
$$\downarrow\uparrow A_l \not\vdash A_l$$

suggest that $\downarrow\uparrow A_o \simeq \Box A_o$ behaves like the modality of *necessity*, which can be characterized categorically as a *comonad*, while $\uparrow\downarrow A_o \simeq \Diamond A_l$ behaves like a form of *possiblity*, which can be characterized categorically as a *strong monad*.

We pursue this conjecture a little further to see if the shifts distribute over implications. They do. While we distribute the shift, linear implications will turn into over/under and vice versa. We prefer $A \setminus B$ here since the order of its arguments is consistent with linear impleication, but this

choice is arbitrary. We start

$$\dfrac{\dfrac{}{\cdot\,;\,(\downarrow A)\,(\downarrow(A \multimap B)) \vdash \downarrow B}\,?}{\cdot\,;\,\cdot \vdash \downarrow(A \multimap B) \setminus \downarrow A \setminus \downarrow B}\,\setminus R \times 2$$

We pause here to note that the $\downarrow R$ rule is not applicable since the ordered context is not empty. So we apply the $\downarrow L$ rules twice, to empty it out. The remainder is straightforward.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A_{\mathsf{L}} \vdash A_{\mathsf{L}}}\,\mathsf{id} \quad \dfrac{}{B_{\mathsf{L}} \vdash B_{\mathsf{L}}}\,\mathsf{id}}{A_{\mathsf{L}}, A_{\mathsf{L}} \multimap B_{\mathsf{L}} \vdash B_{\mathsf{L}}}\,\multimap L_{\mathsf{L}}}{A_{\mathsf{L}}, A_{\mathsf{L}} \multimap B_{\mathsf{L}}\,;\,\cdot \vdash \downarrow B_{\mathsf{L}}}\,\downarrow R}{\cdot\,;\,(\downarrow A_{\mathsf{L}})\,(\downarrow(A_{\mathsf{L}} \multimap B_{\mathsf{L}})) \vdash \downarrow B_{\mathsf{L}}}\,\downarrow L \times 2}{\cdot\,;\,\cdot \vdash \downarrow(A_{\mathsf{L}} \multimap B_{\mathsf{L}}) \setminus \downarrow A_{\mathsf{L}} \setminus \downarrow B_{\mathsf{L}}}\,\setminus R \times 2$$

Similarly, $\uparrow$ distributes over ordered implication $A \setminus B$. We pause after two steps.

$$\dfrac{\dfrac{}{\uparrow(A_{\mathsf{o}} \setminus B_{\mathsf{o}}), \uparrow A_{\mathsf{o}} \vdash \uparrow B_{\mathsf{o}}}\,?}{\cdot \vdash \uparrow(A_{\mathsf{o}} \setminus B_{\mathsf{o}}) \multimap \uparrow A_{\mathsf{o}} \multimap \uparrow B_{\mathsf{o}}}\,\multimap R \times 2$$

Now we need to apply $\uparrow R$ so we can then apply $\uparrow L$ twice to place the two antecedents in the ordered zone in the correct order.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A_{\mathsf{o}} \vdash A_{\mathsf{o}}}\,\mathsf{id}_{\mathsf{o}} \quad \dfrac{}{B_{\mathsf{o}} \vdash B_{\mathsf{o}}}\,\mathsf{id}_{\mathsf{o}}}{\cdot\,;\,A_{\mathsf{o}}\,(A_{\mathsf{o}} \setminus B_{\mathsf{o}}) \vdash B_{\mathsf{o}}}\,\setminus L}{\uparrow(A_{\mathsf{o}} \setminus B_{\mathsf{o}}), \uparrow A_{\mathsf{o}}\,;\,\cdot \vdash B_{\mathsf{o}}}\,\uparrow L \times 2}{\uparrow(A_{\mathsf{o}} \setminus B_{\mathsf{o}}), \uparrow A_{\mathsf{o}} \vdash \uparrow B_{\mathsf{o}}}\,\uparrow R}{\cdot \vdash \uparrow(A_{\mathsf{o}} \setminus B_{\mathsf{o}}) \multimap \uparrow A_{\mathsf{o}} \multimap \uparrow B_{\mathsf{o}}}\,\multimap R \times 2$$

The observations

$$\begin{array}{l} \cdot\,;\,\cdot \vdash \downarrow(A_{\mathsf{L}} \multimap B_{\mathsf{L}}) \setminus \downarrow A_{\mathsf{L}} \setminus \downarrow B_{\mathsf{L}} \\ \cdot \vdash \uparrow(A_{\mathsf{o}} \setminus B_{\mathsf{o}}) \multimap \uparrow A_{\mathsf{o}} \multimap \uparrow B_{\mathsf{o}} \end{array}$$

provide further evidence for our conjectures of the modal nature of $\downarrow\uparrow$ and $\uparrow\downarrow$ and necessity and strong possibility (see Exercises 1 and 2).

7 Operational Interpretation of Shifts

When we assign processes as proof terms we mark channels ordered or linear, written as x_{o} or x_{l}. This doesn't affect the identity of a channel, only how it is used.

$$(x_{1\mathsf{l}}{:}A_{1\mathsf{l}}, \ldots, x_{n\mathsf{l}}{:}A_{n\mathsf{l}}) \; ; \; (y_{1\mathsf{o}}{:}B_{1\mathsf{o}} \; \ldots \; y_{k\mathsf{o}}{:}B_{k\mathsf{o}}) \vdash P :: (z_{\mathsf{o}} : C_{\mathsf{o}})$$
$$(x_{1\mathsf{l}}{:}A_{1\mathsf{l}}, \ldots, x_{n\mathsf{l}}{:}A_{n\mathsf{l}}) \vdash P :: (z_{\mathsf{l}} : C_{\mathsf{l}})$$

As we remarked several times, the operational interpretation of ordered and their corresponding linear connectives is exactly the same. Order only imposes a restriction on the use of ordered channels. Therefore we could assume that the shifts have no essential *operational* purpose. Indeed, looking at the rules it seems that a message of shift corresponds to communicating a mutual agreement that a channel changes status, from ordered to linear or vice versa. We have to determine which of the rules carries information and which does not.

$$\frac{\Delta \; ; \cdot \vdash A_{\mathsf{o}}}{\Delta \vdash \uparrow A_{\mathsf{o}}} \uparrow R \qquad \frac{\Delta \; ; \; \Omega_1 \; A_{\mathsf{o}} \; \Omega_2 \vdash C_{\mathsf{o}}}{\Delta, \uparrow A_{\mathsf{o}} \; ; \; \Omega_1 \; \Omega_2 \vdash C_{\mathsf{o}}} \uparrow L$$

We see that $\uparrow R$ can always be applied and thus carries no information, while an antecedent $\uparrow A_{\mathsf{o}}$ may have to wait until the succedent is ordered. It also implicitly carries the information on *where* in the context to insert A_{o}. This means $\uparrow R$ will receive and $\uparrow L$ will send.

$$\frac{\Delta \; ; \cdot \vdash P :: (x_{\mathsf{o}} : A_{\mathsf{o}})}{\Delta \vdash \mathsf{shift} \leftarrow \mathsf{recv} \; x_{\mathsf{l}} \; ; \; P :: (x_{\mathsf{l}} : \uparrow A_{\mathsf{o}})} \uparrow R$$

$$\frac{\Delta \; ; \; \Omega_1 \; (x_{\mathsf{o}}{:}A_{\mathsf{o}}) \; \Omega_2 \vdash Q :: (z_{\mathsf{o}} : C_{\mathsf{o}})}{\Delta, x_{\mathsf{l}}{:}\uparrow A_{\mathsf{o}} \; ; \; \Omega_1 \; \Omega_2 \vdash \mathsf{send} \; x_{\mathsf{l}} \; \mathsf{shift} \; ; \; Q :: (z_{\mathsf{o}} : C_{\mathsf{o}})} \uparrow L$$

For the down shift modality, the roles are reversed. $\downarrow R$ does not always apply since there may be ordered antecedents. On the other hand, $\downarrow L$ always applies (read bottom-up, of course) and therefore it carries no information and will receive.

$$\frac{\Delta \vdash P :: (x_{\mathsf{l}} : A_{\mathsf{l}})}{\Delta \; ; \cdot \vdash \mathsf{send} \; x_{\mathsf{o}} \; \mathsf{shift} \; ; \; P :: (x_{\mathsf{o}} : \downarrow A_{\mathsf{l}})} \downarrow R$$

$$\frac{\Delta, x_{\mathsf{l}}{:}A_{\mathsf{l}} \; ; \; \Omega_1 \; \Omega_2 \vdash Q :: (z_{\mathsf{o}} : C_{\mathsf{o}})}{\Delta \; ; \; \Omega_1 \; (x_{\mathsf{o}}{:}\downarrow A_{\mathsf{l}}) \; \Omega_2 \vdash \mathsf{shift} \leftarrow \mathsf{recv} \; x_{\mathsf{o}} \; ; \; Q :: (z_{\mathsf{o}} : C_{\mathsf{o}})} \downarrow L$$

As with internal/external choice and ordered implication/fuse, the up and down shifts form dual pairs and therefore use the same program construct. We can always tell from the situation which of the two is meant.

The operational rules are straightforward.

$$\frac{\mathsf{proc}(x_{\mathsf{L}}, \mathsf{shift} \leftarrow \mathsf{recv}\ x_{\mathsf{L}}\ ;\ P) \quad \mathsf{proc}(z_{\mathsf{O}}, \mathsf{send}\ x_{\mathsf{L}}\ \mathsf{shift}\ ;\ Q)}{\mathsf{proc}(x_{\mathsf{O}}, P) \quad \mathsf{proc}(z_{\mathsf{O}}, Q)}\ \uparrow C$$

$$\frac{\mathsf{proc}(x_{\mathsf{O}}, \mathsf{send}\ x_{\mathsf{O}}\ \mathsf{shift}\ ;\ P) \quad \mathsf{proc}(z_{\mathsf{O}}, \mathsf{shift} \leftarrow \mathsf{recv}\ x_{\mathsf{O}}\ ;\ Q)}{\mathsf{proc}(x_{\mathsf{L}}, P) \quad \mathsf{proc}(z_{\mathsf{O}}, Q)}\ \downarrow C$$

8 Example: A Linear Client Using an Ordered Service

By the independence principle, a process that provides along a linear channel cannot use an ordered channel. But if we have an ordered process that uses linear channels internally, we can use an ordered storage server (say, a stack) to store the linear channel $x_{\mathsf{L}}{:}A_{\mathsf{L}}$. In order to allow this we have to coerce the linear channel to be ordered, of type $\downarrow A_{\mathsf{L}}$, so we define

$x_{\mathsf{L}}{:}A_{\mathsf{L}}\ ;\ \cdot \vdash \mathit{down} :: (y_{\mathsf{O}} : \downarrow A_{\mathsf{L}})$
$y_{\mathsf{O}} \leftarrow \mathit{down} \leftarrow x_{\mathsf{L}} =$
  $\mathsf{send}\ y_{\mathsf{O}}\ \mathsf{shift}\ ;$
  $y_{\mathsf{L}} \leftarrow x_{\mathsf{L}}$

The we assume we have ordered type

$$\begin{aligned}\mathsf{stack}_A = \&\{&\mathsf{ins} : A \setminus \mathsf{stack}_A \\ &\mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : A \bullet \mathsf{stack}\}\}\end{aligned}$$

with the usual ordered implementation. Now assume we are in a situation where we own a channel of type A_{L} and also a stack with elements of type $\downarrow A_{\mathsf{L}}$

$$(x_{\mathsf{L}}{:}A_{\mathsf{L}})\ ;\ (s{:}\mathsf{stack}_{\downarrow A_{\mathsf{L}}}) \vdash (z_{\mathsf{O}} : C_{\mathsf{O}})$$

In order to push x_{L} onto the stack we first lower it to be ordered. Upon retrieval, we do the opposite. In the types we omit the succedent since it plays no role in this code.

$$\begin{array}{lll} & \% & (x_{\mathsf{L}}{:}A_{\mathsf{L}})\ ;\ (s{:}\mathsf{stack}_{\downarrow A_{\mathsf{L}}}) \vdash \ldots \\ w_{\mathsf{O}} \leftarrow \mathit{down} \leftarrow x_{\mathsf{L}}\ ; & \% & \cdot\ ;\ (w_{\mathsf{O}}{:}\downarrow A_{\mathsf{L}})\ (s{:}\mathsf{stack}_{\downarrow A_{\mathsf{L}}}) \vdash \ldots \\ s.\mathsf{ins}\ ;\ \mathsf{send}\ s\ w_0\ ; & \% & \cdot\ ;\ (s{:}\mathsf{stack}_{\downarrow A_{\mathsf{L}}}) \vdash \ldots \\ \ldots \text{ other operations } \ldots & & \end{array}$$

s.del ; % $\cdot\,;\, s{:}\oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : {\downarrow}A_{\mathsf{L}} \bullet \mathsf{stack}_{{\downarrow}A_{\mathsf{L}}}\} \vdash \ldots$
case s (none $\Rightarrow$ wait s ; ...
| some $\Rightarrow$ $u_{\mathsf{O}} \leftarrow$ recv s ; % $\cdot\,;\, (u_{\mathsf{O}}{:}{\downarrow}A_{\mathsf{L}})\ (s{:}\mathsf{stack}_{{\downarrow}A_{\mathsf{L}}}) \vdash \ldots$
shift $\leftarrow$ recv u_{O} ; % $(u_{\mathsf{L}}{:}A_{\mathsf{L}})\ ;\ (s{:}\mathsf{stack}_{{\downarrow}A_{\mathsf{L}}}) \vdash \ldots$
...)

9 Identity Expansion

To verify harmony we should check cut reduction and identity expansion. We begin with the simpler identity expansion.

$$\dfrac{\dfrac{\dfrac{}{\cdot\,;\, A_{\mathsf{O}} \vdash A_{\mathsf{O}}}\,\mathsf{id}_{\mathsf{O}}}{{\uparrow}A_{\mathsf{O}}\,;\,\cdot \vdash A_{\mathsf{O}}}\,{\uparrow}L}{{\uparrow}A_{\mathsf{O}} \vdash {\uparrow}A_{\mathsf{O}}}\,{\uparrow}R \qquad \dfrac{\dfrac{\dfrac{}{A_{\mathsf{L}} \vdash A_{\mathsf{L}}}\,\mathsf{id}_{\mathsf{L}}}{A_{\mathsf{L}}\,;\,\cdot \vdash {\downarrow}A_{\mathsf{L}}}\,{\downarrow}R}{\cdot\,;\,{\downarrow}A_{\mathsf{L}} \vdash {\downarrow}A_{\mathsf{L}}}\,{\downarrow}L$$

10 Cut Reduction

Cut reduction usually has the key insight in the proof of admissibility of cut and therefore ultimately cut elimination. We only show one case of cut reduction (for ${\uparrow}A_{\mathsf{O}}$) and elide other cases as well as admissibility of cut (see also Exercise 3).

Assume the first premise of a cut ends in ${\uparrow}R$ rule.

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}'\\ \Delta'\,;\,\cdot \vdash A_{\mathsf{O}}\end{array}}{\Delta' \vdash {\uparrow}A_{\mathsf{O}}}\,{\uparrow}R \qquad \begin{array}{c}\mathcal{E}\\ \Delta, {\uparrow}A_{\mathsf{O}}\,;\,\Omega \vdash C_{\mathsf{O}}\end{array}}{\Delta, \Delta'\,;\,\Omega \vdash C_{\mathsf{O}}}\,\mathsf{cut}_{\mathsf{LO}} \qquad \dfrac{\dfrac{\begin{array}{c}\mathcal{D}'\\ \Delta'\,;\,\cdot \vdash A_{\mathsf{O}}\end{array}}{\Delta' \vdash {\uparrow}A_{\mathsf{O}}}\,{\uparrow}R \qquad \begin{array}{c}\mathcal{E}\\ \Delta, {\uparrow}A_{\mathsf{O}} \vdash C_{\mathsf{L}}\end{array}}{\Delta, \Delta' \vdash C_{\mathsf{L}}}\,\mathsf{cut}_{\mathsf{LL}}$$

The ${\uparrow}L$ rule has the form

$$\dfrac{\Delta\,;\,\Omega_1\ A_{\mathsf{O}}\ \Omega_2 \vdash C_{\mathsf{O}}}{\Delta, {\uparrow}A_{\mathsf{O}}\,;\,\Omega_1\ \Omega_2 \vdash C_{\mathsf{O}}}\,{\uparrow}L$$

so only the first of these two cases will result in a cut reduction. The second case above will lead to some commuting conversions, if A_{O} is a side formula

of inferences in $\mathcal{E}$. The only case for reduction then has the form

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}'\\ \Delta' \,;\, \cdot \vdash A_{\mathsf{o}}\end{array}}{\Delta' \vdash \uparrow A_{\mathsf{o}}}\uparrow R \quad \dfrac{\begin{array}{c}\mathcal{E}'\\ \Delta \,;\, \Omega_1\ A_{\mathsf{o}}\ \Omega_2 \vdash C_{\mathsf{o}}\end{array}}{\Delta, \uparrow A_{\mathsf{o}} \,;\, \Omega_1\ \Omega_2 \vdash C_{\mathsf{o}}}\uparrow L}{\Delta, \Delta' \,;\, \Omega_1\ \Omega_2 \vdash C_{\mathsf{o}}}\ \mathsf{cut}_{\mathsf{LO}}$$

which reduces to

$$\dfrac{\begin{array}{c}\mathcal{D}'\\ \Delta' \,;\, \cdot \vdash A_{\mathsf{o}}\end{array} \quad \begin{array}{c}\mathcal{E}'\\ \Delta \,;\, \Omega_1\ A_{\mathsf{o}}\ \Omega_2 \vdash C_{\mathsf{o}}\end{array}}{\Delta, \Delta' \,;\, \Omega_1\ \Omega_2 \vdash C_{\mathsf{o}}}\ \mathsf{cut}_{\mathsf{oo}}$$

Exercises

Exercise 1 Define $\Box A_{\mathsf{o}} = \downarrow\uparrow A_{\mathsf{o}}$. Taking the small liberty of omitting the linear zone when it is empty, we have already seen

1. $\Box A_{\mathsf{o}} \vdash A_{\mathsf{o}}$
2. $A_{\mathsf{o}} \nvdash \Box A_{\mathsf{o}}$

Prove or refute each of the following:

1. $\Box A_{\mathsf{o}} \vdash \Box\Box A_{\mathsf{o}}$
2. $\Box(A_{\mathsf{o}} \setminus B_{\mathsf{o}})\ (\Box A_{\mathsf{o}}) \vdash \Box B_{\mathsf{o}}$

Exercise 2 Define $\Diamond A_{\mathsf{L}} = \uparrow\downarrow A_{\mathsf{L}}$. We have already seen

1. $A_{\mathsf{L}} \vdash \Diamond A_{\mathsf{L}}$
2. $\Diamond A_{\mathsf{L}} \nvdash A_{\mathsf{L}}$

Proof or refute each of the following

1. $\Diamond\Diamond A_{\mathsf{L}} \vdash \Diamond A_{\mathsf{L}}$
2. $\Diamond(A_{\mathsf{L}} \multimap B_{\mathsf{L}}), \Diamond A_{\mathsf{L}} \vdash \Diamond B_{\mathsf{L}}$

Exercise 3 Show all cases of cut reduction for $\downarrow A_{\mathsf{L}}$.

Exercise 4 Recall

$$x_{\mathsf{L}}{:}A_{\mathsf{L}} \;;\; \cdot \vdash \mathit{down} :: (y_{\mathsf{O}} : {\downarrow}A_{\mathsf{L}})$$

from Section 8. A priori, there are four interesting possibilities for such coercions. For each, show that it cannot exist or write a process that implements it.

References

[Ben94] 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 (CLS'94)*, Kazimierz, Poland, September 1994. Springer LNCS 933. An extended version appears as Technical Report UCAM-CL-TR-352, University of Cambridge.

[DP01] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, May 2001.

[Mog89] Eugenio Moggi. Computational lambda calculus and monads. In *Proceedings of the Fourth Symposium on Logic in Computer Science*, pages 14–23, Asilomar, California, June 1989. IEEE Computer Society Press.

[PD01] 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.

[PG15] 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.

[Ree09] Jason Reed. A judgmental deconstruction of modal logic. Unpublished manuscript, 2009.

Lecture Notes on Asynchronous Communication

15-816: Substructural Logics
Frank Pfenning

Lecture 13
October 11, 2016

In this lecture we will first update earlier observations regarding asynchronous communication and then develop an alternative logical integration of synchronous and asynchronous communication. The material is adapted from [DCPT12] and [PG15].

1 Asynchronous from Synchronous Communication

By *asynchronous communication* we mean this in the sense of the π-calculus: we send messages along channels and continue execution without waiting for the message to be received. We do expect, however, that the messages arrive in the order they were sent. When modeling asynchronous communication we need to ensure these two properties, which is usually achieved with a *message buffer* for each channel.

Of course, we want to accomplish all of this by logical means, within the scope of the interpretation of propositions as types, proofs as programs, and computation as cut reduction.

As an example, we consider communication of labels to the right, which is the operational interpretation of internal choice ($\oplus$). Operationally, the key observation was:

$$\mathsf{R}.l_k \; ; \; P \quad \simeq \quad P \mid (\mathsf{R}.l_k \; ; \leftrightarrow)$$

Instead of *synchronously* sending label l_k to the right and then continue with P when it is received, we immediately spawn P and also a (tiny) process that only carries l_k and terminates when that is received.

The fact that the second form is well-typed when the first one is means we can already express asynchronous communication. We just need to decide which of the two programs above to write.

From a logical perspective, the asynchronous form below on the right reduces to the synchronous form on the left by a permuting reduction followed by an identity reduction.

$$\frac{\omega \vdash P : A_k \quad (k \in I)}{\omega \vdash \mathsf{R}.l_k \; ; P : \oplus\{l_i : A_i\}_{i \in I})} \oplus R_k \qquad \frac{\omega \vdash P : A_k \quad \dfrac{\dfrac{}{A_k \vdash \leftrightarrow : A_k}\,\mathsf{id} \quad (k \in I)}{A_k \vdash \mathsf{R}.l_k \; ; \leftrightarrow : \oplus\{l_i : A_i\}_{i \in I}} \oplus R_k}{\omega \vdash P \mid (\mathsf{R}.l_k \; ; \leftrightarrow) : \oplus\{l_i : A_i\}_{i \in I}}\,\mathsf{cut}$$

Does this generalize to the setting of ordered and linear logic? Yes, as we will illustrate using an ordered example, but it also works for the linear connectives. This time, we start with the logical rules and deduce what the right asynchronous form must be in analogy with the subsingleton case. We use $\bullet R^*$ as our illustrative example.

$$\frac{\Omega \vdash P :: (x : B)}{(w{:}A)\; \Omega \vdash \mathsf{send}\; x\; w \; ; P :: (x : A \bullet B)} \bullet R^*$$

We want to "unshackle" P so it can execute independently from sending of w along x. This is accomplished by a cut. Filling in what we know, we have the following partial deduction:

$$\frac{\Omega \vdash P :: (x : B) \quad (w{:}A)\; (x{:}B) \vdash \ldots :: (x : A \bullet B)}{(w{:}A)\; \Omega \vdash \ldots :: (x : A \bullet B)}\,\mathsf{cut}$$

We see that there is a name clash, so we rename the x introduced by the cut to x'.

$$\frac{\Omega \vdash P :: (x' : B) \quad (w{:}A)\; (x'{:}B) \vdash \ldots :: (x : A \bullet B)}{(w{:}A)\; \Omega \vdash \ldots :: (x : A \bullet B)}\,\mathsf{cut}$$

Luckily, we can directly prove $A\; B \vdash A \bullet B$, even using our weakened rule

$\bullet R^*$. Filling that in, we obtain

$$\frac{\Omega \vdash [x'/x]P :: (x' : B) \quad \dfrac{\dfrac{}{x'{:}B \vdash x \leftarrow x' :: (x : B)}\,\mathsf{id}}{(w{:}A)\ (x'{:}B) \vdash \mathsf{send}\ x\ w\ ;\ x \leftarrow x' :: (x : A \bullet B)}\,\bullet R^*}{(w{:}A)\ \Omega \vdash (x' \leftarrow [x'/x]P\ ;\ \mathsf{send}\ x\ w\ ;\ x \leftarrow x') :: (x : A \bullet B)}\,\mathsf{cut}$$

Operationally, this corresponds to the transition

$$\frac{\mathsf{proc}(x, x' \leftarrow [x'/x]P\ ;\ \mathsf{send}\ x\ w\ ;\ x \leftarrow x')}{\mathsf{proc}(x', [x'/x]P) \quad \mathsf{proc}(\mathsf{send}\ x\ w\ ;\ x \leftarrow x')}\,\mathsf{cmp}^{x'}$$

where x' must be fresh, which represent the derived asynchronous behavior under the correspondence

$$\mathsf{send}\ x\ w\ ;\ P \quad \simeq \quad x' \leftarrow [x'/x]P\ ;\ \mathsf{send}\ x\ w\ ;\ x \leftarrow x'$$

Similar correspondences hold for all other sending constructs (see Exercise 2).

2 An Asynchronous Semantics

As the previous section illustrates, we can implement asynchronous communication given a language with a synchronous semantics. There are two drawbacks of this solution. One is purely a matter of experience: the asynchronous form is tedious to write, since it involves an additional cut and identity. The other is that in realistic settings, synchronous communication is too complex as primitive, while asynchronous communication provides a much more natural model. Synchronous communication is then usually achieved by a protocol of specific asynchronous exchanges.

If we want communication to be a priori asynchronous, all we have to do is to employ the insights from the previous section to devise asynchronous computation rules for the usual sending constructs. In other words, the construct on the left should behave like the construct to the construct to the right.

$$\mathsf{send}\ x\ w\ ;\ P \quad \simeq \quad x' \leftarrow [x'/x]P\ ;\ \mathsf{send}\ x\ w\ ;\ x \leftarrow x'$$

We accomplish this through a new predicate msg containing only certain specialized forms. The sending rule for $\bullet R^*$ becomes

$$\frac{\mathsf{proc}(x, \mathsf{send}\ x\ w\ ;\ P)}{\mathsf{proc}(x', [x'/x]P) \quad \mathsf{msg}(x, \mathsf{send}\ x\ w\ ;\ x \leftarrow x')}\,\bullet C\_\mathsf{send}^{x'}$$

When typing the configuration we type the message just as we would a process. The only reason to single it out with a separate predicate is to prevent immediate application of the same rule again, and again, etc. Messages are received in a way that is consistent with their interpretation as a (tiny) process, shown below for purpose of illustration.

$$\dfrac{\dfrac{\mathsf{proc}(x, \mathsf{send}\ x\ w\ ;\ x \leftarrow x') \quad \mathsf{proc}(z, y \leftarrow \mathsf{recv}\ x\ ;\ Q_y)}{\mathsf{proc}(x, x \leftarrow x') \quad \mathsf{proc}(z, Q_w)}\ \mathsf{fwd}}{\mathsf{proc}(z, [x'/x]Q_w)}\ \bullet C$$

We previously wrote the semantics of forwarding as a *global* substitution of x' for x, but here we know the client for x is the process Q_w offering along z. By linearity of channels in well-typed configurations, substitution Q_w is all that is required.[1] In fact, there is an alternative semantics for forwarding that takes this into account (see Exercise 3).

We summarize the above two steps from the synchronous semantics, using only processes, as the one-step transition in the asynchronous semantics using messages.

$$\dfrac{\mathsf{msg}(x, \mathsf{send}\ x\ w\ ;\ x \leftarrow x') \quad \mathsf{proc}(z, y \leftarrow \mathsf{recv}\ x\ ;\ Q_y)}{\mathsf{proc}(z, [x'/x]Q_w)}\ \bullet C\_\mathsf{recv}$$

3 Polarizing Propositions

In the previous section we have developed a semantics were all communication is asynchronous. Is this expressive enough to model synchronous communication if that is what we want in some places? Speaking purely operationally, the standard solution is to send the intended message and then wait for an acknowledgment of receipt before continuing. We *could* introduce this kind of operational solution here in an ad hoc way, but it would be much more satisfactory if it had a good logical justification.

And, yes, there is such a justification or I probably wouldn't have brought it up.

Surprisingly, we only need to generalize slightly the mechanism of modes and shifts in order to find a logical foundation for synchronous communication in an asynchronous language.

[1] Note, however, if we do not perform the two steps together then process z could send x do some other process and the second step would be wrong.

The first key idea is to classify propositions based on whether they send or receive when viewed from the provider perspective. This is the same as asking which right rules for the connectives are invertible in the sense that they can always be applied during the search for a proof of $\Omega \vdash A$, where A is constructed by that connective. This is also the same as saying that the proof if identity expansion has the right rule as its final inference. We call those with invertible right rules *negative* connectives, while those with non-invertible right rules are *positive* connectives.

$$\begin{array}{lrcl} \text{Negative} & A^-, B^- & ::= & p^- \mid A^+ \setminus B^- \mid B^- \,/\, A^+ \mid A^- \,\&\, B^- \\ \text{Positive} & A^+, B^+ & ::= & p^+ \mid A^+ \bullet B^+ \mid A^+ \circ B^+ \mid \mathbf{1} \mid A^+ \oplus B^+ \end{array}$$

Propositional variables p can be given either polarity. Note that in the case of implicational connectives the polarity of the antecedent is positive, since the role of polarities is reversed for antecedents. At this point, just as in the case of combining logics, we need a way to go back and forth between these classes of propositions. For this we reuse the idea behind shifts, except of going between ordered and linear we remain in the ordered layer, changing only the polarity. The directions of the shifts is determined by the fact that the $\uparrow R$ rule is invertible (and therefore $\uparrow^{\mathsf{o}}_{\mathsf{o}} A$ should be negative) and conversely that the $\downarrow R$ rules is not invertible (and therefore $\downarrow^{\mathsf{o}}_{\mathsf{o}} A$ should be positive).

$$\begin{array}{lrcl} \text{Negative} & A^-, B^- & ::= & p^- \mid A^+ \setminus B^- \mid B^- \,/\, A^+ \mid A^- \,\&\, B^- \mid \uparrow^{\mathsf{o}}_{\mathsf{o}} A^+ \\ \text{Positive} & A^+, B^+ & ::= & p^+ \mid A^+ \bullet B^+ \mid A^+ \circ B^+ \mid \mathbf{1} \mid A^+ \oplus B^+ \mid \downarrow^{\mathsf{o}}_{\mathsf{o}} A^- \end{array}$$

The fact that negative propositions are "above" the positive propositions here does not imply an independence principle: the *mode* of both layers is the same (O, in this case) and therefore a proof of a positive succedent can depend on negative antecedents and vice versa. The rules for these constructs are boring. They will become more interesting when we discuss *focusing* where the shifts will play a critical role. In the purely ordered setting (so all the propositions are ordered) we just have:

$$\frac{\Omega \vdash A^+}{\Omega \vdash \uparrow^{\mathsf{o}}_{\mathsf{o}} A^+}\uparrow R \qquad \frac{\Omega_1\ A^+\ \Omega_2 \vdash C}{\Omega_1\ (\uparrow^{\mathsf{o}}_{\mathsf{o}} A^+)\ \Omega_2 \vdash C}\uparrow L$$

$$\frac{\Omega \vdash A^-}{\Omega \vdash \downarrow^{\mathsf{o}}_{\mathsf{o}} A^-}\downarrow R \qquad \frac{\Omega_1\ A^-\ \Omega_2 \vdash C}{\Omega_1\ (\downarrow^{\mathsf{o}}_{\mathsf{o}} A^-)\ \Omega_2 \vdash C}\downarrow L$$

The operational semantics is the same as for the shifts from the previous lecture. $\uparrow^{\mathsf{o}}_{\mathsf{o}} A^+$ is negative and therefore receives a shift. Conversely $\downarrow^{\mathsf{o}}_{\mathsf{o}} A^-$ is positive and therefore sends a shift.

An interesting aspect of the operational semantics is the overall polarization: when a process offers along a channel $x : A^+$ it will send a message, and then continue to send messages until we reach $x : \mathbf{1}$ and we terminate, or we reach $x : \downarrow^{\mathsf{O}}_{\mathsf{O}} B^-$. In the second case, the process now sends a last message—a shift—and then starts to receive messages according to the type B^-. Again, the process will continuously receive since the type remains negative, until we reach a form $x : \uparrow^{\mathsf{O}}_{\mathsf{O}} C^+$ and the cycle begins anew.

4 Example: Polarizing Stacks

As an example, we reconsider stacks.

$$\begin{array}{l}\mathsf{stack}_A = \&\{\ \mathsf{ins} : A \setminus \mathsf{stack}_A, \\ \qquad\qquad \mathsf{del} : \oplus\{\ \mathsf{none} : \mathbf{1}, \\ \qquad\qquad\qquad\quad \mathsf{some} : A \bullet \mathsf{stack}_A\}\}\end{array}$$

The stack interface starts with an external choice, so it is negative which we indicate by stack^-_A. When we go through an insertion and recurse, the type remains negative except we see that A itself occurs *under* a stack ($A \setminus \mathsf{stack}_A$) and therefore should be positive. Filling in this partial information, we have so far:

$$\begin{array}{l}\mathsf{stack}^-_{A^+} = \&\{\ \mathsf{ins} : A^+ \setminus \mathsf{stack}^-_{A^+}, \\ \qquad\qquad \mathsf{del} : \oplus\{\ \mathsf{none} : \mathbf{1}, \\ \qquad\qquad\qquad\quad \mathsf{some} : A^+ \bullet \mathsf{stack}^-_{A^+}\}\}\end{array}$$

At this point we notice some mismatches. In particular, internal choice is a positive connective, but when we enter the del branch of the external choice we expect a negative proposition. So we need to insert a shift and switch to positive polarity. Conversely, when we reach the recursive appeal to the stack type, we expect a positive polarity while stack is negative so we need another shift. We omit the superscript and subscripts on the shifts, since they always stay at level O.

$$\begin{array}{l}\mathsf{stack}^-_{A^+} = \&\{\ \mathsf{ins} : A^+ \setminus \mathsf{stack}^-_{A^+}, \\ \qquad\qquad \mathsf{del} : \uparrow \oplus\{\ \mathsf{none} : \mathbf{1}, \\ \qquad\qquad\qquad\quad \mathsf{some} : A^+ \bullet \downarrow \mathsf{stack}^-_{A^+}\}\}\end{array}$$

We obtained this type by adding the minimal number of shifts to polarize its unpolarized form. Such a minimal translation always exists and is easy to describe.

Let's talk through some sequences of interactions. As long as we insert elements into the stack, we encounter no shift and the direction of the communication remains the same. We might send, for example, the following sequence of six messages:

$$\text{to provider} \quad \mathsf{ins}\ x_1\ \mathsf{ins}\ x_2\ \mathsf{ins}\ x_3$$

Actually, let's see what this looks like in the explicit notation of messages, assuming we start sending along channel y_1.

$$\begin{array}{l}
\mathsf{msg}(y_1, y_1.\mathsf{ins}\ ;\ y_1 \leftarrow y_2) \\
\mathsf{msg}(y_2, \mathsf{send}\ y_2\ x_1\ ;\ y_2 \leftarrow y_3) \\
\mathsf{msg}(y_3, y_3.\mathsf{ins}\ ;\ y_3 \leftarrow y_4) \\
\mathsf{msg}(y_4, \mathsf{send}\ y_4\ x_2\ ;\ y_4 \leftarrow y_5) \\
\mathsf{msg}(y_5, y_5.\mathsf{ins}\ ;\ y_5 \leftarrow y_6) \\
\mathsf{msg}(y_6, \mathsf{send}\ y_6\ x_3\ ;\ y_6 \leftarrow y_7)
\end{array}$$

We see that the messages form a *linked list segment* from y_1 to y_7 where the mediating channels y_2, y_3, y_4, y_5, y_6 provide the links. These linked list segments in fact act as queues implementing message buffers. In [PG15] we created buffer queues as a separate data structure in the operational semantics, but I no longer believe their syntactic overhead is warranted.

Since communication is buffered and we have no bound on the number of insertions, the buffer must also be unbounded. Continuing the example, if we now decide to delete an element, we have to send a del label and then a shift to change direction of the communication:

$$\begin{array}{ll}
\text{to provider} & \mathsf{ins}\ x_1\ \mathsf{ins}\ x_2\ \mathsf{ins}\ x_3\ \mathsf{del}\ \mathsf{shift} \\
\text{to client} &
\end{array}$$

At this point we wait for the response, which means that the stack provider has to drain the whole buffer, up to and including the final shift and then respond. The response in this case will be the label some, followed by the channel x_3, followed by a shift.

$$\begin{array}{ll}
\text{to provider} & \mathsf{ins}\ x_1\ \mathsf{ins}\ x_2\ \mathsf{ins}\ x_3\ \mathsf{del}\ \mathsf{shift} \\
\text{to client} & \mathsf{some}\ x_3\ \mathsf{shift} \\
\text{to provider} &
\end{array}$$

Continuing for a few more interaction cycles:

to provider	ins x_1 ins x_2 ins x_3 del shift
to client	some x_3 shift
to provider	del shift
to client	some x_2 shift
to provider	del shift
to client	some x_1 shift
to provider	del shift
to client	none close

5 Synchronous from Asynchronous Communication

We recall the polarized type of stacks.

$$\begin{aligned}\mathsf{stack}^-_{A^+} = \&\{\ & \mathsf{ins} : A^+ \setminus \mathsf{stack}^-_{A^+}, \\ & \mathsf{del} : \uparrow \oplus\{\ \mathsf{none} : \mathbf{1}, \\ & \qquad\quad \mathsf{some} : A^+ \bullet \downarrow \mathsf{stack}^-_{A^+}\}\}\end{aligned}$$

As noted, the implementation requires an unbounded buffer because there are no changes of direction (which would be indicated by shifts) if we continuously insert elements.

If we want to keep the buffer bounded we can force synchronization after each element has been inserted. Amazingly, we can express this just be inserting a double-shift like so:

$$\begin{aligned}\mathsf{stack}^-_{A^+} = \&\{\ & \mathsf{ins} : A^+ \setminus \uparrow \downarrow \mathsf{stack}^-_{A^+}, \\ & \mathsf{del} : \uparrow \oplus\{\ \mathsf{none} : \mathbf{1}, \\ & \qquad\quad \mathsf{some} : A^+ \bullet \downarrow \mathsf{stack}^-_{A^+}\}\}\end{aligned}$$

Operationally, the first shift ($\uparrow$) will receive a shift message *from* the client, while the second shift ($\downarrow$) will sent a shift *to* the client and then wait again for messages. This second shift acts as an acknowledgment that all messages before have been received. This is because communication can only change direction when the buffer is empty, that is, the shift at the end of a message sequence has been received. Here is an example interaction, with

the additional synchronization:

to provider	ins x_1 shift
to client	shift
to provider	ins x_2 shift
to client	shift
to provider	ins x_3 shift
to client	shift
to provider	del shift
to client	some x_3 shift
to provider	...

We can now calculate the needed buffer size for this particular data structure and see that it is 3: The longest sequence of messages that can be in the buffer at any time are 'ins x shift' (to the provider) and 'some x shift' (to the client). In general, the calculation finds the longest path of the same polarity through the graph consisting of all the mutually recursive type definitions. If that is infinite, the buffer must be unbounded.

If we want to force fully synchronous communication, that is, every sent message waits for an acknowledgment, we just have to insert a double shift at every point in a type. The first shift sends a shift message, while the one immediately following waits for the acknowledgment in the form of a shift that is returns. This is a bit overly agressive, because if there is already a shift in the type, it is not necesary to make it a triple-shift (see [PG15] for more detail).

One pleasing property of this approach is that synchronization points are not left to the implementation, but are manifest in the type. Contrast this with Section 1 where it is only a matter of the code we write whether we want to force asynchrony in an otherwise synchronous language. This, and the fact that it is a more realistic abstraction, leads me to believe that asynchrony is the better default. The good news is that the proof theory supports both points of view, so there is no real right or wrong.

Exercises

Exercise 1 Provide an implementation of asynchronous send for $\backslash L^*$ as for $\bullet R^*$ in Section 1, and verify that synchronous and asynchronous forms are related by a commuting reduction followed by an identity reduction. If not, explain.

Exercise 2 Recall the correspondence between synchronous and asynchronous sends given at the end of Section 1 which was derived from $\bullet R^*$.

$$\mathsf{send}\ x\ w\ ;\ P \quad \simeq \quad x' \leftarrow [x'/x]P\ ;\ \mathsf{send}\ x\ w\ ;\ x \leftarrow x'$$

Provide analogous correspondences for $\mathbf{1}R$, $\oplus R$, $\&L$, and $\backslash L^*$.

Exercise 3 Develop an alternative semantics for forwarding $x \leftarrow y$ which involves communication only with directly connected processes (either along x or y) rather than the heavy-handed global replacement we have used so far. How are your rules related to the identity reductions (which can be read off from the proof of admissibility of cut)?

Exercise 4 Polarize each of the following types and analyze the maximal buffer sizes. If unbounded, insert synchronization points and then analyze the finite buffer size required for the resulting types.

1. $\mathsf{list}_A = \oplus\{\mathsf{cons} : A \bullet \mathsf{list}_A, \mathsf{nil} : \mathbf{1}\}$
2. $\mathsf{seg}_A = \mathsf{list}_A \,/\, \mathsf{list}_A$
3. $\mathsf{mapper}_{AB} = \&\{\mathsf{next} : (B \bullet \mathsf{mapper}_{AB}) \,/\, A, \mathsf{done} : \mathbf{1}\}$
4. $\mathsf{t\_map}_{AB} = \mathsf{list}_A \setminus (\mathsf{mapper}_{AB} \setminus \mathsf{list}_B)$
5. $\mathsf{folder}_{AB} = \&\{\mathsf{next} : (B \setminus (B \bullet \mathsf{folder}_{AB})) \,/\, A, \mathsf{done} : \mathbf{1}\}$
6. $\mathsf{t\_fold}_{AB} = (B \bullet \mathsf{folder}_{AB} \bullet \mathsf{list}_A) \setminus B$

References

[DCPT12] Henry DeYoung, Luís Caires, Frank Pfenning, and Bernardo Toninho. Cut reduction in linear logic as asynchronous session-typed communication. In P. Cégielski and A. Durand, editors, *Proceedings of the 21st Conference on Computer Science Logic*, CSL 2012, pages 228–242, Fontainebleau, France, September 2012. Leibniz International Proceedings in Informatics.

[PG15] 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 Structural Logic

15-816: Substructural Logics
Frank Pfenning

Lecture 14
October 13, 2016

At this point we introduce *structural logic*, that is, a logic in which we assume all structural rules for the antecedents: exchange, weakening, and contraction. This is a sequent calculus presentation of what is usually called *intuitionistic logic*, which was developed long before substructural logics. Since all the logics we have considered so far are intuitionistic in nature, however, this is not a distinguishing characteristic.

We begin with a brief excursion to consider quantifiers. Quantified variables may occur arbitrarily often in their scope, and they may occur in any order, which means they are the first fully structural entities we see apart from persistent propositions in the first few lectures.

1 Existential Quantification

From a logical perspective, quantifiers allow us to talk about universal (such as "*all men are mortal*") or existential (such as "*some inference systems do not satisfy cut elimination*)" propositions. From a programming perspective, they allow us to compute over basic data values such as integers or strings. Because of the latter, we will insist on quantifiers being *typed*.

We can prove $\exists x{:}\tau.A$ if we can prove $[t/x]A$ for some term t of type τ. This means we require a new judgment, $\Sigma \vdash t : \tau$, where Σ is a collection of assumptions about the type of variables, $x_1{:}\tau_1, \ldots, x_n{:}\tau_n$. Since there are basic terms with (so far) no particular logic interpretation, we will hedge our bets about which rules this judgment needs to satisfy and collect some requirements from our rules. Typing assumptions for variables now also

become part of main sequent calculus judgment which, to be most general, is still ordered.

$$\frac{\Sigma \vdash t : \tau \quad \Sigma \,;\, \Omega \vdash [t/x]A}{\Sigma \,;\, \Omega \vdash \exists x{:}\tau.\ A}\ \exists R \qquad \frac{\Sigma, a{:}\tau \,;\, \Omega_L\ ([a/x]A)\ \Omega_R \vdash C}{\Sigma \,;\, \Omega_L\ (\exists x{:}\tau.\ A)\ \Omega_R \vdash C}\ \exists L^a$$

In the left rule we introduce a *fresh* parameter a, which does not occur in the whole judgment in the conclusion. We may omit the condition via the general presupposition that the variables declared in a context Σ are all distinct so that $\Sigma, a{:}\tau$ only makes sense if a is not yet declared in Σ.

Because quantified variables are intended to occur multiple times in a proposition, but also possibly not at all, the hypotheses in Σ will be subject to exchange, weakening, and contraction, that is, they can be used arbitrarily often. Just as important is the substitution property, formulated here has two admissible rules of inference

$$\frac{\Sigma \vdash t : \tau \quad \Sigma, a{:}\tau \vdash s : \sigma}{\Sigma \vdash [t/a]s : \sigma}\ \mathsf{subst} \qquad \frac{\Sigma \vdash t : \tau \quad \Sigma, a{:}\tau \,;\, \Omega \vdash C}{\Sigma \,;\, [t/a]\Omega \vdash [t/a]C}\ \mathsf{subst}$$

We obtain the conclusion by systematically going through the proof of the second premise and substituting t for a everywhere.

With this background, how does cut reduction play out? Let's write down the cut where the right rule meets the left rule.

$$\frac{\dfrac{\begin{array}{c}\mathcal{T}\\ \Sigma \vdash t : \tau\end{array} \quad \begin{array}{c}\mathcal{D}'\\ \Sigma \,;\, \Omega \vdash [t/x]A\end{array}}{\Sigma \,;\, \Omega \vdash \exists x{:}\tau.\ A}\ \exists R \quad \dfrac{\begin{array}{c}\mathcal{E}'\\ \Sigma, a{:}\tau \,;\, \Omega_L\ ([a/x]A)\ \Omega_R \vdash C\end{array}}{\Sigma \,;\, \Omega_L\ (\exists x{:}\tau.\ A)\ \Omega_R \vdash C}\ \exists L^a}{\Sigma \,;\, \Omega_L\ \Omega\ \Omega_R \vdash C}\ \mathsf{cut}_{\exists x.A}$$

We have already taken the liberty of writing Σ in both premises of the cut: we can always equalize them by weakening in order to bring them into this form, if necessary.

We would like to reduce the cut on $\exists x{:}\tau.\ A$ to a cut on A. But this does not work: $\mathcal{D}'$ is a proof of $[t/x]A$ and the antecedent in $\mathcal{E}'$ is $[a/x]A$. Now we remember that we were asked to choose a to be fresh, is, it doesn't occur in Σ or Ω_L or A or Ω_R or C. So, for example, $[t/a]C = C$. With substitution and these considerations we obtain

$$\begin{array}{c}[t/a]\mathcal{E}'\\ \Sigma \,;\, \Omega_L\ ([t/a][a/x]A)\ \Omega_R \vdash C\end{array}$$

Now we only need to see that $[t/a][a/x]A = [t/x]A$, again because a was chosen fresh and does not occur in A. In summary we then have:

$$\dfrac{\dfrac{\begin{array}{cc}\mathcal{T} & \mathcal{D}' \\ \Sigma \vdash t : \tau & \Sigma\ ;\ \Omega \vdash [t/x]A\end{array}}{\Sigma\ ;\ \Omega \vdash \exists x{:}\tau.\ A}\ \exists R \quad \dfrac{\begin{array}{c}\mathcal{E}' \\ \Sigma, a{:}\tau\ ;\ \Omega_L\ ([a/x]A)\ \Omega_R \vdash C\end{array}}{\Sigma\ ;\ \Omega_L\ (\exists x{:}\tau.\ A)\ \Omega_R \vdash C}\ \exists L^a}{\Sigma\ ;\ \Omega_L\ \Omega\ \Omega_R \vdash C}\ \mathsf{cut}_{\exists x.A}$$

$$\Longrightarrow_R \quad \dfrac{\begin{array}{cc}\mathcal{D}' & [t/a]\mathcal{E}' \\ \Sigma\ ;\ \Omega \vdash [t/x]A & \Sigma\ ;\ \Omega_L\ ([t/x]A)\ \Omega_R \vdash C\end{array}}{\Sigma\ ;\ \Omega_L\ \Omega\ \Omega_R \vdash C}\ \mathsf{cut}_{[t/x]A}$$

If we are counting characters then $[t/x]A$ may be larger than $\exists x.\ A$, but if we consider the *structure* of propositions, then $[t/x]A$ is a subformula of $\exists x.\ A$. We could also resort to just counting the total number of logical connectives and quantifiers, which goes down since t is just a term and may not contain logical connectives or quantifiers.

2 A Computational Interpretation of Existential Quantification

It should be intuitively clear that an application of $\exists R$ carries information, namely the *witness* t. The computational interpretation of $\exists R$ therefore just send the witness, while $\exists L$ receives it.

$$\dfrac{\Sigma \vdash t : \tau \quad \Sigma\ ;\ \Omega \vdash P :: (y : [t/x]A)}{\Sigma\ ;\ \Omega \vdash \mathsf{send}\ y\ t\ ;\ P :: (y : \exists x{:}\tau.\ A)}\ \exists R$$

$$\dfrac{\Sigma, a{:}\tau\ ;\ \Omega_L\ (y : [a/x]A)\ \Omega_R \vdash Q_a :: (z : C)}{\Sigma\ ;\ \Omega_L\ (y : \exists x{:}\tau.\ A)\ \Omega_R \vdash a \leftarrow \mathsf{recv}\ x\ ;\ Q_a :: (z : C)}\ \exists L^a$$

Operationally, all that happens is the term t is transmitted from one process to the other.

$$\dfrac{\mathsf{proc}(y, \mathsf{send}\ y\ t\ ;\ P) \quad \mathsf{proc}(z, a \leftarrow \mathsf{recv}\ x\ ;\ Q_a)}{\mathsf{proc}(y, P) \quad \mathsf{proc}(z, Q_t)}\ \exists C$$

The intention is to only execute processes with no free term variables, and we can see that the computation rule maintains this invariant. This is similar to the restriction that in functional languages we only execute closed

programs. Also important is that there are no linearity or ordering restrictions on term variables $x{:}\tau$ or $a{:}\tau$. Since they stand for basic values such as integers or strings, they can used freely and need not be used.

3 Universal Quantification

The universal quantifier is symmetric to the existential.

$$\frac{\Sigma, a{:}\tau\ ;\ \Omega \vdash [a/x]A}{\Sigma\ ;\ \Omega \vdash \forall x{:}\tau.\ A}\ \forall R^a \qquad \frac{\Sigma \vdash t : \tau \quad \Sigma\ ;\ \Omega_L\ ([t/x]A)\ \Omega_R \vdash C}{\Sigma\ ;\ \Omega_L\ (\forall x{:}\tau.\ A)\ \Omega_R \vdash C}\ \forall L$$

This is reflected in the cut reduction (see Exercise 1). The proof term assignment reuses the same terms as for the existential, but the roles of provider and client are reversed.

$$\frac{\Sigma, a{:}\tau\ ;\ \Omega \vdash P_a :: (y : [a/x]A)}{\Sigma\ ;\ \Omega \vdash a \leftarrow \mathsf{recv}\ y\ ;\ P_a :: (y : \forall x{:}\tau.\ A)}\ \forall R^a$$

$$\frac{\Sigma \vdash t : \tau \quad \Sigma\ ;\ \Omega_L\ (y : [t/x]A)\ \Omega_R \vdash Q :: (z : C)}{\Sigma\ ;\ \Omega_L\ (y : \forall x{:}\tau.\ A)\ \Omega_R \vdash \mathsf{send}\ y\ t\ ;\ P :: (z : C)}\ \forall L$$

Computationally:

$$\frac{\mathsf{proc}(y, a \leftarrow \mathsf{recv}\ y\ ;\ P_a) \quad \mathsf{proc}(z, \mathsf{send}\ y\ t\ ;\ Q)}{\mathsf{proc}(y, P_t) \quad \mathsf{proc}(z, Q)}\ \forall C$$

We can see that order is largely independent to the meaning of the quantifiers, but we have to take care to treat variables structurally, that is, allow them to be used arbitrarily often, in arbitrary order.

Identity expansion, as well as cut and identity elimination go through as before. In the proof of admissibility of cut, we only have to remember that $[t/x]A$ is structurally a subformula of $\forall x{:}\tau.\ A$ and $\exists x{:}\tau.\ A$.

4 Two Logical Examples: Quantifier Exchange

We give[1] two short logical examples which illustrate the quantifiers can be exchanged in one direction but not the other. We use here an uninterpreted

[1] Actually, we skipped these during lecture; they are provided here as bonus material.

type ι about which we make no assumptions.

$$\forall x{:}\iota.\, \exists y{:}\iota.\, A(x,y) \not\vdash \exists y{:}\iota.\, \forall x{:}\iota.\, A(x,y)$$
$$\exists y{:}\iota.\, \forall x{:}\iota.\, A(x,y) \vdash \forall x{:}\iota.\, \exists y{:}\iota.\, A(x,y)$$

Note that $A(x,y)$ is an arbitrary proposition, possibly depending on x and y We start with the first example:

$$\vdots$$
$$\forall x{:}\iota.\, \exists y{:}\iota.\, A(x,y) \vdash \exists y{:}\iota.\, \forall x{:}\iota.\, A(x,y)$$

At this point neither $\forall L$ or $\exists R$ apply, unless we make as assumption about the type ι. Let's assume it has at least one element a, which means we rewrite the judgment as

$$\vdots$$
$$a{:}\iota\ ;\ \forall x{:}\iota.\, \exists y{:}\iota.\, A(x,y) \vdash \exists y{:}\iota.\, \forall x{:}\iota.\, A(x,y)$$

Now we have to make a decision whether to start on the left or on the right. Both attempts will eventually fail to produce a proof—we show only one.

$$\dfrac{\begin{array}{c}\vdots\\ a{:}\iota\ ;\ \exists y{:}\iota.\, A(a,y) \vdash \exists y{:}\iota.\, \forall x{:}\iota.\, A(x,y)\end{array}}{a{:}\iota\ ;\ \forall x{:}\iota.\, \exists y{:}\iota.\, A(x,y) \vdash \exists y{:}\iota.\, \forall x{:}\iota.\, A(x,y)}\ \forall L$$

Next we use $\exists L$, which is actually invertible so we don't have to look for other rules.

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ a{:}\iota, b{:}\iota\ ;\ A(a,b) \vdash \exists y{:}\iota.\, \forall x{:}\iota.\, A(x,y)\end{array}}{a{:}\iota\ ;\ \exists y{:}\iota.\, A(a,y) \vdash \exists y{:}\iota.\, \forall x{:}\iota.\, A(x,y)}\ \exists L^b}{a{:}\iota\ ;\ \forall x{:}\iota.\, \exists y{:}\iota.\, A(x,y) \vdash \exists y{:}\iota.\, \forall x{:}\iota.\, A(x,y)}\ \forall L$$

Looking ahead, we can see the only promising witness to pick for y on the right is a

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ a{:}\iota, b{:}\iota\ ;\ A(a,b) \vdash \forall x{:}\iota.\, A(a,y)\end{array}}{a{:}\iota, b{:}\iota\ ;\ A(a,b) \vdash \exists y{:}\iota.\, \forall x{:}\iota.\, A(x,y)}\ \exists R}{a{:}\iota\ ;\ \exists y{:}\iota.\, A(a,y) \vdash \exists y{:}\iota.\, \forall x{:}\iota.\, A(x,y)}\ \exists L^b}{a{:}\iota\ ;\ \forall x{:}\iota.\, \exists y{:}\iota.\, A(x,y) \vdash \exists y{:}\iota.\, \forall x{:}\iota.\, A(x,y)}\ \forall L$$

Using the $\forall R$ rule however must pick a *new* parameter c. We cannot reuse B. The resulting sequent therefore has no proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{no rule applies}}{a{:}\iota, b{:}\iota, c{:}\iota \;;\; A(a,b) \vdash A(a,c)}\,\forall R^c}{a{:}\iota, b{:}\iota \;;\; A(a,b) \vdash \forall x{:}\iota.\, A(a,y)}\,\exists R}{a{:}\iota, b{:}\iota \;;\; A(a,b) \vdash \exists y{:}\iota.\, \forall x{:}\iota.\, A(x,y)}\,\exists L^b}{a{:}\iota \;;\; \exists y{:}\iota.\, A(a,y) \vdash \exists y{:}\iota.\, \forall x{:}\iota.\, A(x,y)}\,\forall L}{a{:}\iota \;;\; \forall x{:}\iota.\, \exists y{:}\iota.\, A(x,y) \vdash \exists y{:}\iota.\, \forall x{:}\iota.\, A(x,y)}$$

This does not constitute a formal proof that the proposed entailment does not hold, but with cut elimination (which does continue to hold) and refuting all possible attempt, it would turn into one.

The provable example is more straightforward and we show only the completed proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{a{:}\iota, b{:}\iota \;;\; A(a,b) \vdash A(a,b)}\,\mathsf{id}}{a{:}\iota, b{:}\iota \;;\; \forall x{:}\iota.\, A(x,b) \vdash A(a,b)}\,\forall L}{a{:}\iota, b{:}\iota \;;\; \forall x{:}\iota.\, A(x,b) \vdash \exists y{:}\iota\, A(a,y)}\,\exists R}{a{:}\iota \;;\; \exists y{:}\iota.\, \forall x{:}\iota.\, A(x,y) \vdash \exists y{:}\iota\, A(a,y)}\,\exists L^b}{\exists y{:}\iota.\, \forall x{:}\iota.\, A(x,y) \vdash \forall x{:}\iota.\, \exists y{:}\iota.\, A(x,y)}\,\forall R^a$$

5 A Computational Example: Parallel Insertion Sort

As a computational example we will use a parallel form of insertion sort. We add the elements of a list, one by one, into a priority queue and then remove them in order.

$$\begin{aligned}\mathsf{pqueue} = \&\{\; & \mathsf{ins} : \forall x{:}\mathsf{int}.\, \mathsf{pqueue}, \\ & \mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : \exists x{:}\mathsf{int}.\, \mathsf{pqueue}\}\}\end{aligned}$$

We see that the quantified variables do not occur in our language of types. This is because we have not yet introduced *type families*, which correspond to logical predicates. When there is no such dependency, we simplify the notation and write

$$\begin{array}{rcll} \tau \to A & = & \forall x{:}\tau.\, A & \text{for } x \text{ not in } A \\ \tau \wedge A & = & \exists x{:}\tau.\, A & \text{for } x \text{ not in } A \end{array}$$

The type of priority queues then looks slightly simpler.

$$\begin{array}{l}\mathsf{pqueue} = \&\{\ \mathsf{ins} : \mathsf{int} \to \mathsf{pqueue},\\ \qquad\qquad \mathsf{del} : \oplus\{\mathsf{none} : \mathbf{1}, \mathsf{some} : \mathsf{int} \wedge \mathsf{pqueue}\}\}\end{array}$$

We have two process definitions: one to create an empty priority queue and one holding an element n and a channel q referencing the remainder of the queue.

$$\begin{array}{rcl} \cdot\ ;\ \cdot & \vdash & \mathit{empty} :: (p : \mathsf{pqueue}) \\ n{:}\mathsf{int}\ ;\ q{:}\mathsf{pqueue} & \vdash & \mathit{elem} :: (p : \mathsf{pqueue}) \end{array}$$

First, the definition of *empty*, which very much follows our previous implementation of stacks, queues, and stores.

$$\cdot\ ;\ \cdot \vdash \mathit{empty} :: (p : \mathsf{pqueue})$$

$$\begin{array}{l} p \leftarrow \mathit{empty} = \\ \quad \mathsf{case}\ p\ (\mathsf{ins} \Rightarrow n \leftarrow \mathsf{recv}\ p\ ; \\ \qquad\qquad\quad e \leftarrow \mathit{empty}\ ; \\ \qquad\qquad\quad p \leftarrow \mathit{elem}\ n \leftarrow e \\ \qquad\ \mid \mathsf{del} \Rightarrow p.\mathsf{none}\ ;\ \mathsf{close}\ p) \end{array}$$

For the definition of *elem* we indicate the dependence on a parameter n:int by writing elem n on the left-hand side of the definition. We assume functions *max* and *min* on integers.

$$n{:}\mathsf{int}\ ;\ q{:}\mathsf{pqueue} \vdash \mathit{elem} :: (p : \mathsf{pqueue})$$

$$\begin{array}{l} p \leftarrow \mathit{elem}\ n \leftarrow q = \\ \quad \mathsf{case}\ p\ (\mathsf{ins} \Rightarrow k \leftarrow \mathsf{recv}\ p\ ; \\ \qquad\qquad\quad q.\mathsf{ins}\ ;\ \mathsf{send}\ q\ (\mathit{max}\ n\ k)\ ; \\ \qquad\qquad\quad p \leftarrow \mathit{elem}\ (\mathit{min}\ n\ k) \leftarrow q \\ \qquad\ \mid \mathsf{del} \Rightarrow p.\mathsf{some}\ ;\ \mathsf{send}\ p\ n\ ; \\ \qquad\qquad\quad p \leftarrow q) \end{array}$$

Even though we reuse the syntax for sending and receiving of channels, sending and receiving of basic data values has a different semantics. In particular, they are not subject to linearity or order even though this example lies squarely within ordered logic. The fact that both n and k are used twice in the ins branch is perfectly legal. The type checker, of course, will know through the assigned type whether variables stand for data values or channels. Current implementations track this more obviously by distinguishing channels syntactically from ordinary variables.

To sort a list in ascending order, we add all the elements from the list into the priority queue, and then remove them all. The lists are slightly different from the lists before because they hold integers, not channels.

$$\mathsf{list} = \oplus\{\mathsf{cons} : \mathsf{int} \wedge \mathsf{list}, \mathsf{nil} : \mathbf{1}\}$$

We have three process definitions: *sort* tieing everything together *enqueue* which transfers the elements from the input list to a priority queue, and *dequeue* which transfers the elements from the priority to the output list.

$$\begin{array}{rcl} k{:}\mathsf{list} & \vdash & \mathit{sort} :: (l : \mathsf{list}) \\ (k{:}\mathsf{list})\ (p{:}\mathsf{pqueue}) & \vdash & \mathit{enqueue} :: (l : \mathsf{list}) \\ p{:}\mathsf{pqueue} & \vdash & \mathit{dequeue} :: (l : \mathsf{list}) \end{array}$$

$k{:}\mathsf{list} \vdash \mathit{sort} :: (l : \mathsf{list})$
$l \leftarrow \mathit{sort} \leftarrow k =$
  $p \leftarrow \mathit{empty}$;
  $l \leftarrow \mathit{enqueue} \leftarrow k\ p$

$(k{:}\mathsf{list})\ (p{:}\mathsf{pqueue}) \vdash \mathit{enqueue} :: (l : \mathsf{list})$

$l \leftarrow \mathit{enqueue} \leftarrow k\ p =$
  $\mathsf{case}\ k\ (\mathsf{cons} \Rightarrow n \leftarrow \mathsf{recv}\ k$;
    $p.\mathsf{ins}$; $\mathsf{send}\ p\ k$;
    $l \leftarrow \mathit{enqueue} \leftarrow k\ p$
  $\mid \mathsf{nil} \Rightarrow\ \mathsf{wait}\ k$;
    $l \leftarrow \mathit{dequeue} \leftarrow p)$

$p{:}\mathsf{pqueue} \vdash \mathit{dequeue} :: (l : \mathsf{list})$

$l \leftarrow \mathit{dequeue} \leftarrow p =$
  $p.\mathsf{del}$;
  $\mathsf{case}\ p\ (\mathsf{none} \Rightarrow\ \mathsf{wait}\ p$;
    $l.\mathsf{nil}$; $\mathsf{close}\ l$
  $\mid \mathsf{some} \Rightarrow n \leftarrow \mathsf{recv}\ p$;
    $l.\mathsf{cons}$; $\mathsf{send}\ l\ n$;
    $l \leftarrow \mathit{dequeue} \leftarrow p)$

Let's do a quick parallel complexity analysis. Our implementation of priority queues is structurally the same as the previous implementation of queues, stacks, and stores. Therefore, the reaction time for both insertions and deletions is constant. To sort a list of length n we perform n insertions

followed by n deletions, which gives $O(n)$ throughput and latency for sorting. The total amount of work is $O(n^2)$ since each insertion will take $O(k)$ work, where k is the length of the list so far, while each deletion takes a constant amount of work.

6 Structural Intuitionistic Logic

We now introduce plaing intuitionistic logic, with the structural rules of exchange, weakening, and contraction. First, we build in exchange with the same technique as for linear logic: The collection of antecedents is considered as a multiset where the order of the antecedents does not matter. We write $\Gamma \vdash A$ for such a sequent in structural intuitionistic logic.

For weaking and contraction, there are two standard techniques. One is to add the following explicit rules:

$$\frac{\Gamma \vdash C}{\Gamma, A \vdash C}\ \mathsf{weaken} \qquad \frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C}\ \mathsf{contract}$$

These are structural rules and independent of what any particular proposition A might be.

There are two downsides to this simple and elegant formulation of structural logic. One is that it introduces a lot of nondeterminism into proof construction, since at any point we could decide to apply either weakening and contraction. Contraction in particular is prolific, since it could be immediately applied again, and again, etc. The other downside it that the rule of contraction significantly complicates the proof of admissibility of cut.

An alternative approach is to build weakening and contraction into the rules themselves. *Weakening* is incorporated by allowing unused antecedent in the identity rule (and also the $\top R$ and $\bot L$ rules later; see Section 6). *Contraction* is incorporated by propagating all antecedents to all premises of multi-premise rules. We illustrate with id, cut and the rules for implication $A \to B$.

$$\frac{}{\Gamma, A \vdash A}\ \mathsf{id}_A \qquad \frac{\Gamma \vdash A \quad \Gamma, A \vdash C}{\Gamma \vdash C}\ \mathsf{cut}_A$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \to R \qquad \frac{\Gamma, A \to B \vdash A \quad \Gamma, A \to B, B \vdash C}{\Gamma, A \to B \vdash C} \to L$$

Notice in particular how the antecedent $A \to B$ in the $\to L$ rule persists in both premises. This is in fact a good intuition for this formulation of

the logic: antecedents are *persistent*, just as we had persistent propositions when discussing logical inference. From the perspective of provability, some of these antecedents are redundant. For example, in the second premise of $\to L$, the persistent antecedent $A \to B$ is no longer needed since we already have the strong assumption B. These kinds of optimizations often interfere with our operational interpretation of the logic. Therefore, they should be carefully considered in each situation rather than being built into the very definition of the logic.

A first, important observation is that weakening is an admissible rule. Moreover, the proof resulting from weakening has *exactly the same structure* as the proof before weakening, because we only adjoin an extra unused antecedent to every sequent. This means if we perform structural induction over proofs, we can freely apply weakening and still apply the induction hypothesis.

Theorem 1 (Admissibility of Weakening and Contraction)

$$\frac{\Gamma \vdash C}{\Gamma, A \vdash C}\ \mathsf{weaken} \qquad \frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C}\ \mathsf{contract}$$

Moreover, in both cases the proof of the conclusion is structurally identical to the proof of the premise.

Proof: Formally, by structural induction on the given proofs. For weakening, we adjoin an extra, unused antecedent A to every sequent in the given proof to obtain the proof of the conclusion. For contraction, we replace every use of either one of the two copies of A in every sequent of the given proof by a use of the same, single A to obtain the proof of the conclusion. □

There are some new and interesting cases in the proof of admissibility of cut. We show only two: a new case for the identity, and the principal case for implication. As before, we write $\Gamma \Vdash A$ for a cut-free proof of A from Γ.

Theorem 2 (Admissibility of Cut in the Cut-Free Structural Sequent Calculus)

$$\frac{\Gamma \Vdash A \quad \Gamma, A \Vdash C}{\Gamma \Vdash C}\ \mathsf{cut}$$

Proof: By a nested induction, first on the structure of the cut formula A, and second on the proofs $\mathcal{D}$ and $\mathcal{E}$ of the two premises. The proof breaks down into the same classes of cases as before: identity, principal, and commutative cases.

Case:

$$\overset{\mathcal{D}}{\Gamma', B \Vdash A} \quad \text{arbitrary, and} \quad \mathcal{E} = \frac{}{\Gamma', B, A \Vdash B}\ \mathsf{id}_B$$

where $\Gamma = (\Gamma', B)$. We need to show $\Gamma', B \Vdash B$ which follows by id_B. What is interesting about this case is that $\mathcal{D}$ is dropped entirely because it proves an unused proposition A.

Case:

$$\mathcal{D} = \frac{\overset{\mathcal{D}_1}{\Gamma, A_2 \Vdash A_1}}{\Gamma \Vdash A_2 \to A_1}\ {\to}R \quad \text{and} \quad \mathcal{E} = \frac{\overset{\mathcal{E}_2}{\Gamma, A_2 \to A_1 \Vdash A_2} \quad \overset{\mathcal{E}_1}{\Gamma, A_2 \to A_1, A_1 \Vdash C}}{\Gamma, A_2 \to A_1 \Vdash C}\ {\to}L$$

We need to show $\Gamma \Vdash C$. We would like to cut $\mathcal{E}_2$ with $\mathcal{D}_1$ and $\mathcal{D}_1$ with $\mathcal{E}_1$, but there will be an extra copy of $A_2 \to A_1$ left over. Se we first apply what is called a *cross-cut*, applying the induction hypothesis to *all of* $\mathcal{D}$ and $\mathcal{E}_2$ and also $\mathcal{E}_1$ to eliminate the persistent copies of $A_2 \to A_1$.

$\mathcal{E}_2' = \Gamma \Vdash A_2$	By i.h. on $A_2 \to A_1$, $\mathcal{D}$, and $\mathcal{E}_2$
$\mathcal{D}' = \Gamma, A_1 \Vdash A_2 \to A_1$	By weakening on $\mathcal{D}'$
$\mathcal{E}_1' = \Gamma, A_1 \Vdash C$	By i.h. on $A_2 \to A_2$, $\mathcal{D}'$, and $\mathcal{E}_1$
$\mathcal{D}_1' = \Gamma \Vdash A_1$	By i.h. on A_2, $\mathcal{E}_2'$, and $\mathcal{D}_1$
$\mathcal{F} = \Gamma \Vdash C$	By i.h. on A_1, $\mathcal{D}_1'$, and $\mathcal{E}_1'$

□

Please make sure you understand why each appeal to the induction hypothesis in this proof is justified.

There is not much more to say about the rules for the other connectives. What may be interesting is that, logically speaking, the two forms of conjunction $A \otimes B$ and $A \mathbin{\&} B$ "collapse" in that they are now logically equivalent. However, computationally they can still be different, so from an intuitionistic perspective we should be careful before eliminating one or the other. We therefore avoid writing $A \wedge B$ for conjunction, prefering $A \mathbin{\&} B$ for the conjunction with the same notation in linear logic, and $A \times B$

for that corresponding to $A \otimes B$ in linear logic.

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \,\&\, B}\,\&R \qquad \frac{\Gamma, A \,\&\, B, A \vdash C}{\Gamma, A \,\&\, B \vdash C}\,\&L_1 \quad \frac{\Gamma, A \,\&\, B, B \vdash C}{\Gamma, A \,\&\, B \vdash C}\,\&L_2$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \times B}\,\times R \qquad \frac{\Gamma, A \times B, A, B \vdash C}{\Gamma, A \times B \vdash C}\,\times L$$

Disjunction $A \vee B$ is unsurprising and has the rules completely analogous to the linear case for $A \oplus B$, except for the persistence of the disjunction in the $\vee L$ case.

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\,\vee R_1 \quad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\,\vee R_2 \qquad \frac{\Gamma, A \vee B, A \vdash C \quad \Gamma, A \vee B, B \vdash C}{\Gamma, A \vee B \vdash C}\,\vee L$$

Finally, the cases for $\top$ and $\bot$, which are the units $\&$ and $\vee$, respectively.

$$\frac{}{\Gamma \vdash \top}\,\top R \qquad \text{no } \top L \text{ rule}$$

$$\text{no } \bot R \text{ rule} \qquad \frac{}{\Gamma, \bot \vdash C}\,\bot L$$

Exercises

Exercise 1 Show the cut reduction for universal quantification.

Exercise 2 Show the identity expansion for existential quantification.

Exercise 3 Show the identity expansion for universal quantification.

Exercise 4 *Show two cases in the proof of admissibility of cut for quantifiers in ordered logic.*

1. *The principal case for the cut formula $\exists x{:}\tau.\ A$.*
2. *A commutative case for an $\exists L$ rule on a side formula of the cut.*

Exercise 5 The parallel insertion sort in Section 5 uses the priority queue in a restricted manner: first, it only inserts integers and then it only deletes integers.

Create more precise types for *enqueue* and *dequeue* which enforce this protocol. Under which conditions or rules do *empty*, *elem*, *enqueue*, *dequeue* and *sort* type-check against these more precise types?

Exercise 6 Implement set of integers as binary search trees. Use the interface

$$
\begin{array}{l}
\mathsf{list} = \oplus\{\mathsf{cons} : \mathsf{int} \wedge \mathsf{list}, \mathsf{nil} : \mathbf{1}\} \\
\mathsf{bool} = \oplus\{\mathsf{true} : \mathbf{1}, \mathsf{false} : \mathbf{1}\} \\
\mathsf{tree} = \&\{\ \mathsf{ins} : \mathsf{int} \to \mathsf{tree}, \\
\qquad\qquad \mathsf{member} : \mathsf{int} \to \mathsf{bool} \bullet \mathsf{tree}, \\
\qquad\qquad \mathsf{to\_list} : \mathsf{list}\}
\end{array}
$$

where to_list should create a sorted list in ascending order. The tree does not need to be balanced. Also implement one additional binary operation on trees such as union or intersection.

Exercise 7 We reconsider the principal case for $A_2 \to A_1$ in the proof of admissibility of cut in structural intutionistic logic. Try to apply the induction hypothesis first to $\mathcal{E}_2$ and (a weakened form of) $\mathcal{D}_1$ and then to (a weakened form of the result) and $\mathcal{E}_1$. Then eliminate the extra copy of $A_2 \to A_1$ from the result with another appeal to the induction hypothesis with $\mathcal{D}$. Poinpoint exactly where this argumentation goes wrong.

Exercise 8 Prove the principal case for $A_1 \times A_2$ in the proof of admissibility of cut for structural intuitionistic logic.

Lecture Notes on Of Course!

15-816: Substructural Logics
Frank Pfenning

Lecture 15
October 20, 2016

In this lecture we combine structural and substructural intuitionistic logics into a single system, using the previously discussed device of shift modalities to go between layers. This is the idea behind Benton's LNL [Ben94], who basis his constructions on the categorical concept of an adjunction.

As we have already seen, the idea is quite general. In this lecture we will restrict ourselves to the structural and linear fragments, leaving other considerations such as ordered, affine, or strict logics to a future lecture in order to reduce it to its essentials. From our approach the exponential modality $!A$ (read: *of course A!*) of Girard's linear logic [Gir87] will arise naturally as a composition of two shifts.

We will also resume our analysis of the operational interpretation of shifts, to see specifically what happens in the context of LNL.

1 Combining Linear and Structural Logic

Our starting point quite straightforwardly follows the approach laid out in Lecture 12. We have two separate layers of propositions, connected by two shifts. We use U (suggesting *unrestricted*) for the structural mode and L, as before, for the linear mode. We have, by definition $\mathsf{U} > \mathsf{L}$ since U admits exchange, weakening, and contraction while L admits only exchange.

$$\begin{array}{lrcl} \text{Structural} & A_{\mathsf{U}} & ::= & \ldots \mid A_{\mathsf{U}} \to B_{\mathsf{U}} \mid \uparrow^{\mathsf{U}}_{\mathsf{L}} A_{\mathsf{L}} \\ \text{Linear} & A_{\mathsf{L}} & ::= & \ldots \mid A_{\mathsf{L}} \multimap B_{\mathsf{L}} \mid \downarrow^{\mathsf{U}}_{\mathsf{L}} A_{\mathsf{U}} \end{array}$$

The independence principle states:

> *A structural succedent may not depend on a linear antecedent.*

We can make this explicit by allowing only the following two judgment forms:

$$\Gamma_{\mathsf{U}} \vdash A_{\mathsf{U}}$$
$$\Gamma_{\mathsf{U}} \,;\, \Delta_{\mathsf{L}} \vdash A_{\mathsf{L}}$$

It is possible to separate the antecedents into two zones since both modes admit exchange. From now on we will use Γ only for structural antecdents and Δ only for linear ones, so we can omit the subscript.

To begin with, we obtain the following rules of identity and cut. The phenomenon of obtain three cut rules should be familiar from Lecture 12. As we pointed out there, they can be unified into a single rule using *adjoint logic* [Ree09].

$$\frac{}{\Gamma, A_{\mathsf{U}} \vdash A_{\mathsf{U}}}\ \mathsf{id}_{\mathsf{U}} \qquad \frac{}{\Gamma \,;\, A_{\mathsf{L}} \vdash A_{\mathsf{L}}}\ \mathsf{id}_{\mathsf{L}}$$

$$\frac{\Gamma \vdash A_{\mathsf{U}} \quad \Gamma, A_{\mathsf{U}} \vdash C_{\mathsf{U}}}{\Gamma \vdash C_{\mathsf{U}}}\ \mathsf{cut}_{\mathsf{UU}} \qquad \frac{\Gamma \vdash A_{\mathsf{U}} \quad \Gamma, A_{\mathsf{U}} \,;\, \Delta \vdash C_{\mathsf{L}}}{\Gamma \,;\, \Delta \vdash C_{\mathsf{L}}}\ \mathsf{cut}_{\mathsf{UL}}$$

$$\frac{\Gamma \,;\, \Delta \vdash A_{\mathsf{L}} \quad \Gamma \,;\, \Delta', A_{\mathsf{L}} \vdash C_{\mathsf{L}}}{\Gamma \,;\, \Delta', \Delta \vdash C_{\mathsf{L}}}\ \mathsf{cut}_{\mathsf{LL}}$$

As pointed out in the last lecture, the presence of weakening and contraction allows us to view structural antecedents as persistent, so they are propagated to all premises of each inference rule.

The next question are the rules for the shift modalities. They follow exactly the same pattern as before, with a small twist to incorporate the persistence of structural assumptions in the $\uparrow L$ rule. They are entirely based on the independence principle, which is built into the formulation of the judgments themselves.

$$\frac{\Gamma \,;\, \cdot \vdash A_{\mathsf{L}}}{\Gamma \vdash \uparrow^{\mathsf{U}}_{\mathsf{L}} A_{\mathsf{L}}}\ \uparrow R \qquad \frac{\Gamma, \uparrow^{\mathsf{U}}_{\mathsf{L}} A_{\mathsf{L}} \,;\, \Delta, A_{\mathsf{L}} \vdash C_{\mathsf{L}}}{\Gamma, \uparrow^{\mathsf{U}}_{\mathsf{L}} A_{\mathsf{L}} \,;\, \Delta \vdash C_{\mathsf{L}}}\ \uparrow L$$

$$\frac{\Gamma \vdash A_{\mathsf{U}}}{\Gamma \,;\, \cdot \vdash \downarrow^{\mathsf{U}}_{\mathsf{L}} A_{\mathsf{U}}}\ \downarrow R \qquad \frac{\Gamma, A_{\mathsf{U}} \,;\, \Delta \vdash C_{\mathsf{L}}}{\Gamma \,;\, \Delta, \downarrow^{\mathsf{U}}_{\mathsf{L}} A_{\mathsf{U}} \vdash C_{\mathsf{L}}}\ \downarrow L$$

The resulting system enjoys all the important properties such as admissibility of cut and identity and cut elimination. At the interface between the two judgment there is one instance of a cross-cut when $\uparrow R$ is matched against $\uparrow L$. We dispense with the details since there are no particular new ideas to be conveyed.

2 Operational Interpretation of Shifts, Revisited

Before we dive into the operational interpretation of the full structural component of this combined logic, we look only at the shifts. The $\uparrow R$ and $\downarrow L$ rules are invertible, that is, they can always be applied when the proposition $\uparrow A_{\mathsf{L}}$ appears on the right or $\downarrow A_{\mathsf{U}}$ appears on the left. This means these two rules will receive while $\uparrow L$ and $\downarrow R$ will send. What they send and receive is an indication to shift to a new mode of communication. Before, when we shifted, we re-used the same channel. This is no longer possible now because in the $\uparrow L$ rule the persistent channel remains. First, the proof term assignment.

$$\frac{\Gamma\ ;\cdot \vdash P_y :: (y_{\mathsf{L}} : A_{\mathsf{L}})}{\Gamma \vdash y_{\mathsf{L}} \leftarrow (\mathsf{L})x_{\mathsf{U}}\ ;\ P_y :: (x_{\mathsf{U}} : \uparrow^{\mathsf{U}}_{\mathsf{L}} A_{\mathsf{L}})}\ \uparrow R$$

$$\frac{\Gamma, (x_{\mathsf{U}} : \uparrow^{\mathsf{U}}_{\mathsf{L}} A_{\mathsf{L}})\ ;\ \Delta, (y_{\mathsf{L}} : A_{\mathsf{L}}) \vdash Q_y :: (z_{\mathsf{L}} : C_{\mathsf{L}})}{\Gamma, (x_{\mathsf{U}} : \uparrow^{\mathsf{U}}_{\mathsf{L}} A_{\mathsf{L}})\ ;\ \Delta \vdash y_{\mathsf{L}} \leftarrow (\mathsf{L})x_{\mathsf{U}}\ ;\ Q_y :: (z_{\mathsf{L}} : C_{\mathsf{L}})}\ \uparrow L$$

In the synchronous communication rule we mark a process offering along a persistent channel itself as persistent. This is because this process may have multiple clients (inclding in Q_y!), so we cannot evolve it, but we can spawn a fresh linear copy offering along the new channel w_{L}.

$$\frac{\underline{\mathsf{proc}}(x_{\mathsf{U}}, y_{\mathsf{L}} \leftarrow (\mathsf{L})x_{\mathsf{U}}\ ;\ P_y) \quad \mathsf{proc}(z_{\mathsf{L}}, y_{\mathsf{L}} \leftarrow (\mathsf{L})x_{\mathsf{U}}\ ;\ Q_y)}{\mathsf{proc}(w_{\mathsf{L}}, P_w) \quad \mathsf{proc}(z_{\mathsf{L}}, Q_w)}\ \uparrow C^w$$

It is even more obvious in the synchronous version that the persistent process must receive, otherwise it could continuously spawn new messages!

$$\frac{\mathsf{proc}(z_{\mathsf{L}}, y_{\mathsf{L}} \leftarrow (\mathsf{L})x_{\mathsf{U}}\ ;\ Q_y)}{\mathsf{msg}(w_{\mathsf{L}}, y_{\mathsf{L}} \leftarrow (\mathsf{L})x_{\mathsf{U}}\ ;\ w_{\mathsf{L}} \leftarrow y_{\mathsf{L}}) \quad \mathsf{proc}(z_{\mathsf{L}}, Q_w)}\ \uparrow C^w\_\mathsf{send}$$

$$\frac{\underline{\mathsf{proc}}(x_{\mathsf{U}}, y_{\mathsf{L}} \leftarrow (\mathsf{L})x_{\mathsf{U}}\ ;\ P_y) \quad \mathsf{msg}(w_{\mathsf{L}}, y_{\mathsf{L}} \leftarrow (\mathsf{L})x_{\mathsf{U}}\ ;\ w_{\mathsf{L}} \leftarrow y_{\mathsf{L}})}{\mathsf{proc}(w_{\mathsf{L}}, P_w)}\ \uparrow C^w\_\mathsf{recv}$$

The rules for $\downarrow^{\mathsf{U}}_{\mathsf{L}} A$ reverse the roles.

$$\frac{\Gamma \vdash P_y :: (y_{\mathsf{U}} : A_{\mathsf{U}})}{\Gamma\ ;\cdot \vdash y_{\mathsf{U}} \leftarrow (\mathsf{U})x_{\mathsf{L}}\ ;\ P_y :: (x_{\mathsf{L}} : \downarrow^{\mathsf{U}}_{\mathsf{L}} A_{\mathsf{U}})}\ \downarrow R$$

$$\frac{\Gamma, (y_{\mathsf{U}} : A_{\mathsf{U}})\ ;\ \Delta \vdash Q_y :: (z_{\mathsf{L}} : C_{\mathsf{L}})}{\Gamma\ ;\ \Delta, (x_{\mathsf{L}} : \downarrow^{\mathsf{U}}_{\mathsf{L}} A_{\mathsf{U}}) \vdash y_{\mathsf{U}} \leftarrow (\mathsf{U})x_{\mathsf{L}}\ ;\ Q_y :: (z_{\mathsf{L}} : C_{\mathsf{L}})}\ \downarrow L$$

In the operational semantics we *create* a persistent process.

$$\frac{\mathsf{proc}(x_{\mathsf{L}}, y_{\mathsf{U}} \leftarrow (\mathsf{U})x_{\mathsf{L}} \;;\; P_y) \quad \mathsf{proc}(z_{\mathsf{L}}, y_{\mathsf{U}} \leftarrow (\mathsf{U})x_{\mathsf{L}} \;;\; Q_y)}{\underline{\mathsf{proc}}(w_{\mathsf{U}}, P_w) \quad \mathsf{proc}(z_{\mathsf{L}}, Q_w)} \downarrow C^w$$

Here, the first process sends the persistent channel, as we can see from the asynchronous version of the semantics.

$$\frac{\mathsf{proc}(x_{\mathsf{L}}, y_{\mathsf{U}} \leftarrow (\mathsf{U})x_{\mathsf{L}} \;;\; P_y) \quad \mathsf{proc}(z_{\mathsf{L}}, y_{\mathsf{U}} \leftarrow (\mathsf{U})x_{\mathsf{L}} \;;\; Q_y)}{\underline{\mathsf{proc}}(w_{\mathsf{U}}, P_w) \quad \mathsf{msg}(x_{\mathsf{L}}, y_{\mathsf{U}} \leftarrow (\mathsf{U})x_{\mathsf{L}} \;;\; y_{\mathsf{U}} \leftarrow w_{\mathsf{U}})} \downarrow C^w \_\mathsf{send}$$

$$\frac{\mathsf{msg}(x_{\mathsf{L}}, y_{\mathsf{U}} \leftarrow (\mathsf{U})x_{\mathsf{L}} \;;\; y_{\mathsf{U}} \leftarrow w_{\mathsf{U}}) \quad \mathsf{proc}(z_{\mathsf{L}}, y_{\mathsf{U}} \leftarrow (\mathsf{U})x_{\mathsf{L}} \;;\; Q_y)}{\mathsf{proc}(z_{\mathsf{L}}, Q_w)} \downarrow C^w \_\mathsf{recv}$$

3 Example: Map

As we have seen in Exercise 10.2 of Lecture 10, it is possible to define concurrent versions of map and fold using recursive types. Another natural approach is to allow the transformer that is mapped over a list to be persistent. We abbreviate $\uparrow^{\mathsf{U}}_{\mathsf{L}} A$ as $\uparrow A$ and similarly for $\downarrow^{\mathsf{U}}_{\mathsf{L}} A$ since in this lecture we only consider structural and linear modes. We leave linear channels undecorated and write x_{U} for unrestricted channels which can be used arbitrarily often.

$\mathsf{list}_A = \oplus\{\mathsf{cons} : A \otimes \mathsf{list}_A, \mathsf{nil} : \mathbf{1}\}$
$f_{\mathsf{U}} : \uparrow(A \multimap B) \;;\; k{:}\mathsf{list}_A \vdash \mathit{map} :: (l : \mathsf{list}_B)$

$l \leftarrow \mathit{map} \leftarrow k \; f_{\mathsf{U}} =$
  $\mathsf{case}\ k\ \ (\mathsf{nil} \Rightarrow \mathsf{wait}\ k \;;\; l.\mathsf{nil} \;;\; \mathsf{close}\ l$
  $\quad\quad | \ \mathsf{cons} \Rightarrow \ldots)$

We already took advantage here of weakening: f_{U} is not used in the nil branch of *map*. We will need it twice in the cons branch: once to apply to the element, and then pass it on to be mapped over the rest of the list.

$\mathsf{list}_A = \oplus\{\mathsf{cons} : A \otimes \mathsf{list}_A, \mathsf{nil} : \mathbf{1}\}$
$f_{\mathsf{U}} : \uparrow(A \multimap B)$; $k{:}\mathsf{list}_A \vdash \mathit{map} :: (l : \mathsf{list}_B)$

$l \leftarrow \mathit{map} \leftarrow k\ f_{\mathsf{U}} =$
  $\mathsf{case}\ k\ \ (\mathsf{nil} \Rightarrow \mathsf{wait}\ k\ ;\ l.\mathsf{nil}\ ;\ \mathsf{close}\ l$
  $\quad \mid \mathsf{cons} \Rightarrow x \leftarrow \mathsf{recv}\ k\ ;$
  $\qquad\qquad y \leftarrow (\mathsf{L})f_{\mathsf{U}}\ ;$ $\quad\%\quad y{:}A \multimap B$
  $\qquad\qquad \mathsf{send}\ y\ x\ ;$ $\quad\%\quad y{:}B$
  $\qquad\qquad l.\mathsf{cons}\ ;\ \mathsf{send}\ l\ y\ ;$
  $\qquad\qquad l \leftarrow \mathit{map} \leftarrow k\ f_{\mathsf{U}})$

Below is a slight different style of expressing this computation proposed in lecture. It uses library processes for *nil* and *cons*. Here, the call to *map* is not a tail call. This means, without any further optimizations, it will spawn a new process and therefore it is likely less efficient. We don't particularly care at this point about low-level efficiency or even how many processes are spawned, but it is still worth noting this difference.

$\mathsf{list}_A = \oplus\{\mathsf{cons} : A \otimes \mathsf{list}_A, \mathsf{nil} : \mathbf{1}\}$
$\cdot \vdash \mathit{nil} :: (l : \mathsf{list}_A)$
$x{:}A, k{:}\mathsf{list}_A \vdash \mathit{cons} :: (l : \mathsf{list}_A)$

$f_{\mathsf{U}} : \uparrow(A \multimap B)$; $k{:}\mathsf{list}_A \vdash \mathit{map} :: (l : \mathsf{list}_B)$

$l \leftarrow \mathit{map} \leftarrow k\ f_{\mathsf{U}} =$
  $\mathsf{case}\ k\ \ (\mathsf{nil} \Rightarrow \mathsf{wait}\ k\ ;\ l \leftarrow \mathit{nil}$
  $\quad \mid \mathsf{cons} \Rightarrow x \leftarrow \mathsf{recv}\ k\ ;$
  $\qquad\qquad y \leftarrow (\mathsf{L})f_{\mathsf{U}}\ ;$ $\quad\%\quad y{:}A \multimap B$
  $\qquad\qquad \mathsf{send}\ y\ x\ ;$ $\quad\%\quad y{:}B$
  $\qquad\qquad l' \leftarrow \mathit{map} \leftarrow k\ f_{\mathsf{U}}\ ;$
  $\qquad\qquad l \leftarrow \mathit{cons} \leftarrow l'\ y)$

As a sample mapping function we write one that turns an element into a singleton list.

$x{:}A \vdash \mathit{singleton} :: (l{:}\mathsf{list}_A)$
$l \leftarrow \mathit{singleton} \leftarrow x =$
  $n \leftarrow \mathit{nil}\ ;$
  $l \leftarrow \mathit{cons} \leftarrow x\ n$

$\cdot \vdash \mathit{map\_singleton} :: (f_u : \uparrow(A \multimap \mathsf{list}_A))$

$f_u \leftarrow \mathit{map\_singleton} =$
  $y \leftarrow (\mathsf{L})f_u\ ;$

$x \leftarrow \mathsf{recv}\ y\ ;$
$y \leftarrow \mathsf{singleton} \leftarrow x$

4 Of Course!

In Girard's linear logic [Gir87] there is no explicit structural layer, but we have a so-called exponential modality $!A$ (pronounced "*of course* A" or sometimes "*bang* A" to allow an A to be used multiple times. Briefly, Girard started from the idea that the intuitionistic function space $A \to B$ could be decomposed into a modality and a linear function space $(!A) \multimap B$. This idea of a fine-grained analysis of computation while retaining the means to express all the prior functions also pervades this course.

We arrived at this idea following a different path. Rather than decomposing existing languages, we have started from a substructural point of view and added various liberties. Starting from this direction we notice that $!A \simeq \downarrow^{\mathsf{U}}_{\mathsf{L}}\uparrow^{\mathsf{U}}_{\mathsf{L}} A\mathsf{L}$. In the (intuitionistic) version of Girard's logic, we have the following inference rules pertaining to the exponential modality.

$$\frac{\Delta \vdash C}{\Delta, !A \vdash C}\ \mathsf{weaken} \qquad \frac{\Delta, !A, !A \vdash C}{\Delta, !A \vdash C}\ \mathsf{contract}$$

$$\frac{!\Delta \vdash A}{!\Delta \vdash !A}\ !R \qquad \frac{\Delta, A \vdash C}{\Delta, !A \vdash C}\ !L$$

Here we use $!\Delta$ is stand for a collection of antecedents all of whose propositions have the form $!A$. While there are many interesting semantic approaches to understand this and its classical counterpart, the proof theory and the cut elimination proofs are not nearly as elegant as for the substructural approach we have followed here. Some notes on prior work can be found in [Pfe94, CCP03].

Using the expansion of $!A_{\mathsf{L}} = \downarrow\uparrow A_{\mathsf{L}}$ we can validate all of the rules above by showing that they are admissible. For example:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\uparrow A \vdash \uparrow A}\ \mathsf{id}_{\mathsf{U}}}{\uparrow A\ ;\cdot \vdash !A}\downarrow R \quad \dfrac{\dfrac{}{\uparrow A \vdash \uparrow A}\ \mathsf{id}_{\mathsf{U}}}{\uparrow A\ ;\cdot \vdash !A}\downarrow R}{\uparrow A\ ;\cdot \vdash !A \otimes !A}\otimes R}{\cdot\ ;\ !A \vdash !A \otimes !A}\downarrow L \qquad \dfrac{\cdot\ ;\ \Delta, !A, !A \vdash C}{\cdot\ ;\ \Delta, !A \otimes !A \vdash C}\otimes L}{\Delta, !A \vdash C}\ \mathsf{cut}$$

Because of several standard embeddings of structural intuitionistic logic in linear logic [TCP12], this means that in a certain sense we do not need the full structural layer in what we have discussed. It is sufficient to just have

$$\begin{array}{lllll} \text{Structural} & A_{\mathsf{U}} & ::= & \uparrow^{\mathsf{U}}_{\mathsf{L}} A_{\mathsf{L}} \\ \text{Linear} & A_{\mathsf{L}} & ::= & \ldots \mid A_{\mathsf{L}} \multimap B_{\mathsf{L}} \mid \downarrow^{\mathsf{U}}_{\mathsf{L}} A_{\mathsf{U}} \end{array}$$

and the rest is a question of *pragmatics*: how easy or difficult is it to program certain algorithms in the resulting language as compared to coding them when the structural layer is more complete.

Exercises

Exercise 1 Implement the operation of fold from Exercise 10.3, but with fold a persistent process analogous to map in Section 3.

Exercise 2 Show that all the rules for intuitionistic linear logic using $!A$ are derivable or admissible in the combined structural/linear logic using the definition of $!A$ as $\downarrow_{\mathsf{L}}^{\mathsf{U}}\uparrow_{\mathsf{L}}^{\mathsf{U}} A_{\mathsf{L}}$. Give a derivation of the rule where possible.

Exercise 3 Prove that if $\vdash A_{\mathsf{L}}$ and A_{L} uses $\downarrow_{\mathsf{L}}^{\mathsf{U}}\uparrow_{\mathsf{L}}^{\mathsf{U}}$ as its only modality, then it is provable in linear logic under the rules in Section 4.

References

[Ben94] 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 (CLS'94)*, Kazimierz, Poland, September 1994. Springer LNCS 933. An extended version appears as Technical Report UCAM-CL-TR-352, University of Cambridge.

[CCP03] 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.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Pfe94] Frank Pfenning. Structural cut elimination in linear logic. Technical Report CMU-CS-94-222, Department of Computer Science, Carnegie Mellon University, December 1994.

[Ree09] Jason Reed. A judgmental deconstruction of modal logic. Unpublished manuscript, 2009.

[TCP12] Bernardo Toninho, Luís Caires, and Frank Pfenning. Functions as session-typed processes. In L. Birkedal, editor, *15th International Conference on Foundations of Software Science and Computation Structures*, FoSSaCS'12, pages 346–360, Tallinn, Estonia, March 2012. Springer LNCS.

Lecture Notes on Computation in Structural Logic

15-816: Substructural Logics
Frank Pfenning

Lecture 16
October 25, 2016

In this lecture we attempt to extend our computational interpretation of ordered and linear logic to include structural logic (all of these being intuitionistic, of course). This completes what we started in Lecture 15, Section 2 where we provided an operational semantics only for the shift operators.

Our analysis will turn out to be different from the usual, celebrated *Curry-Howard isomorphism* between intuitionstic logic and the simply-typed λ-calculus [How69]. There are two reasons for this divergence. For one, and perhaps most importantly, we are working here with a sequent calculus instead of natural deduction. This means that the engine of computation is cut reduction, instead of the usual substitution. Cut reduction is a local transformation on a proof and proceeds in very small steps. You may remember the slogan of *cut reduction as communication*. In natural deduction, *substitution* is the engine of computation which is a much more global, "big step" operation. These two have been related in the past, most notably perhaps is Herbelin's analysis [Her94]. One new ingredient here is the explicit presence of concurrency, and that integration of ordered, linear, and structural computations.

Disclaimer: This lecture very much represents my very recent understanding of the state of affairs, based mostly on intuition without any carefully formulated, much less checked proofs.

1 Structural Intuitionistic Logic

We are aiming at the following combined system.

$$\begin{array}{rrcl} \text{Structural} & A_{\mathsf{U}} & ::= & p_{\mathsf{U}} \mid A_{\mathsf{U}} \to B_{\mathsf{U}} \mid A_{\mathsf{U}} \mathbin{\&} B_{\mathsf{U}} \mid A_{\mathsf{U}} \times B_{\mathsf{U}} \mid \mathbf{1} \mid A_{\mathsf{U}} + B_{\mathsf{U}} \mid \uparrow^{\mathsf{U}}_{\mathsf{L}} A_{\mathsf{L}} \\ \text{Linear} & A_{\mathsf{L}} & ::= & p_{\mathsf{L}} \mid A_{\mathsf{L}} \multimap B_{\mathsf{L}} \mid A_{\mathsf{L}} \mathbin{\&} B_{\mathsf{L}} \mid A_{\mathsf{L}} \otimes B_{\mathsf{L}} \mid \mathbf{1} \mid A_{\mathsf{L}} \oplus B_{\mathsf{L}} \mid \downarrow^{\mathsf{U}}_{\mathsf{L}} A_{\mathsf{U}} \end{array}$$

We already treated the linear mode and the shift modalites, although a slight update will be necessary. So today we focus on the structural layer only. Because of that, we will omit the subscript U and just write A, B, etc.

We begin with $A \times B$. It seems odd that our structural logic contains both $A \times B$ and $A \mathbin{\&} B$. Indeed, it turns out that they are logically equivalent in the sense that $A \times B \vdash A \mathbin{\&} B$ and $A \mathbin{\&} B \vdash A \times B$. But they behave very differently operationally, so it may be worthwhile to support both.

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \times B}\ \times R \qquad \frac{\Gamma, A \times B, A, B \vdash C}{\Gamma, A \times B \vdash C}\ \times L$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \mathbin{\&} B}\ \mathbin{\&} R \qquad \frac{\Gamma, A \mathbin{\&} B, A \vdash C}{\Gamma, A \mathbin{\&} B \vdash C}\ \mathbin{\&} L_1 \qquad \frac{\Gamma, A \mathbin{\&} B, B \vdash C}{\Gamma, A \mathbin{\&} B \vdash C}\ \mathbin{\&} L_2$$

Since antecedents persist, the two right rules for $A \times B$ and $A \mathbin{\&} B$ coincide! This should raise a red flag, but the presence of weakening and contraction allows us to verify harmony for each of the connectives despite the differing left rules. This exemplifies a lesson I learned after many years of working in this field: the presence of structural rules makes observations "fuzzier", systems less crisp, and allows one to get away with some things that are not as elegant as one would like. This is one reason that I am teaching this course now and start with the weakest logic I could easily make sense of (subsingleton logic), working my way up to the present point (combined structural and substructural logics).

The next point will be to derive a one-premise right rule for $A \times B$. As a reminder: we do this here (and also for $A \to B$ in Section 3) so that the spawning of new processes is limited to the cut rule, which is a significant simplification of the operational semantics. We can go back and forth between the one-premise and two-premise rules, using cut in one direction

and identity in the other.

$$\frac{\Gamma, A \vdash B}{\Gamma, A \vdash A \times B}\ \times R^*$$

$$\frac{\Gamma \vdash A \quad \dfrac{\dfrac{\Gamma \vdash B}{\Gamma, A \vdash B}\ \mathsf{weaken}}{\Gamma, A \vdash A \times B}\ \times R^*}{\Gamma \vdash A \times B}\ \mathsf{cut}_A \qquad \frac{\dfrac{}{\Gamma, A \vdash A}\ \mathsf{id}_A \quad \Gamma, A \vdash B}{\Gamma, A \vdash A \times B}\ \times R$$

2 Assigning Process Terms to Proofs of Positive Propositions

Next, we have to design a process assignment. Experience dictates that we should try the sending rule first. By analogy with the linear logic, where $\otimes R$ sends, here, $\times R$ should send.

$$\frac{\Gamma, w{:}A \vdash P :: (x : B)}{\Gamma, w{:}A \vdash \mathsf{send}\ x\ w\ ;\ P :: (x : A \times B)}\ \times R^*$$

Persistence of antecedents implies that even though we send w along x, we also retain w. The left rule is more complicated: in the premise, we have to figure out how to label the new antecedent A and B. Previously, we would have written $x{:}B$, but now we need to retain $x{:}A \times B$ since it may continue to occur in Q, so we rename it to x'.

$$\frac{\Gamma, x{:}A \times B, y{:}A, x'{:}B \vdash Q :: (z : C)}{\Gamma, x{:}A \times B \vdash (y, x') \leftarrow \mathsf{recv}\ x\ ;\ Q :: (z : C)}\ \times L$$

This means we actually receive two channels: $y{:}A$, corresponding to w that is being sent, and a continuation channel $x'{:}B$. In the end, though, perhaps it is not too surprising that $x{:}A \times B$ will send both a $y{:}A$ and an $x'{:}B$. Both of these, together with x and also z can occur in Q. We have chosen not to display this dependence explicitly but rely on the judgment in the premise to express this information.

Now, however, we should be concerned with a mismatch: $\times R^*$ appears to send only one channel (namely w) while $\times L$ expects two. But during asynchronous communication we also have to create a new channel x' to represent the continuation of the process, so that

$$\mathsf{send}\ x\ w\ ;\ P \simeq x' \leftarrow [x'/x]P\ ;\ (\mathsf{send}\ x\ w\ ;\ x \leftarrow x')$$

Using this as a guidance, we get the following rule which introduces a new continuation channel c and substitutes this for x in P.

$$\frac{\mathsf{proc}(x, \mathsf{send}\ x\ w\ ;\ P)}{\mathsf{proc}(c, [c/x]P) \quad \mathsf{msg}(x, \mathsf{send}\ x\ w\ ;\ x \leftarrow c)}\ \times C\_\mathsf{send}^c$$

The message reads: *send w along x and continue as c.* So we do indeed have two channels for the recipient, even if one is not explicit in the syntax of the sender.

$$\frac{\mathsf{msg}(x, \mathsf{send}\ x\ w\ ;\ x \leftarrow c) \quad \mathsf{proc}(z, (y, x') \leftarrow \mathsf{recv}\ x\ ;\ Q)}{\mathsf{proc}(z, [c/x][w/y]Q)}\ \times C\_\mathsf{recv}$$

Ah, we have ignored one important aspect here: these rules are written as if all communications are linear. But they are not! The original channel $x : A \times B$ along which we offer may have multiple clients. This seems okay when the message is sent (since now $\mathsf{msg}(x, \ldots)$ provides along x), but it becomes problematic when the message is received. In order to avoid that other clients of x to be left dangling, we make this message *persistent*.

$$\frac{\mathsf{proc}(x, \mathsf{send}\ x\ w\ ;\ P)}{\mathsf{proc}(c, [c/x]P) \quad \underline{\mathsf{msg}}(x, \mathsf{send}\ x\ w\ ;\ x \leftarrow c)}\ \times C\_\mathsf{send}^c$$

$$\frac{\underline{\mathsf{msg}}(x, \mathsf{send}\ x\ w\ ;\ x \leftarrow c) \quad \mathsf{proc}(z, (y, x') \leftarrow \mathsf{recv}\ x\ ;\ Q)}{\mathsf{proc}(z, [c/x][w/y]Q)}\ \times C\_\mathsf{recv}$$

The processes themselves proceed with their continuations P and Q, after some channel substitution, after the persistent message has been sent or received, respectively.

We next consider disjunction as another positive proposition, $A + B$ (usually written as $A \vee B$) or, as a more convenient type, $+\{l_i : A_i\}_{i \in I}$. First, logically:

$$\frac{\Gamma \vdash A_k}{\Gamma \vdash +\{l_i : A_i\}_{i \in I}}\ +R_k \qquad \frac{\Gamma, +\{l_i : A_i\}_{i \in I}, A_i \vdash C \quad (\text{for all } i \in I)}{\Gamma, +\{l_i : A_i\}_{i \in I} \vdash C}\ +L$$

Again, judging merely from the perspective of provability, the antecedent $+\{l_i : A_i\}_{i \in I}$ is redundant, but we adhere to the principle of persistence of antecedents in structural logic. The process terms look quite similar to the

rules for linear $\oplus$, but we have to account for the continuation which we call x' in the rule.

$$\frac{\Gamma \vdash P :: (x : A_k)}{\Gamma \vdash x.l_k \ ; P :: (x : +\{l_i : A_i\}_{i \in I})} \ +R_k$$

$$\frac{\Gamma, x{:}+\{l_i : A_i\}_{i \in I}, x'{:}A_i \vdash Q_i :: (z : C) \quad (\text{for all } i \in I)}{\Gamma, x{:}+\{l_i : A_i\}_{i \in I} \vdash \mathsf{case}\ x\ (l_i(x') \Rightarrow Q_i)_{i \in I} :: (z : C)} \ +L$$

The computation rule follows the previous pattern: since the offer is along a persistent channel, we create a persistent message and a fresh continuation channel c.

$$\frac{\mathsf{proc}(x, x.l_k \ ; P)}{\mathsf{proc}(c, [c/x]P) \quad \underline{\mathsf{msg}}(x, x.l_k \ ; x \leftarrow c)} \ +C\_\mathsf{send}^c$$

Receiving the message will select the correct branch and also substitute the continuation channel c for x'.

$$\frac{\underline{\mathsf{msg}}(x, x.l_k \ ; x \leftarrow c) \quad \mathsf{proc}(z, \mathsf{case}\ x\ (l_i(x') \Rightarrow Q_i)_{i \in I})}{\mathsf{proc}(z, [c/x']Q_k)} \ +C\_\mathsf{recv}$$

We do not need to mention the message in the conclusion since it is persistent. Persistence is again critical since there may be many clients of x and we cannot leave them dangling without a provider. One of these clients could be Q_k itself since it may depend on x. As was noted in lecture, this dependence on Q_k in x is not strictly necessary, but it is sometimes convenient.

As the last positive proposition we have $\mathbf{1}$. The only novelty here is that we have no continuation.

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

The left rule looks like a typo, but it is not. The principal formula of the inference persists, but no other antecedents are generated. Operationally, it may make a little more sense.

$$\frac{}{\Gamma \vdash \mathsf{close}\ x :: (x : \mathbf{1})} \ \mathbf{1}R \qquad \frac{\Gamma, x{:}\mathbf{1} \vdash Q :: (z : C)}{\Gamma, x{:}\mathbf{1} \vdash \mathsf{wait}\ x \ ; Q :: (z : C)} \ \mathbf{1}L$$

$$\frac{\mathsf{proc}(x, \mathsf{close}\ x)}{\underline{\mathsf{msg}}(x, \mathsf{close}\ x)} \ \mathbf{1}C\_\mathsf{send} \qquad \frac{\underline{\mathsf{msg}}(x, \mathsf{close}\ x) \quad \mathsf{proc}(z, \mathsf{wait}\ x \ ; Q)}{\mathsf{proc}(z, Q)} \ \mathbf{1}C\_\mathsf{recv}$$

This means potentially many clients can check that a persistent provider has terminated by closing its persistent channel. There does not seem to be much point to allow this in Q, but we should be careful and find a good proof-theoretic justification for omitting $x{:}\mathbf{1}$ in the premise before we change our rules.

Now we can think about the meaning of purely positive types, such as

$$\mathsf{list}_A = +\{\mathsf{cons} : A \times \mathsf{list}_A, \mathsf{nil} : \mathbf{1}\}$$

We see a parallel with functional programming here: any datatype declaration with eager constructors and no embedded functions are represented here as a purely positive recursive type. The labels of the sum represent the constructors.

Image we have a process $P :: (x : \mathsf{list}_A)$. If it runs to completion, if will asynchronously send a number of persistent messages. For example, if P sends messages corresponding to the list $[a, b]$, they would look like:

$\underline{\mathsf{msg}}(x, x.\mathsf{cons}\ ;\ x \leftarrow x_1)$
$\underline{\mathsf{msg}}(x_1, \mathsf{send}\ x_1\ a\ ;\ x_1 \leftarrow x_2)$
$\underline{\mathsf{msg}}(x_2, x_2.\mathsf{cons}\ ;\ x_2 \leftarrow x_3)$
$\underline{\mathsf{msg}}(x_3, \mathsf{send}\ x_3\ b\ ;\ x_3 \leftarrow x_4)$
$\underline{\mathsf{msg}}(x_4, x_4.\mathsf{nil}\ ;\ x_4 \leftarrow x_5)$
$\underline{\mathsf{msg}}(x_5, \mathsf{close}\ x_5)$

We can see that this is an explicit linked list representation of the list, where channels act like pointers. These messages are persistent, which means multiple clients can access this data structure, simultaneously in multiple places. It is important however that is is *immutable*, where receiving a message is synonymous with reading the associated data.

In some sense this is a somewhat wasteful representation. We could for example, construct longer messages which would be outside of our currently envisioned grammer of what messages are. For example, to represent all of these in one big message, we could have

$\underline{\mathsf{msg}}(x, x.\mathsf{cons}\ ;\ \mathsf{send}\ x\ a\ ;\ x.\mathsf{cons}\ ;\ \mathsf{send}\ x\ b\ ;\ x.\mathsf{nil}\ ;\ \mathsf{close}\ x)$

However, contrary to what I said in lecture, such compact representations are actually more difficult in structural logic since they preclude direct access to the middle of these blocks, or they require new messages to be created when one is received. In the linear case, this would be less problematic than here.

Nevertheless, there is a logical technique called *focusing* that may justify big blocks of messages. In fact, focusing will be the subject of the next few lectures in this course.

3 Assigning Processes to Proofs for Negative Propositions

The negative propositions in the structural fragments are $A^+_{\mathsf{U}} \to B^-_{\mathsf{U}}$, $\uparrow^{\mathsf{U}}_{\mathsf{L}} A^+_L$, and $A^-_{\mathsf{U}} \mathbin{\&} B^-_{\mathsf{U}}$. The pattern for $\uparrow^{\mathsf{U}}_{\mathsf{L}} A_L$ in the last lecture was different from what we saw above for positive connectives. Essentially, a persistent process of type $\uparrow A$ just waited to receive a shift to a fresh linear channel c and then spawned a fresh copy of itself which offered along c. When we had only one proposition in the structural layer, this was sufficient. Here, we have to ask how we obtain a persistent process in the first place. Not every process can be persistent, since processes, whether they offer along a persistent channel or not, must be able to make progress in their computation. As far as I can see, this problem is best solved by having another type of semantic object in addition to proc and msg which is a persistent service srvc. We may view msg and srvc as duals, where msg sends while srvc receives.

But, first, the one-premise version of the usual rules.

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ {\to}R \qquad \frac{\Gamma, A, A \to B, B \vdash C}{\Gamma, A, A \to B \vdash C}\ {\to}L^*$$

When assigning process terms, sending is somewhat tricky. We have to send a $w{:}A$ along a channel $x{:}A \to B$ and also a continuation channel $x'{:}B$.

$$\frac{\Gamma, w{:}A, x{:}A \to B, x'{:}B \vdash Q :: (z : C)}{\Gamma, w{:}A, x{:}A \to B \vdash x' \leftarrow \mathsf{send}\ x\ w\ ;\ Q :: (z : C)}\ {\to}L^*$$

The syntax here might suggest that we pass w to x and receive an x', but sending is actually asynchronous and we send a continuation channel for x' as well. We can then later communicate along that new channel to communicate further with the recipient process. The right rule is simpler by comparison.

$$\frac{\Gamma, y{:}A \vdash P :: (x : B)}{\Gamma \vdash y \leftarrow \mathsf{recv}\ x\ ;\ P :: (x : A \to B)}\ {\to}R$$

Computationally, receiving along a persistent channel with potentially many clients means that we create a persistent service.

$$\frac{\mathsf{proc}(x, y \leftarrow \mathsf{recv}\ x\ ;\ P)}{\underline{\mathsf{srvc}}(x, y \leftarrow \mathsf{recv}\ x\ ;\ P)}\ {\to}C\_\mathsf{srvc}$$

Now send and recv use a linear message.

$$\frac{\mathsf{proc}(z, x' \leftarrow \mathsf{send}\ x\ w\ ;\ Q)}{\mathsf{msg}(c, x' \leftarrow \mathsf{send}\ x\ w\ ;\ c \leftarrow x') \quad \mathsf{proc}(z, [c/x']Q)}\ \rightarrow C\_\mathsf{send}^c$$

$$\frac{\underline{\mathsf{srvc}}(x, y \leftarrow \mathsf{recv}\ x\ ;\ P) \quad \mathsf{msg}(c, x' \leftarrow \mathsf{send}\ x\ w\ ;\ c \leftarrow x')}{\mathsf{proc}(c, [c/x][w/y]P)}\ \rightarrow C\_\mathsf{recv}$$

We leave $\&\{l_i : A_i\}_{i \in I}$ to Exercise 1.

4 Examples: Map and Finite Differences

We use two simple examples to illustrate programming. The first is to map a process of type $A \rightarrow B$ over a list. The second computes a list of differences between elements of a given list.

$\mathsf{list}_A = +\{\mathsf{cons} : A \times \mathsf{list}_A, \mathsf{nil} : \mathbf{1}\}$

$f{:}A \rightarrow B, k{:}\mathsf{list}_A \vdash \mathit{map} :: (l : \mathsf{list}_B)$

$l \leftarrow \mathit{map} \leftarrow f\ k =$
  $\mathsf{case}\ k\ \ (\mathsf{nil}(k_1) \Rightarrow \mathsf{wait}\ k_1\ ;\ l.\mathsf{nil}\ ;\ \mathsf{close}\ l$
  $\quad\quad | \ \mathsf{cons}(k_1) \Rightarrow (x, k_2) \leftarrow \mathsf{recv}\ k_1$
  $\quad\quad\quad\quad\quad\quad\quad\quad y \leftarrow \mathsf{send}\ f\ x$
  $\quad\quad\quad\quad\quad\quad\quad\quad l.\mathsf{cons}\ ;\ \mathsf{send}\ l\ y$
  $\quad\quad\quad\quad\quad\quad\quad\quad l \leftarrow \mathit{map} \leftarrow f\ k)$

For the second example we use the type of integer lists and an existential quantifiers $\exists x{:}\mathsf{int}.\ \mathsf{intlist}$, abbreviated as $\mathsf{int} \wedge \mathsf{intlist}$. Note that it satisfies the same rules as $A \times B$, except it sends an integer instead of a channel of type A.

Our example takes a list of integers and computes the list of differences between successive integers, which will be one element shorter unless the given list is already empty. We avoid further syntactic sugar, which should not be too difficult to imagine.

$\mathsf{intlist} = +\{\mathsf{cons} : \mathsf{int} \wedge \mathsf{intlist}, \mathsf{nil} : \mathbf{1}\}$

$k{:}\mathsf{intlist} \vdash \mathit{diffs} :: (l : \mathsf{intlist})$

$l \leftarrow \mathit{diffs} \leftarrow k =$
  $\mathsf{case}\ k\ \ (\mathsf{nil}(k_1) \Rightarrow \mathsf{wait}\ k_1\ ;\ l.\mathsf{nil}\ ;\ \mathsf{close}\ l$

$$
\begin{array}{l}
\mid \mathsf{cons}(k_1) \Rightarrow (y, k_2) \leftarrow \mathsf{recv}\ k_1\ ; \\
\qquad\qquad \mathsf{case}\ k_2\ \ (\mathsf{nil}(k_3) \Rightarrow \mathsf{wait}\ k_3\ ;\ l.\mathsf{nil}\ ;\ \mathsf{close}\ l \\
\qquad\qquad\qquad\qquad \mid \mathsf{cons}(k_3) \Rightarrow (z, k_4) \leftarrow \mathsf{recv}\ k_3\ ; \\
\qquad\qquad\qquad\qquad\qquad\qquad l.\mathsf{cons}\ ;\ \mathsf{send}\ l\ (z - y)\ ; \\
\qquad\qquad\qquad\qquad\qquad\qquad l \leftarrow \mathit{diffs} \leftarrow k_2))
\end{array}
$$

A key aspect of this example, which makes it non-linear, is that the recursive call to *diffs* is passed k_2, which is the tail of k, rather than k_4, which is the tail of the tail (and which is is ignored). This structure arises because we look ahead one element in the list to compute the difference.

5 Upshift, Revisited

In the more general setting of this lecture, we revise the computation rules for the up modality slightly, taking advantage of the $\underline{\mathsf{srvc}}$ predicate.

$$\frac{\mathsf{proc}(z_{\mathsf{L}}, y_{\mathsf{L}} \leftarrow (\mathsf{L})x_{\mathsf{U}}\ ;\ Q_y)}{\mathsf{msg}(w_{\mathsf{L}}, y_{\mathsf{L}} \leftarrow (\mathsf{L})x_{\mathsf{U}}\ ;\ w_{\mathsf{L}} \leftarrow y_{\mathsf{L}}) \quad \mathsf{proc}(z_{\mathsf{L}}, Q_w)}\ \uparrow C^w\_\mathsf{send}$$

$$\frac{\mathsf{proc}(x_{\mathsf{U}}, y_{\mathsf{L}} \leftarrow (\mathsf{L})x_{\mathsf{U}}\ ;\ P_y)}{\underline{\mathsf{srvc}}(x_{\mathsf{U}}, y_{\mathsf{L}} \leftarrow (\mathsf{L})x_{\mathsf{U}}\ ;\ P_y)}\ \uparrow C\_\mathsf{srvc}$$

$$\frac{\underline{\mathsf{srvc}}(x_{\mathsf{U}}, y_{\mathsf{L}} \leftarrow (\mathsf{L})x_{\mathsf{U}}\ ;\ P_y) \quad \mathsf{msg}(w_{\mathsf{L}}, y_{\mathsf{L}} \leftarrow (\mathsf{L})x_{\mathsf{U}}\ ;\ w_{\mathsf{L}} \leftarrow y_{\mathsf{L}})}{\mathsf{proc}(w_{\mathsf{L}}, P_w)}\ \uparrow C^w\_\mathsf{recv}$$

Exercises

Exercise 1 Give the process term assignment and computation rules for $\&\{l_i : A_i\}_{i \in I}$ in structural logic.

Exercise 2 Prove the logical equivalence between $A \& B$ and $A \times B$ in structural logic. The write out the processes

$p{:}A \,\&\, B \vdash \mathit{back} :: (q : A \times B)$
$q{:}A \times B \vdash \mathit{forth} :: (p : A \,\&\, B)$

where $A \,\&\, B = \&\{\mathsf{inl} : A, \mathsf{inr} : B\}$ and the proof term assignment and computation rules come from Exercise 1.

Can you say succinctly what these two processes do?

References

[Her94] Hugo Herbelin. A lambda-calculus structure isomorphic to Gentzen-style sequent calculus structure. In L. Pacholski and J. Tiuryn, editors, *8th International Workshop on Computer Science Logic*, pages 61–75, Kazimierz, Poland, September 1994. Springer LNCS 933.

[How69] 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.

Lecture Notes on Chaining

15-816: Substructural Logics
Frank Pfenning

Lecture 17
October 27, 2016

In this lecture we return to the basics of logics: how do we organize efficient proof search? This question was the origin of proof theory and it remains a central one, even if the questions around proof reduction have obtained a comparable status.

What are the key papers in history of proof theory? I am not a historian, but Frege [Fre79] seems to have initiated the formal study of mathematical proof, and Hilbert [Hil05] the study of the structure of proofs to show that certain axiomatic systems are free from contradiction. A major milestone was Gentzen's dissertation [Gen35] in which he introduces natural deduction, the sequent calculus, and cut elimination (his *Hauptsatz*) from which consistency follows easily. The discovery of the Curry-Howard isomorphism between intuitionistic natural deduction and the typed λ-calculus [How69] was a central discovery by establishing a strong connection between programming notations and constructive logic. On the side of proof search, Andreoli [And92] introduced focusing for linear logic [Gir87], which provides a much deeper understanding of proof search than either natural deduction or the sequent calculus. Focusing has proven as universal as cut elimination and eventually reshaped our understanding of proof search. It has also provided us with significant insights into computation based on proof reduction.

We will cover focusing in this and the next lecture, although it will come up throughout the remainder of this course.

1 Inversion, Chaining, and Focusing

Focusing can be seen as arising from two tightly coupled but complementary observations about proof search. The first, perhaps most easily understood, is that certain rules in the sequent calculus are *invertible*, which means that the premises can be proved if and only if the conclusion can be proved. If we see an opportunity to apply such a rule when constructing a proof in a bottom-up way, we can always safely do so. Actually, Andreoli's notion of *asynchronous connective* [And92] is slightly more refined because it abstracts away from the particular way in which we write the rules. Because of an unfortunate clash of terminology with concurrency theory, we just refer to asynchronous connectives as *negative*. A connective is *negative* if we can always decompose it via its right rule, independently of the rest of the sequent. For example, $A \mathbin{\&} B$ is negative, because whenever we are confronted with the goal of proving $\Omega \vdash A \mathbin{\&} B$ we can break this down into the subgoals $\Omega \vdash A$ and $\Omega \vdash B$ without thinking or pausing. If there is a proof, there will be one ending with the $\&R$ rule. On the other hand, $A \bullet B$ is not negative because, for example, a cut-free proof of $A \bullet (B \bullet C) \vdash (A \bullet B) \bullet C$ must end in a $\bullet L$ rule. Remarkably, connectives that are not negative turn out to be *positive* in the sense that we can always decompose them with a left rule when they appear as antecedents, independently of the rest of the sequent. There are other ways to define positive and negative connectives (see, for example, Zeilberger [Zei09]).

Chaining is a somewhat less obvious concept. Let's call a sequent *stable* if it contains only negative antecedents and positive succedents, which means that none of its propositions can a priori be decomposed. When we have reached a stable sequent we have a choice between whether to apply a right rule to the succedent or a left rule to one of the antecedents. Chaining says we can make this decision and then continue to apply right or left rules on this particular proposition and its subformulas as long as they remain positive (on the right) or negative (on the left). For example, we may be trying to prove

$$\Omega \vdash (A \oplus B) \oplus (C \oplus D)$$

for negative propositions A, B, C, and D. Because $\oplus$ is positive, we may not be able to apply a right rule, but *if* we decide to do so we can decompose this all the way to prove one of $\Omega \vdash A$ or $\Omega \vdash B$ or $\Omega \vdash C$ or $\Omega \vdash D$. We do not have to pause and consider a left rule for an antecedent after applying $\oplus R$ once.

Chaining can be considered independently of inversion: during proof

search we can decide to focus on a particular positive succedent or negative antecedent even if the sequent is not stable. We can then continue to focus on its subformulas as we continue in proof search.

Inversion and chaining can be applied independently during proof search. Proofs which satisfy both strategies are called *focused*.

2 Capturing Chaining

A pervasive theme in proof theory is to capture strategies of proof search as deductive calculi. In fact, the sequent calculus was devised by Gentzen as a way to capture proof search in natural deduction. This approach has many benefits, most importantly perhaps that theorems about proof search strategies can be stated and proven without reference to an explicit external language of strategies. Instead, it often turn out (as it will here) that key properties become *internal* properties of a deductive system and therefore become subject to the battery of techniques from proof theory.

In this lecture we are interested in calculi that are more restrictive than cut-free, identity-expanded proofs. This means, proofs contain no applications of cut and the identity rule is only applied to atomic propositions. Unless otherwise stated, you should assume this for all deductive systems in today's lecture.

At a high level of abstraction, natural deduction arises from a single judgment, that of truth, and a single judgmental concept, that of *hypothetical judgment*. Sequent calculus arises when we introduce a distinction between antecedent and succedent, leading to two judgments still connected by a hypothetical judgment. Chaining arises if we have three judgments: antecedents, succedents, and propositions in focus. We also need a further principle, namely unicity: there can be at most one proposition in focus. We have already seen unicity in singleton logic and (implicitly) in all other calculi since there can be at most one succedent in a sequent.

Before we write down the judgments, we commit to the polarized form of the logic we have already seen in Section 3 of Lecture 13 where we used it to characterize communication behavior. You might recall that processes of positive type send while processes of negative type receive. Moreover, any polarization $A^{\pm}$ of an ordinary proposition A is provable if and only if A is.

$$\begin{array}{lrcl} \text{Negative} & A^-, B^- & ::= & p^- \mid A^+ \setminus B^- \mid B^- \,/\, A^+ \mid A^- \,\&\, B^- \mid \uparrow A^+ \\ \text{Positive} & A^+, B^+ & ::= & p^+ \mid A^+ \bullet B^+ \mid A^+ \circ B^+ \mid \mathbf{1} \mid A^+ \oplus B^+ \mid \downarrow A^- \end{array}$$

The considerations above mean that only a positive proposition can be in focus as a succedent, and only a negative proposition can be in focus as an antecedent. We have three forms of judgment, where we write $[A]$ for a proposition in focus. When we do not indicate the polarity of a proposition it can be either positive or negative.

$$\begin{array}{rcl} \Omega & \Vdash & A \\ \Omega & \Vdash & [C^+] \\ \Omega_L\ [A^-]\ \Omega_R & \Vdash & C \end{array}$$

As a shorthand, we write $\overline{\Omega}$ for Ω or $\Omega_L\ [A^-]\ \Omega_R$ and $\overline{C}$ for C or $[C^+]$. We maintain the following presupposition for all judgments:

There is at most one proposition in focus in any sequent.

The right rules for negative propositions or left rules for positive proposition can be applied at any time and are not subject to focus. For example:

$$\frac{A^+\ \overline{\Omega} \Vdash B^-}{\overline{\Omega} \Vdash A^+ \setminus B^-}\setminus R \qquad \frac{\overline{\Omega}_L\ A\ B\ \overline{\Omega}_R \Vdash \overline{C}}{\overline{\Omega}_L\ (A \bullet B)\ \overline{\Omega}_R \Vdash \overline{C}}\bullet L$$

$$\frac{\overline{\Omega} \Vdash A^+}{\overline{\Omega} \Vdash \uparrow A^+}\uparrow R \qquad \frac{\overline{\Omega}_L\ A^-\ \overline{\Omega}_R \Vdash \overline{C}}{\overline{\Omega}_L\ (\downarrow A^-)\ \overline{\Omega}_R \Vdash \overline{C}}\downarrow L$$

To enter a phase of chaining we pick an arbitrary proposition of the correct polarity and put it into focus. This is a judgmental rule in the sense that it does not depend on any particular logical connective.

$$\frac{\Omega \Vdash [A^+]}{\Omega \Vdash A^+}\ \mathsf{focus}^+ \qquad \frac{\Omega_L\ [A^-]\ \Omega_R \Vdash C}{\Omega_L\ A^-\ \Omega_R \Vdash C}\ \mathsf{focus}^-$$

Once a proposition has been focused on, we can apply right and left rules to them. We show some sample set of rules.

$$\frac{\Omega_L \Vdash [A^+] \quad \Omega_R \Vdash [B^+]}{\Omega_L\ \Omega_R \Vdash [A^+ \bullet B^+]}\bullet R \qquad \frac{\Omega \Vdash [A^+] \quad \Omega_L\ [B^-]\ \Omega_R \Vdash C}{\Omega_L\ \Omega\ [A^+ \setminus B^-]\ \Omega_R \Vdash C}\setminus L$$

In each rule, the subformulas remain in focus. This is one reason why in $A^+ \setminus B^-$, A is positive while B is negative. We must lose focus when we

encounter a shift since it changes to a polarity which cannot be under focus in the given position.

$$\frac{\Omega \Vdash A^-}{\Omega \Vdash [\downarrow A^-]}\downarrow R \qquad \frac{\Omega_L\ A^+\ \Omega_R \Vdash C}{\Omega_L\ [\uparrow A^+]\ \Omega_R \Vdash C}\uparrow L$$

A few rules deserve special mention. $\mathbf{1}$ is positive, and therefore has following two rules.

$$\frac{}{\cdot \Vdash [\mathbf{1}]}\ \mathbf{1}R \qquad \frac{\overline{\Omega}_L\ \overline{\Omega_R} \Vdash \overline{C}}{\overline{\Omega}_L\ \mathbf{1}\ \overline{\Omega}_R \Vdash \overline{C}}\ \mathbf{1}L$$

Note that an attempt to prove $\Omega \Vdash [\mathbf{1}]$ simply fails if Ω is not empty.

Next we consider identity, which we a priori restricted to atomic propositions. Atoms may be either positive or negative, which means they can be in focus on the right or on the left respectively. From these considerations, we obtain:

$$\frac{}{p^+ \Vdash [p^+]}\ \mathsf{id}^+ \qquad \frac{}{[p^-] \Vdash p^-}\ \mathsf{id}^-$$

A remarkable property is that a proof attempt that is focused on a positive atom p^+ will simply fail unless the collection of antecedents consists of exactly p^-. Similarly, a proof focused on p^- will fail unless it is the only antecedent and the succedent is exactly p^-.

We also note that when applying rules like $\bullet R$ and $\backslash L$ one has to decide how to split the antecedents between the two premises. We will introduce a general mechanism called *resource management* for reducing this form of nondeterminism in a future lecture.

3 Example: Parsing Revisited

We will see that chaining can be remarkably restrictive when we consider how to perform proof search. Our example comes from the Lambek calculus, where proof search corresponds to parsing.

We will try to parse *Alice likes Bob here*. Recall from Section 5 of Lecture 1 that parsing requires us to prove s, which represents a sentence.

$$\begin{array}{cccccc} \mathit{Alice} & \mathit{likes} & \mathit{Bob} & \mathit{here} & & \\ : & : & : & : & & \\ n & n \setminus (s \,/\, n) & n & s \setminus s & \vdash ? : s \end{array}$$

We could start at the beginning of the sentence to combine *Alice* with *likes* or at the end of the sentence to reduce to problem of parsing the whole sentence to parsing *Alice likes Bob* as a sentence. Let's see how these two first steps work out without chaining.

$$\dfrac{\dfrac{}{n \vdash n}\,\mathsf{id} \qquad \begin{array}{c}\textit{(Alice likes) Bob here}\\ \vdots\\ (s \,/\, n)\; n\; (s \setminus s) \vdash s\end{array}}{n\; (n \setminus (s \,/\, n))\; n\; (s \setminus s) \vdash s}\setminus L \quad \text{or} \quad \dfrac{\begin{array}{c}\textit{Alice likes Bob}\\ \vdots\\ n\; (n \setminus (s \,/\, n))\; n \vdash s\end{array} \qquad \dfrac{}{s \vdash s}\,\mathsf{id}}{n\; (n \setminus (s \,/\, n))\; n\; (s \setminus s) \vdash s}\setminus L$$

With chaining, we can restrict the proof search such that only one of these will be possible.

To start with, we have to polarize the proposition. We pursue two options: one is where every atomic proposition is positive, and one where every atomic proposition is negative. We can also hedge our bets and make some positive and some negative (see Exercise 2).

First, we label all atoms as positive and insert the minimal number of shifts to obtain a properly polarized proposition.

$$n^+\; (n^+ \setminus (\uparrow s^+ \,/\, n^+))\; n^+\; (s^+ \setminus \uparrow s^+) \Vdash s^+$$

At this point we can not, for example, focus on $s^+ \setminus \uparrow s^+$. If we try:

$$\dfrac{\begin{array}{cc}\text{fails: no rule applicable} & \vdots\\ n^+\; (n^+ \setminus (\uparrow s^+ \,/\, n^+))\; n^+ \Vdash [s^+] & [\uparrow s^+] \Vdash s^+\end{array}}{\dfrac{n^+\; (n^+ \setminus (\uparrow s^+ \,/\, n^+))\; n^+\; [s^+ \setminus \uparrow s^+] \Vdash s^+}{n^+\; (n^+ \setminus (\uparrow s^+ \,/\, n^+))\; n^+\; (s^+ \setminus \uparrow s^+) \Vdash s^+}\,\mathsf{focus}^-}$$

In the first premise, no rule is applicable since the atom s^+ is in focus, but the antecedent is not just s^+. The second premise would actually be provable after one more forced step.

We can try each possibility, but they all fail immediately, leaving only $n^+ \setminus (\uparrow s^+ \,/\, n^+)$. We show the full phase of focusing, until we close each branch or no proposition is in focus any longer. You should verify, that

these are the only possibilities.

$$\dfrac{\dfrac{}{n^+ \Vdash [n^+]}\,\mathsf{id}^+ \quad \dfrac{\dfrac{\dfrac{\vdots}{s^+\ (s^+ \setminus \uparrow s^+) \Vdash s^+}}{[\uparrow s^+]\ (s^+ \setminus \uparrow s^+) \Vdash s^+}\uparrow L}{[\uparrow s^+ \,/\, n^+]\ n^+\ (s^+ \setminus \uparrow s^+) \Vdash s^+}\,/L}{\dfrac{n^+\ [n^+ \setminus (\uparrow s^+ \,/\, n^+)]\ n^+\ (s^+ \setminus \uparrow s^+) \Vdash s^+}{n^+\ (n^+ \setminus (\uparrow s^+ \,/\, n^+))\ n^+\ (s^+ \setminus \uparrow s^+) \Vdash s^+}\,\mathsf{focus}^-}\setminus L$$

At this point we can once again only focus on one antecedent, namely $s^+ \setminus \uparrow s^+$.

$$\dfrac{\dfrac{}{s^+ \Vdash [s^+]}\,\mathsf{id}^+ \quad \dfrac{\dfrac{\vdots}{s^+ \Vdash s^+}}{[\uparrow s^+] \Vdash s^+}\uparrow L}{s^+\ (s^+ \setminus \uparrow s^+) \Vdash s^+}\,\mathsf{focus}^-$$

We can now complete the proof by finally focusing on s^+ on the right.

$$\dfrac{\dfrac{}{s^+ \Vdash [s^+]}\,\mathsf{id}^+ \quad \dfrac{\dfrac{\dfrac{}{s^+ \Vdash [s^+]}\,\mathsf{id}^+}{s^+ \Vdash s^+}\,\mathsf{focus}^+}{[\uparrow s^+] \Vdash s^+}\uparrow L}{s^+\ (s^+ \setminus \uparrow s^+) \Vdash s^+}\,\mathsf{focus}^-$$

Alternatively to this whole proof, we now make all atomic propositions *negative* and insert the minimum number of shifts to the result is well-polarized.

$$n^-\ (\downarrow n^- \setminus (s^- \,/\, \downarrow n^-))\ n^-\ (\downarrow s^- \setminus s^-) \Vdash s^-$$

As you should verify from the rules, the only possibility here is to focus on $\downarrow s^- \setminus s^-$; all other propositions we either cannot focus on at all (like s^- on

the right), or will fail quickly without the focusing phase completing.

$$\dfrac{\dfrac{\dfrac{\vdots}{n^-\ (\downarrow n^- \setminus (s^- \,/\, \downarrow n^-))\ n^- \Vdash s^-}}{n^-\ (\downarrow n^- \setminus (s^- \,/\, \downarrow n^-))\ n^- \Vdash [\downarrow s^-]}\downarrow R \quad \dfrac{}{[s^-] \Vdash s^-}\,\mathsf{id}^-}{n^-\ (\downarrow n^- \setminus (s^- \,/\, \downarrow n^-))\ n^-\ [\downarrow s^- \setminus s^-] \Vdash s^-}\setminus L$$
$$\dfrac{}{n^-\ (\downarrow n^- \setminus (s^- \,/\, \downarrow n^-))\ n^-\ (\downarrow s^- \setminus s^-) \Vdash s^-}\,\mathsf{focus}^-$$

At this point, only the focus on $\downarrow n^- \setminus (s^- \,/\, n^-)$ will succeed. The focusing phase will build this partial proof:

$$\dfrac{\dfrac{\dfrac{\vdots}{n^- \Vdash n^-}}{n^- \Vdash [\downarrow n^-]}\downarrow R \quad \dfrac{\dfrac{\dfrac{\vdots}{n^- \Vdash n^-}}{n^- \Vdash [\downarrow n^-]}\downarrow R \quad \dfrac{}{[s^-] \Vdash s^-}\,\mathsf{id}^-}{[s^- \,/\, \downarrow n^-]\ n^- \Vdash s^-}/L}{n^-\ [\downarrow n^- \setminus (s^- \,/\, \downarrow n^-)]\ n^- \Vdash s^-}\setminus L$$
$$\dfrac{}{n^-\ (\downarrow n^- \setminus (s^- \,/\, \downarrow n^-))\ n^- \Vdash s^-}\,\mathsf{focus}^-$$

The remaining subproofs can be completed in two steps by focusing on n^- in the antecedent.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{[n^-] \Vdash n^-}\,\mathsf{id}^-}{n^- \Vdash n^-}\,\mathsf{focus}^-}{n^- \Vdash [\downarrow n^-]}\downarrow R \quad \dfrac{\dfrac{\dfrac{\dfrac{}{[n^-] \Vdash n^-}\,\mathsf{id}^-}{n^- \Vdash n^-}\,\mathsf{focus}^-}{n^- \Vdash [\downarrow n^-]}\downarrow R \quad \dfrac{}{[s^-] \Vdash s^-}\,\mathsf{id}^-}{[s^- \,/\, \downarrow n^-]\ n^- \Vdash s^-}/L}{n^-\ [\downarrow n^- \setminus (s^- \,/\, \downarrow n^-)]\ n^- \Vdash s^-}\setminus L$$
$$\dfrac{}{n^-\ (\downarrow n^- \setminus (s^- \,/\, \downarrow n^-))\ n^- \Vdash s^-}\,\mathsf{focus}^-$$

4 Soundness of Chaining

The next order of business is to convince ourselves (and everyone else who cares) that the system "works correctly". This means that (in polarized ordered logic) we prove exactly the same sequents in the chaining calculus as in the without the restrictions imposed by chaining. We need to show:

Soundness: If $\Omega \Vdash C$ then $\Omega \vdash C$.

Completeness: If $\Omega \vdash C$ then $\Omega \Vdash C$.

In general, proof search procedures restrict the allowed inferences in order to cut down on the search space, so soundness is usually straightforward while completeness is difficult.

Soundness here is very easy: take a chaining proof and erase all the focusing annotations, that is, remove the brackets $[-]$. The positive and negative focusing rules then disappear, since premise and conclusion become the same sequent, and all other rules become valid rules in the polarized, unfocused sequent calculus.

Formally, we would generalize the induction hypothesis over all three judgment forms and prove the theorem by mutual induction on the given derivation.

5 Completness of Chaining

How do we approach the completeness proof? Usually, we would just try the a straightforward structural induction over the sequent proof to see how it breaks down. This might provide some hints how to complete it.

Theorem 1 (Completeness of Chaining) *If* $\Omega \vdash A$ *then* $\Omega \Vdash A$.

Proof: (Attempt) By induction over the structure of $\Omega \vdash A$ (the derivation $\mathcal{D}$). For a while, this goes well (depending on how we start).

Case:

$$\mathcal{D} = \dfrac{\dfrac{\mathcal{D}_2}{A_1^+ \;\Omega \vdash A_2^-}}{\Omega \vdash A_1^+ \setminus A_2^-}\setminus R \qquad \text{Then} \qquad \dfrac{\dfrac{\text{i.h. on } \mathcal{D}_2}{A_1^+ \;\Omega \Vdash A_2^-}}{\Omega \Vdash A_1^+ \setminus A_2^-}\setminus R$$

This pattern repeats for right rules on negative propositions and left rules on positive propositions. Clearly, our system was specifically engineered to make this possible. It breaks down, for example, in the case of a right rule on a positive proposition since the induction hypothesis will not give us anything in focus.

Case:

$$\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Omega_L \vdash A_1^+ & \Omega_R \vdash A_2^+\end{array}}{\Omega_L\ \Omega_R \vdash A_1^+ \bullet A_2^+}\ \bullet R$$

By applying the induction hypothesis we get something like

$$\begin{array}{cc}\text{i.h. on } \mathcal{D}_1 & \text{i.h. on } \mathcal{D}_2 \\ \Omega_L \Vdash A_1^+ & \Omega_R \Vdash A_2^+ \\ & \vdots \\ & \Omega_L\ \Omega_R \Vdash A_1^+ \bullet A_2^+\end{array}$$

We might try focusing on $A_1^+ \bullet A_2^+$, but this fails:

$$\dfrac{\dfrac{\begin{array}{c}\text{i.h. on } \mathcal{D}_1 \\ \Omega_L \Vdash A_1^+\end{array}}{\Omega_L \Vdash [A_1^+]}\ ?? \quad \dfrac{\begin{array}{c}\text{i.h. on } \mathcal{D}_2 \\ \Omega_R \Vdash A_2^+\end{array}}{\Omega_L \Vdash [A_2^+]}\ ??}{\Omega_L\ \Omega_R \Vdash [A_1^+ \bullet A_2^+]}\ \bullet R}{\Omega_L\ \Omega_R \Vdash A_1^+ \bullet A_2^+}\ \mathsf{focus}^+$$

The problem here is that it is *not* the case that, in general, $\Omega \Vdash A^+$ implies $\Omega \Vdash [A^+]$! As a simple counterexample, consider $\mathbf{1} \Vdash \mathbf{1}$: we have to apply $\mathbf{1}L$ before we can focus on $\mathbf{1}$ on the right. No amount of generalization of the induction hypothesis will make a counterexample go away. However, there is a different way forward: we can close the gap using cut and identity for the chaining calculus! We may have some hope that these will be provable.

$$\dfrac{\begin{array}{c}\text{i.h. on } \mathcal{D}_2 \\ \Omega_R \Vdash A_2^+\end{array} \qquad \dfrac{\dfrac{\begin{array}{c}\text{i.h. on } \mathcal{D}_1 \\ \Omega_L \Vdash A_1^+\end{array} \qquad \dfrac{\dfrac{}{A_1^+ \Vdash [A_1^+]}\ \mathsf{id}^+ \quad \dfrac{}{A_2^+ \Vdash [A_2^+]}\ \mathsf{id}^+}{A_1^+\ A_2^+ \Vdash [A_1^+ \bullet A_2^+]}\ \bullet R}{A_1^+\ A_2^+ \Vdash A_1^+ \bullet A_2^+}\ \mathsf{focus}^+}{\Omega_L\ A_2^+ \Vdash A_1^+ \bullet A_2^+}\ \mathsf{cut}_{A_1^+}}{\Omega_L\ \Omega_R \Vdash A_1^+ \bullet A_2^+}\ \mathsf{cut}_{A_2^+}$$

All other cases will follow a similar pattern, so if cut and identity are admissible in the chaining calculus, then it will be complete. □

It takes tremendous experience to find this particular elegant proof. Andreoli's original proof and many thereafter (for example, Howe [How98]) were much more complicated and less scalable. The idea to use admissibility of cut and identity in this manner originates with Chaudhuri [Cha06] for linear logic. Some remaining infelicities with identity expansion were finally solved by Simmons for ordered logic [Sim12] and intuitionistic structural logic [Sim14].

6 Admissibility of Identity in the Chaining Calculus

In the next lecture we will sketch the admissibility of cut in the chaining calculus; here we show admissibility of identity.

As usual, admissibility of identity follows by induction on the structure of the the proposition. We need three forms.

Theorem 2 (Admissibility of Identity for Chaining) *The following are admissible:*

$$\frac{}{[A^-] \Vdash A^-}\ \mathsf{id}^-_A \qquad \frac{}{A^+ \Vdash [A^+]}\ \mathsf{id}^+_A \qquad \frac{}{A \Vdash A}\ \mathsf{id}_A$$

Proof: By induction on the structure of A, where id_A can call on the induction hypothesis for id^-_A or id^+_A, depending on the polarity of A. For example:

$$\dfrac{\dfrac{\begin{array}{cc}\text{i.h. on } A_2^+ & \text{i.h. on } A_1^- \\ A_2^+ \vdash [A_2^+] & [A_1^-] \vdash A_1^-\end{array}}{[A_1^- \,/\, A_2^+]\ A_2^+ \vdash A_1^-}\ /L}{[A_1^- \,/\, A_2^+] \vdash A_1^- \,/\, A_2^+}\ /R$$

□

Exercises

Exercise 1 We might drop the focus$^-$ and focus$^+$ rules if instead the $\uparrow R$ and $\downarrow L$ rule put the positive or negative proposition, respectively, in focus. Try to discover and perhaps prove if this would yield an equivalent calculus. If yes, discuss its merits and demerits.

Exercise 2 Investigate how the set of proofs are restricted in the parsing example from Section 3 if

1. n is negative and s is positive, and
2. n is positive and s is negative.

Exercise 3 In the completeness proof for chaining, show the cases for

1. $\backslash L$,
2. $\mathbf{1}R$,
3. id$^-$,
4. $\uparrow R$, and
5. $\uparrow L$.

You should assume that identity and cut are admissible in the chaining calculus.

References

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):197–347, 1992.

[Cha06] Kaustuv Chaudhuri. *The Focused Inverse Method for Linear Logic*. PhD thesis, Carnegie Mellon University, December 2006. Available as technical report CMU-CS-06-162.

[Fre79] Gottlob Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Verlag von Louis Nebert, 1879. English translation *Begriffsschrift, a formula language, modeled upon that of arithmatic, for pure thought* in J. van Heijenoort, editor, *From Frege to Gödel; A Source Book in Mathematical Logic, 1879–1931*, pp. 1–82, Harvard University Press, 1967.

[Gen35] 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.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Hil05] David Hilbert. Über die grundlagen der logik und der arithmetik. In *Verhandlungen des 3. Internationalen Mathematiker-Kongresses, Heidelberg, August 1904*, pages 174–185, Leipzig, 1905. A. Krazer.

[How69] 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.

[How98] Jacob M. Howe. *Proof Search Issues in Some Non-Classical Logics*. PhD thesis, University of St. Andrews, Scotland, 1998.

[Sim12] Robert J. Simmons. *Substructural Logical Specifications*. PhD thesis, Carnegie Mellon University, November 2012. Available as Technical Report CMU-CS-12-142.

[Sim14] Robert J. Simmons. Structural focalization. *ACM Transactions on Computational Logic*, 15(3):21:1–21:33, 2014.

[Zei09] Noam Zeilberger. *The Logical Basis of Evaluation Order and Pattern-Matching*. PhD thesis, Department of Computer Science, Carnegie Mellon University, May 2009. Available as Technical Report CMU-CS-09-122.

Lecture Notes on Focusing

15-816: Substructural Logics
Frank Pfenning

Lecture 18
November 1, 2016

As discussed in the last lecture, focusing [And92] is one of the major achievements of proof theory. It decomposes into inversion and chaining, which we presented last time. In this lecture we first complete the development of chaining by sketching its proof of cut elimination and then we introduce full focusing.

1 Summary of Chained Inference

We summarize chaining from last lecture. While we present it here for ordered logic, it applies as well to other structural and substructural logics; we will see examples in the next lecture.

$$\begin{array}{lrcl} \text{Negative} & A^-, B^- & ::= & p^- \mid A^+ \setminus B^- \mid B^- \,/\, A^+ \mid A^- \,\&\, B^- \mid \uparrow A^+ \\ \text{Positive} & A^+, B^+ & ::= & p^+ \mid A^+ \bullet B^+ \mid A^+ \circ B^+ \mid \mathbf{1} \mid A^+ \oplus B^+ \mid \downarrow A^- \end{array}$$

There are three judgments

$$\begin{array}{rcl} \Omega & \Vdash & A \\ \Omega & \Vdash & [C^+] \\ \Omega_L\ [A^-]\ \Omega_R & \Vdash & C \end{array}$$

We abbreviate

$$\begin{array}{rcl} \overline{\Omega} & ::= & \Omega \mid \Omega_L\ [A^-]\ \Omega_R \\ \overline{C} & ::= & C \mid [C^+] \end{array}$$

and globally presuppose for any judgment $\overline{\Omega} \Vdash \overline{C}$:

There is at most one proposition in focus in any sequent.

We provide the rules for a selection of the connectives.

$$\frac{}{p^+ \Vdash [p^+]}\ \mathsf{id}^+ \qquad \frac{}{[p^-] \Vdash p^-}\ \mathsf{id}^-$$

$$\frac{\Omega \Vdash [A^+]}{\Omega \Vdash A^+}\ \mathsf{focus}^+ \qquad \frac{\Omega_L\ [A^-]\ \Omega_R \Vdash C}{\Omega_L\ A^-\ \Omega_R \Vdash C}\ \mathsf{focus}^-$$

$$\frac{\overline{\Omega} \Vdash A^+}{\overline{\Omega} \Vdash \uparrow A^+}\ \uparrow R \qquad \frac{\Omega_L\ A^+\ \Omega_R \Vdash C}{\Omega_L\ [\uparrow A^+]\ \Omega_R \Vdash C}\ \uparrow L$$

$$\frac{\Omega \Vdash A^-}{\Omega \Vdash [\downarrow A^-]}\ \downarrow R \qquad \frac{\overline{\Omega}_L\ A^-\ \overline{\Omega}_R \Vdash \overline{C}}{\overline{\Omega}_L\ (\downarrow A^-)\ \overline{\Omega}_R \Vdash \overline{C}}\ \downarrow L$$

$$\frac{A^+\ \overline{\Omega} \Vdash B^-}{\overline{\Omega} \Vdash A^+ \setminus B^-}\ \setminus R \qquad \frac{\Omega \Vdash [A^+] \quad \Omega_L\ [B^-]\ \Omega_R \Vdash C}{\Omega_L\ \Omega\ [A^+ \setminus B^-]\ \Omega_R \Vdash C}\ \setminus L$$

$$\frac{\Omega_L \Vdash [A^+] \quad \Omega_R \Vdash [B^+]}{\Omega_L\ \Omega_R \Vdash [A^+ \bullet B^+]}\ \bullet R \qquad \frac{\overline{\Omega}_L\ A\ B\ \overline{\Omega}_R \Vdash \overline{C}}{\overline{\Omega}_L\ (A \bullet B)\ \overline{\Omega}_R \Vdash \overline{C}}\ \bullet L$$

$$\frac{}{\cdot \Vdash [\mathbf{1}]}\ \mathbf{1}R \qquad \frac{\overline{\Omega}_L\ \overline{\Omega_R} \Vdash \overline{C}}{\overline{\Omega}_L\ \mathbf{1}\ \overline{\Omega}_R \Vdash \overline{C}}\ \mathbf{1}L$$

$$\frac{\overline{\Omega} \Vdash A \quad \overline{\Omega} \Vdash B}{\overline{\Omega} \Vdash A \,\&\, B}\ \&R \qquad \frac{\Omega_L\ [A^-]\ \Omega_R \Vdash C}{\Omega_L\ [A^- \,\&\, B^-]\ \Omega_R \Vdash C}\ \&L_1 \qquad \frac{\Omega_L\ [B^-]\ \Omega_R \Vdash C}{\Omega_L\ [A^- \,\&\, B^-]\ \Omega_R \Vdash C}\ \&L_2$$

$$\frac{\Omega \Vdash [A^+]}{\Omega \Vdash [A^+ \oplus B^+]}\ \oplus R_1 \qquad \frac{\Omega \Vdash [B^+]}{\Omega \Vdash [A^+ \oplus B^+]}\ \oplus R_2 \qquad \frac{\overline{\Omega}_L\ A^+\ \overline{\Omega}_R \Vdash \overline{C} \quad \overline{\Omega}_L\ B^+\ \overline{\Omega}_R \Vdash \overline{C}}{\overline{\Omega}_L\ (A^+ \oplus B^+)\ \overline{\Omega}_R \Vdash \overline{C}}\ \oplus L$$

2 Admissibility of Cut in Chained Inference

Neither the chaining calculus nor the upcoming full focusing calculus allow the rule of cut. It would violate the basic goal of restricting proof search. However, as we have seen in the last lecture, admissibility of cut (together with admissibility of identity), is the key to completeness of focusing. Keeping in mind our central presupposition that no more than one

proposition can be focus, we obtain the following versions of cut. Note that in cut^*_A at most one of the overlined antecedents or succedents can contain a proposition in focus.

$$\frac{\overline{\Omega} \Vdash A \quad \overline{\Omega}_L\ A\ \overline{\Omega}_R \Vdash \overline{C}}{\overline{\Omega}_L\ \overline{\Omega}\ \overline{\Omega}_R \Vdash \overline{C}}\ \mathsf{cut}^*_A$$

$$\frac{\Omega \Vdash [A^+] \quad \overline{\Omega}_L\ A^+\ \overline{\Omega}_R \Vdash \overline{C}}{\overline{\Omega}_L\ \Omega\ \overline{\Omega}_R \Vdash \overline{C}}\ \mathsf{cut}^+_A \qquad \frac{\overline{\Omega} \Vdash A^- \quad \Omega_L\ [A^-]\ \Omega_R \Vdash C}{\Omega_L\ \overline{\Omega}\ \Omega_R \Vdash C}\ \mathsf{cut}^-_A$$

Theorem 1 (Admissibility of Cut in Chained Deduction)
The rules cut^*_A, cut^+_A *and* cut^-_A *are all admissible.*[1]

Proof: By nested induction, first on the structure of A and second simultaneously on the structure of the two given deductions.

The only significant change compared to the usual proof of admissibility of cut is that we restrict the forms of commuting reductions for cut^*_A to preserve the invariant in the conclusion. □

3 Full Focusing

We obtain the full focusing system by forcing all possible inversion steps to be completed before allowing focus. This also means while a proposition is in focus, inferences can only be applied to the focus formula and no other rules are applicable.

This can be specified in two ways. One is to just restrict focus^+ and focus^- so that no inversion rule can apply. This means inversion steps can be applied arbitrarily, which entails some nondeterminism because there may be multiple invertible propositions in the antecedents or succedent. However, this is *don't-care nondeterminism* since the remaining subgoals after all inversion rules have been applied will always be the same, a property called *confluence*.

Alternatively (as advocated, for example, by Simmons [Sim14]) we can write the rules to force a particular order of application of these rules, say, left-to-right. This leads to a simpler proof of its completeness via admissibility of cut, since one does not have to prove confluence.

For the purpose of these notes, we use the don't-care nondeterministic version since it has less syntactic overhead. The following rules capture the

[1] As of the time I am writing up these notes, not all of these have been checked carefully.

same connectives as before. We say $\Omega \Vdash C$ is *stable* if Ω consists only of negative propositions or positive atoms and C is either a positive proposition or negative atom. A focusing sequent is stable exactly if no inversion rule applies. In the rules below, inversion rules no longer allow other propositions to be in focus.

$$\frac{}{p^+ \Vdash [p^+]}\ \mathsf{id}^+ \qquad \frac{}{[p^-] \Vdash p^-}\ \mathsf{id}^-$$

$$\frac{(\Omega \Vdash A^+)\ \mathsf{stable} \quad \Omega \Vdash [A^+]}{\Omega \Vdash A^+}\ \mathsf{focus}^+$$

$$\frac{(\Omega_L\ A^-\ \Omega_R \Vdash C)\ \mathsf{stable} \quad \Omega_L\ [A^-]\ \Omega_R \Vdash C}{\Omega_L\ A^-\ \Omega_R \Vdash C}\ \mathsf{focus}^-$$

$$\frac{\Omega \Vdash A^+}{\Omega \Vdash \uparrow A^+}\ \uparrow R \qquad \frac{\Omega_L\ A^+\ \Omega_R \Vdash C}{\Omega_L\ [\uparrow A^+]\ \Omega_R \Vdash C}\ \uparrow L$$

$$\frac{\Omega \Vdash A^-}{\Omega \Vdash [\downarrow A^-]}\ \downarrow R \qquad \frac{\Omega_L\ A^-\ \Omega_R \Vdash C}{\Omega_L\ (\downarrow A^-)\ \Omega_R \Vdash C}\ \downarrow L$$

$$\frac{A^+\ \Omega \Vdash B^-}{\Omega \Vdash A^+ \setminus B^-}\ \setminus R \qquad \frac{\Omega \Vdash [A^+] \quad \Omega_L\ [B^-]\ \Omega_R \Vdash C}{\Omega_L\ \Omega\ [A^+ \setminus B^-]\ \Omega_R \Vdash C}\ \setminus L$$

$$\frac{\Omega_L \Vdash [A^+] \quad \Omega_R \Vdash [B^+]}{\Omega_L\ \Omega_R \Vdash [A^+ \bullet B^+]}\ \bullet R \qquad \frac{\Omega_L\ A\ B\ \Omega_R \Vdash C}{\Omega_L\ (A \bullet B)\ \Omega_R \Vdash C}\ \bullet L$$

$$\frac{}{\cdot \Vdash [\mathbf{1}]}\ \mathbf{1}R \qquad \frac{\Omega_L\ \Omega_R \Vdash C}{\Omega_L\ \mathbf{1}\ \Omega_R \Vdash C}\ \mathbf{1}L$$

$$\frac{\Omega \Vdash A \quad \Omega \Vdash B}{\Omega \Vdash A \,\&\, B}\ \&R \qquad \frac{\Omega_L\ [A^-]\ \Omega_R \Vdash C}{\Omega_L\ [A^- \,\&\, B^-]\ \Omega_R \Vdash C}\ \&L_1 \qquad \frac{\Omega_L\ [B^-]\ \Omega_R \Vdash C}{\Omega_L\ [A^- \,\&\, B^-]\ \Omega_R \Vdash C}\ \&L_2$$

$$\frac{\Omega \Vdash [A^+]}{\Omega \Vdash [A^+ \oplus B^+]}\ \oplus R_1 \qquad \frac{\Omega \Vdash [B^+]}{\Omega \Vdash [A^+ \oplus B^+]}\ \oplus R_2 \qquad \frac{\Omega_L\ A^+\ \Omega_R \Vdash C \quad \Omega_L\ B^+\ \Omega_R \Vdash C}{\Omega_L\ (A^+ \oplus B^+)\ \Omega_R \Vdash C}\ \oplus L$$

We do not present here the proof of soundness, completeness or the admissibility of cut and identity for this calculus, which can be found in [Sim12, Sim14] for closely related calculi.

4 Application: Focused Parsing

As an example of focusing, we reconsider the example of parsing *Alice likes Bob here* from Lecture 17, Section 3.

$$\begin{array}{cccccc} \textit{Alice} & & \textit{likes} & \textit{Bob} & \textit{here} & \\ : & & : & : & : & \\ n & n \setminus (s \,/\, n) & & n & s \setminus s & \vdash ? : s \end{array}$$

We have to decide on a polarity for the atoms n and s. We start by making both of them *positive*. This gives us the following theorem proving problem:

$$n^+ \quad (n^+ \setminus (\uparrow s^+ \,/\, n^+)) \quad n^+ \quad (s^+ \setminus \uparrow s^+) \Vdash s^+$$

Since we can focus only on negative antecedents and positive succedents, we can only focus on $n^+ \setminus (\uparrow s^+ \,/\, n^+)$, $s^+ \setminus \uparrow s^+$, or the succedent s^+. Focusing on the succedent will immediately fail since the antecedents are not equal to just s^+. We can try to focus on $s^+ \setminus \uparrow s^+$, but this will fail for a similar reason:

$$\dfrac{\overset{\text{no rule applicable}}{n^+ \; (n^+ \setminus (\uparrow s^+ \,/\, n^+)) \; n^+ \Vdash [s^+] \quad [\uparrow s^+] \Vdash s^+}}{n^+ \; (n^+ \setminus (\uparrow s^+ \,/\, n^+)) \; n^+ \; [s^+ \setminus \uparrow s^+] \Vdash s^+} \setminus L$$

So it only remains to focus on the transitive verb *likes*. Note in the focusing phase, all the steps are forced once we have decided to focus, leaving only one subgoal.

$$\dfrac{\dfrac{}{n^+ \Vdash [n^+]}\,\mathsf{id}^+ \quad \dfrac{\dfrac{\vdots}{s^+ \; (s^+ \setminus \uparrow s^+) \Vdash s^+}}{[\uparrow s^+] \; (s^+ \setminus \uparrow s^+) \Vdash s^+} \uparrow L}{[\uparrow s^+ \,/\, n^+] \; n^+ \; (s^+ \setminus \uparrow s^+) \Vdash s^+} \,/L$$
$$\dfrac{\dfrac{}{n^+ \Vdash [n^+]}\,\mathsf{id}^+ \quad [\uparrow s^+ \,/\, n^+] \; n^+ \; (s^+ \setminus \uparrow s^+) \Vdash s^+}{n^+ \; [n^+ \setminus (\uparrow s^+ \,/\, n^+)] \; n^+ \; (s^+ \setminus \uparrow s^+) \Vdash s^+} \setminus L$$

The remaining subgoal can be proved only by focusing on $s^+ \setminus \uparrow s^+$:

$$\dfrac{\dfrac{}{s^+ \Vdash [s^+]}\,\mathsf{id}^+ \quad \dfrac{\dfrac{\vdots}{s^+ \Vdash s^+}}{[\uparrow s^+] \Vdash s^+} \uparrow R}{s^+ \; [s^+ \setminus \uparrow s^+] \Vdash s^+} \setminus L$$

The final subgoal can now be proved by focusing on the succedent s^+:

$$\frac{}{s^+ \Vdash [s^+]}\ \mathsf{id}^+$$

Note that in all of these steps there was no choice: in every stable sequent, there was only on possibility to focus. In essence, focusing has reduced the number of proofs to just one, which is a highly significant restriction compared the nondeterminism present if we proceed in small steps.

If we mark all atoms as negative, there is a small choice right at the beginning, because we could focus on either $\downarrow n^- \setminus (s^- \,/\, \downarrow n^-)$ or $\downarrow s^- \setminus s^-$. Only the latter will succeed, so we show that proof.

$$\frac{\dfrac{\begin{array}{c}\vdots\\ n^-\ (\downarrow n^- \setminus (s^- \,/\, \downarrow n^-))\ n^- \Vdash s^-\end{array}}{n^-\ (\downarrow n^- \setminus (s^- \,/\, \downarrow n^-))\ n^- \Vdash [\downarrow s^-]}\downarrow R \qquad \frac{}{[s^-] \Vdash s^-}\ \mathsf{id}^-}{n^-\ (\downarrow n^- \setminus (s^- \,/\, \downarrow n^-))\ n^- [\downarrow s^- \setminus s^-] \Vdash s^-}\setminus L$$

Only one focus is possible now.

$$\frac{\dfrac{\begin{array}{c}\vdots\\ n^- \Vdash n^-\end{array}}{n^- \Vdash [\downarrow n^-]}\downarrow R \qquad \dfrac{\dfrac{\begin{array}{c}\vdots\\ n^- \Vdash n^-\end{array}}{n^- \Vdash [\downarrow n^-]}\downarrow R \qquad \frac{}{[s^-] \Vdash s^-}\ \mathsf{id}^-}{[s^- \,/\, \downarrow n^-]\ n^- \Vdash s^-}/L}{n^-\ [\downarrow n^- \setminus (s^- \,/\, \downarrow n^-)]\ n^- \Vdash s^-}\setminus L$$

The remaining (identical) subgoals following by focusing on the left.

$$\frac{}{[n^-] \Vdash n^-}\ \mathsf{id}^-$$

5 Summary

Focusing [And92] is a tremendous simplification and restructuring of the search space provided by the cut-free sequent calculus. Instead of having to make individual decisions on inference rules, which are really tiny steps, focusing allows big steps of inference. It is also widely applicable,

for example, to ordered logic [Pol01], linear, intuitionistic and classical logics [LM09], with very elegant proofs of completeness based on cut and identity [Cha06, Sim12, Sim14]. Indeed, as we will see in the next lecture, deduction is so controlled that it can be seen as the foundation for *logic programming*, where computation is modeled as inference.

6 Synthetic Inference Rules

Intuitively, focusing proofs of arbitrary sequents start by breaking down all invertible connectives to obtain a stable sequent. From a stable sequent, we then focus on a particular proposition which will be broken down in a chained phase of inference. Once we have lost focus (in the $\downarrow R$ and $\uparrow L$ rules) when the enter a phase of inversion until we reach another stable sequent along each branch of the proof that has not yet been completed. The idea behind *synthetic rules of inference* [And02] is to replace the general rules of inference entirely by specialized ones that implement this strategy.

Let's see how this plays out in the parsing example, starting with the positive polarization.

$$n^+ \; (n^+ \setminus (\uparrow s^+ \,/\, n^+)) \; n^+ \; (s^+ \setminus \uparrow s^+) \Vdash s^+$$

The subformulas we might focus on in a potential proof of this sequent are antecedents $\mathsf{tv}^- = n^+ \setminus (\uparrow s^+ \,/\, n^+)$, $\mathsf{adv}^- = s^+ \setminus \uparrow s^+$ and the succedent s^+. Let's see what would happen if we focused on each of these propositions in a general (stable) sequent. First, focusing on tv^-.

$$\dfrac{(\Omega_1 = \Omega_{11}\; \Omega_{12}) \quad \overset{\vdots}{\Omega_{12} \Vdash [n^+]} \quad \overset{\vdots}{\Omega_{11}\; [\uparrow s^+ \,/\, n^+]\; \Omega_2 \Vdash C}}{\Omega_1\; [n^+ \setminus (\uparrow s^+ \,/\, n^+)]\; \Omega_2 \Vdash C}\; \setminus R$$

Now the first subgoal $\Omega_{12} \Vdash [n^+]$ can only succeed if $\Omega_{12} = n^+$, so we can fill that in as a consequence of focusing.

$$\dfrac{(\Omega_1 = \Omega_{11}\; \Omega_{12}) \quad \dfrac{(\Omega_{12} = n^+)}{\Omega_{12} \Vdash [n^+]}\,\mathsf{id}^+ \quad \overset{\vdots}{\Omega_{11}\; [\uparrow s^+ \,/\, n^+]\; \Omega_2 \Vdash C}}{\Omega_1\; [n^+ \setminus (\uparrow s^+ \,/\, n^+)]\; \Omega_2 \Vdash C}\; \setminus R$$

In the second subgoal we continue the focusing phase, this time splitting up Ω_2.

$$\dfrac{(\Omega_1 = \Omega_{11}\ \Omega_{12}) \quad \dfrac{(\Omega_{12} = n^+)}{\Omega_{12} \Vdash [n^+]}\ \mathsf{id}^+ \quad \dfrac{(\Omega_2 = \Omega_{21}\ \Omega_{22}) \quad \vdots\ \Omega_{21} \Vdash [n^+] \quad \vdots\ \Omega_{11}\ [\uparrow s^+]\ \Omega_{22} \Vdash C}{\Omega_{11}\ [\uparrow s^+ \,/\, n^+]\ \Omega_2 \Vdash C}\ /L}{\Omega_1\ [n^+ \setminus (\uparrow s^+ \,/\, n^+)]\ \Omega_2 \Vdash C}\ \setminus L$$

In the first remaining subgoal we succeed, but only if $\Omega_{21} = n^+$; in the second remaining subgoal we lose focus.

$$\dfrac{(\Omega_1 = \Omega_{11}\ \Omega_{12}) \quad \dfrac{(\Omega_{12} = n^+)}{\Omega_{12} \Vdash [n^+]}\ \mathsf{id}^+ \quad \dfrac{(\Omega_2 = \Omega_{21}\ \Omega_{22}) \quad \dfrac{(\Omega_{21} = n^+)}{\Omega_{21} \Vdash [n^+]}\ \mathsf{id}^+ \quad \dfrac{\Omega_{11}\ s^+\ \Omega_{22} \Vdash C}{\Omega_{11}\ [\uparrow s^+]\ \Omega_{22} \Vdash C}\ \uparrow R}{\Omega_{11}\ [\uparrow s^+ \,/\, n^+]\ \Omega_2 \Vdash C}\ /L}{\Omega_1\ [n^+ \setminus (\uparrow s^+ \,/\, n^+)]\ \Omega_2 \Vdash C}\ \setminus L$$

Summarizing all this into one synthetic rule, we obtain for $\mathsf{tv}^- = n^+ \setminus (\uparrow s^+ \,/\, n^+)$

$$\dfrac{\Omega_{11}\ s^+\ \Omega_{22} \Vdash C}{\Omega_{11}\ n^+\ \mathsf{tv}^-\ n^+\ \Omega_{22} \Vdash C}\ \mathsf{tv}$$

Similarly, if we focus on $\mathsf{adv}^- = s^+ \setminus \uparrow s^+$ we obtain

$$\dfrac{(\Omega_1 = \Omega_{11}\ \Omega_{12}) \quad \dfrac{\Omega_{12} = s^+}{\Omega_{12} \Vdash [s^+]}\ \mathsf{id}^+ \quad \dfrac{\Omega_{11}\ s^+\ \Omega_2 \Vdash C}{\Omega_{11}\ [\uparrow s^+]\ \Omega_2 \Vdash C}\ \uparrow R}{\Omega_1\ [s^+ \setminus \uparrow s^+]\ \Omega_2 \Vdash C}\ \setminus L$$

which we can summarize as

$$\dfrac{\Omega_{11}\ s^+\ \Omega_2 \Vdash C}{\Omega_{11}\ s^+\ \mathsf{adv}^-\ \Omega_2 \Vdash C}\ \mathsf{adv}$$

Finally, we can focus on s^+ in the succedent:

$$\dfrac{(\Omega = s^+)}{\Omega \Vdash [s^+]}\ \mathsf{id}^+$$

which we summarize as

$$\dfrac{}{s^+ \Vdash s^+}\ s$$

Writing all three rules down together:

$$\frac{\Omega_{11}\ s^+\ \Omega_{22} \Vdash C}{\Omega_{11}\ n^+\ \mathsf{tv}^-\ n^+\ \Omega_{22} \Vdash C}\ \mathsf{tv}^-$$

$$\frac{\Omega_{11}\ s^+\ \Omega_2 \Vdash C}{\Omega_{11}\ s^+\ \mathsf{adv}^-\ \Omega_2 \Vdash C}\ \mathsf{adv}^-$$

$$\frac{}{s^+ \Vdash s^+}\ \mathsf{s}^+$$

The remarkable property is that *any focused proof* of

$$n^+\ (n^+ \setminus (\uparrow s^+ \,/\, n^+))\ n^+\ (s^+ \setminus \uparrow s^+) \Vdash s^+$$

or, in abbreviated form

$$n^+\ \mathsf{tv}^-\ n^+\ \mathsf{adv}^- \Vdash s^+$$

can be written with only these three derived rules of inference. This is because tv^-, adv^- and s^+ are the only propositions we could possibly focus on, and focused proofs are complete. Let's explore the qualities of this search space. Neither rules adv or s are applicable, so must start with tv.

$$\frac{\begin{array}{c}\vdots\\ s^+\ \mathsf{adv}^- \Vdash s^+\end{array}}{n^+\ \mathsf{tv}^-\ n^+\ \mathsf{adv}^- \Vdash s^+}\ \mathsf{tv}$$

At this point, only adv is applicable, which yields

$$\frac{\begin{array}{c}\vdots\\ s^+ \Vdash s^+\end{array}}{\dfrac{s^+\ \mathsf{adv}^- \Vdash s^+}{n^+\ \mathsf{tv}^-\ n^+\ \mathsf{adv}^- \Vdash s^+}\ \mathsf{tv}}\ \mathsf{adv}$$

Now, only rule s applies, completing the proof.

$$\frac{\dfrac{\dfrac{}{s^+ \Vdash s^+}\ \mathsf{s}}{s^+\ \mathsf{adv}^- \Vdash s^+}\ \mathsf{adv}}{n^+\ \mathsf{tv}^-\ n^+\ \mathsf{adv}^- \Vdash s^+}\ \mathsf{tv}$$

Note that there was no nondeterminism in this proof at all, and it proceeds in three simple steps. Compare this with the small-step proof in Section 4.

We can also assign negative polarity to all atoms and derive synthetic rules are we have determined which propositions we might focus on. Note that our definitions of tv and adv need to change.

$$\begin{array}{l}\mathsf{tv}^- = \downarrow n^- \setminus (s^- \,/\, \downarrow n^-) \\ \mathsf{adv}^- = \downarrow s^- \setminus s^-\end{array}$$

and our goal becomes

$$n^-\ \mathsf{tv}^-\ n^-\ \mathsf{adv}^- \Vdash s^-$$

We can focus on tv^-, adv^- and n^-.

$$\dfrac{(\Omega_1 = \Omega_{11}\ \Omega_{12}) \quad \dfrac{\Omega_{12} \Vdash n^-}{\Omega_{12} \Vdash [\downarrow n^-]}\downarrow R \quad \dfrac{(\Omega_2 = \Omega_{21}\ \Omega_{22}) \quad \dfrac{\Omega_{21} \Vdash n^-}{\Omega_{21} \Vdash [\downarrow n^-]}\downarrow R \quad \dfrac{(\Omega_{11} = \Omega_{22} = \cdot, C = s^-)}{\Omega_{11}\ [s^-]\ \Omega_{22} \Vdash C}\,\mathsf{id}^-}{\Omega_{11}\ [s^- \,/\, \downarrow n^-]\ \Omega_2 \Vdash C}\,/L}{\Omega_1\ [\downarrow n^- \setminus (s^- \,/\, \downarrow n^-)]\ \Omega_2 \Vdash C}\setminus L$$

Reading off the synthetic rule:

$$\dfrac{\Omega_{12} \Vdash n^- \quad \Omega_{21} \Vdash n^-}{\Omega_{12}\ \mathsf{tv}^-\ \Omega_{21} \Vdash s^-}\,\mathsf{tv}$$

Similarly, for adv^-:

$$\dfrac{(\Omega_1 = \Omega_{11}\ \Omega_{12}) \quad \dfrac{\Omega_{12} \Vdash s^-}{\Omega_{12} \Vdash [\downarrow s^-]}\downarrow R \quad \dfrac{(\Omega_{11} = \Omega_2 = \cdot, C = s^-)}{\Omega_{11}\ [s^-]\ \Omega_2 \Vdash C}\,\mathsf{id}^-}{\Omega_1\ [\downarrow s^- \setminus s^-]\ \Omega_2 \Vdash C}\setminus L$$

which yields the synthetic rule

$$\dfrac{\Omega_{12} \Vdash s^-}{\Omega_{12}\ \mathsf{adv}^- \Vdash s^-}\,\mathsf{adv}$$

Finally, focusing on n^-

$$\dfrac{(\Omega_1 = \Omega_2 = \cdot, C = n^-)}{\Omega_1\ [n^-]\ \Omega_2 \Vdash C}\,\mathsf{id}^-$$

Summarizing the synthetic rules

$$\frac{\Omega_{12} \Vdash n^- \quad \Omega_{21} \Vdash n^-}{\Omega_{12}\ \mathsf{tv}^-\ \Omega_{21} \Vdash s^-}\ \mathsf{tv} \qquad \frac{\Omega_{12} \Vdash s^-}{\Omega_{12}\ \mathsf{adv}^- \Vdash s^-}\ \mathsf{adv} \qquad \frac{}{n^- \Vdash n^-}\ \mathsf{n}$$

Reconsidering our goal

$$n^-\ \mathsf{tv}^-\ n^-\ \mathsf{adv}^- \Vdash s^-$$

two rules are applicable: tv, which fails in two synthetic steps,

$$\dfrac{n^- \Vdash n^- \qquad \dfrac{\text{no rule applicable}}{n^-\ \mathsf{adv}^- \Vdash s^-}\ \mathsf{adv}}{n^-\ \mathsf{tv}^-\ n^-\ \mathsf{adv}^- \Vdash s^-}\ \mathsf{tv}$$

and adv, which succeeds:

$$\dfrac{\dfrac{}{n^- \Vdash n^-}\ \mathsf{n} \qquad \dfrac{}{n^- \Vdash n^-}\ \mathsf{n}}{\dfrac{n^-\ \mathsf{tv}^-\ n^- \Vdash s^-}{n^-\ \mathsf{tv}^-\ n^-\ \mathsf{adv}^- \Vdash s^-}\ \mathsf{adv}}\ \mathsf{tv}$$

In general, there do not appear to be clear heuristics for deciding which polarization of the atoms is better for the purpose of theorem proving [MP08, MP09]. We will see in a later lecture that bottom-up logic programming (in the style of Datalog) and top-down logic programming (in the style of Prolog) can be obtained from purely positive or purely negative atoms in a fragment of the logic [CPP08].

Exercises

Exercise 1 Show one principal, one identity, one left commutative, and one right commutative case in the proof of admissibility of cut on chained sequents.

Exercise 2 Derive synthetic rules of inference for the remaining two possible polarizations of the parsing example in Section 6 (where n is positive and s is negative, and vice versa). Characterize the resulting search space.

References

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):197–347, 1992.

[And02] Jean Marc Andreoli. Focussing proof-net construction as a middleware paradigm. In A. Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction (CADE-18)*, pages 501–516, Copenhagen, Denmark, July 2002. Springer-Verlag LNAI 2392.

[Cha06] Kaustuv Chaudhuri. *The Focused Inverse Method for Linear Logic*. PhD thesis, Carnegie Mellon University, December 2006. Available as technical report CMU-CS-06-162.

[CPP08] Kaustuv Chaudhuri, Frank Pfenning, and Greg Price. A logical characterization of forward and backward chaining in the inverse method. *Journal of Automated Reasoning*, 40(2–3):133–177, 2008. Special issue with selected papers from IJCAR 2006.

[LM09] Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, November 2009.

[MP08] Sean McLaughlin and Frank Pfenning. Imogen: Focusing the polarized inverse method for intuitionistic propositional logic. In I.Cervesato, H.Veith, and A.Voronkov, editors, *Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'08)*, pages 174–181, Doha, Qatar, November 2008. Springer LNCS 5330. System Description.

[MP09] Sean McLaughlin and Frank Pfenning. Efficient intuitionistic theorem proving with the polarized inverse method. In R.A.Schmidt, editor, *Proceedings of the 22nd International Conference on Automated Deduction (CADE-22))*, pages 230–244, Montreal, Canada, August 2009. Springer LNCS 5663.

[Pol01] Jeff Polakow. *Ordered Linear Logic and Applications*. PhD thesis, Department of Computer Science, Carnegie Mellon University, August 2001.

[Sim12] Robert J. Simmons. *Substructural Logical Specifications*. PhD thesis, Carnegie Mellon University, November 2012. Available as Technical Report CMU-CS-12-142.

[Sim14] Robert J. Simmons. Structural focalization. *ACM Transactions on Computational Logic*, 15(3):21:1–21:33, 2014.

Lecture Notes on
Substructural Operational Semantics

15-816: Substructural Logics
Frank Pfenning

Lecture 19
November 3, 2016

Throughout this course we have already used substructural logic to specify the operational semantics of our (small) programming languages. For example, we have used ordered inference to represent computation based on proof reduction in subsingleton logic and we have used linear inference to represent computation in ordered logic. Focusing provides us with the needed connection between ordered and linear inference and *propositions* in ordered and linear logic, completing the picture. This use of the inference in linear logic goes back to CLF [WCPW02, CPWW02] which was in turn inspired by Forum [Chi95, Mil96]. A systematic, taxonomic approach was advocated [Pfe04] and then explored [SP08, SP09, PS09, SP11] culminating in Simmons's dissertation [Sim12].

Before we give further applications, we need to consider how focusing applies to structural logic, and the integration of structural and substructural logics.

1 Example: Increment

Let's recall the representation of binary numbers as string of b0 and b1 and a left endmarker $ (previously written as eps). For example, the number 1011_2 would be represented by the ordered context $ b1 b0 b1 b1. Now we specify incrementing such a number by adding a new ordered proposition inc with the following three rules.

$$\frac{\mathsf{b0} \quad \mathsf{inc}}{\mathsf{b1}} \qquad \frac{\mathsf{b1} \quad \mathsf{inc}}{\mathsf{inc} \quad \mathsf{b0}} \qquad \frac{\$ \quad \mathsf{inc}}{\$ \quad \mathsf{b1}}$$

We can represent them by the following propositions, where $\mathsf{b0}^+$, $\mathsf{b1}^+$, $\$^+$, and inc^+ are all positive atoms. We assume that $\bullet$ more binding strength than $\backslash$ and $/$, so that $A \bullet B \backslash C$ stands for $(A \bullet B) \backslash C$.

$$\begin{array}{l} \mathsf{b0} \bullet \mathsf{inc} \backslash \uparrow \mathsf{b1} \\ \mathsf{b1} \bullet \mathsf{inc} \backslash \uparrow (\mathsf{inc} \bullet \mathsf{b0}) \\ \$ \bullet \mathsf{inc} \backslash \uparrow (\$ \bullet \mathsf{b1}) \end{array}$$

However, there is a fly in the ointment: the inference rules are persistent, while the propositions we may focus on are not. So the above propositions should be shifted from ordered O to structural U. We might try

$$\uparrow^{\mathsf{U}}_{\mathsf{O}} (\mathsf{b0} \bullet \mathsf{inc} \backslash \uparrow^{\mathsf{O}}_{\mathsf{O}} \mathsf{b1})$$

but this does not work, since the an up shift coerces a positive proposition to a negative one, but $(\mathsf{b0} \bullet \mathsf{inc} \backslash \uparrow^{\mathsf{O}}_{\mathsf{O}} \mathsf{b1})$ is already negative. So we could write it as

$$\uparrow^{\mathsf{U}}_{\mathsf{O}} \downarrow^{\mathsf{O}}_{\mathsf{O}} (\mathsf{b0} \bullet \mathsf{inc} \backslash \uparrow^{\mathsf{O}}_{\mathsf{O}} \mathsf{b1})$$

which is logically correct but, as we will see in Section 4, it may not have the expected focusing behavior. So we write

$$^{\mathsf{U}}_{\mathsf{O}}\Uparrow\Downarrow (\mathsf{b0} \bullet \mathsf{inc} \backslash \uparrow^{\mathsf{O}}_{\mathsf{O}} \mathsf{b1})$$

to lift a negative ordered proposition to a negative unrestricted proposition. Before we investigate the properties of $^{\mathsf{U}}_{\mathsf{O}}\Uparrow\Downarrow$, an excursion to look at focusing for structural logic more generally.

2 Focusing for Structural Logic

We start with the polarized form of structural logic. As before we keep two forms of conjunction which are logically, but not computationally equivalent.

$$\begin{array}{lcl} A^- & ::= & p^- \mid A^+ \to B^- \mid A^- \mathbin{\&} B^- \mid \uparrow A^+ \\ A^+ & ::= & p^+ \mid A^+ \times B^+ \mid \mathbf{1} \mid A^+ + B^+ \mid \downarrow A^- \end{array}$$

The key insights are the following:

1. For left inversion rules, we do not need to keep a copy of the principal proposition of the inference. That's because its components are equivalent to the proposition itself. This also means that even structural antecedents are no longer persistent.

2. For left chaining rules, the proposition in focus is also not persistent. This also should be intuitive, since we cannot have the formula and its subformula both be in focus.

We say *antecedents* Γ are stable^- if Γ consists only of negative propositions and positive atoms and *succedent* C is stable^+ if it is a positive proposition or a negative atom. We still have three judgment forms

$$\begin{array}{ll} \Gamma \Vdash A & \\ \Gamma \Vdash [A] & \text{with } \Gamma \text{ stable}^- \\ \Gamma, [A] \Vdash C & \text{with } \Gamma \text{ stable}^- \text{ and } C \text{ stable}^+ \end{array}$$

As in Lecture 18 we present a confluent focusing system with don't-care nondeterministic inversion rules. Simmons [Sim14] presents a system more suited for most implementations and also proofs of admissibility of cut and identity in which inversion takes place deterministically. We first show the structural rules.

$$\frac{}{\Gamma, p^+ \Vdash [p^+]}\ \mathsf{id}^+ \qquad \frac{}{\Gamma, [p^-] \Vdash p^-}\ \mathsf{id}^-$$

$$\frac{(\Gamma \text{ stable}^-) \quad \Gamma \Vdash [C^+]}{\Gamma \Vdash C^+}\ \mathsf{focus}^+$$

$$\frac{(\Gamma \text{ stable}^-, C \text{ stable}^+) \quad \Gamma, A^-, [A^-] \Vdash C}{\Gamma, A^- \Vdash C}\ \mathsf{focus}^-$$

Note that in the negative focusing rule we *copy* the proposition A^-, which will be the only instance of an explicit contraction-like behavior. In all other two-premise rules, we will propagate the antecedents to both premises. The remaining rules can be constructed straightforwardly from the general

principles we have laid out.

$$\frac{\Gamma, A^+ \Vdash B^-}{\Gamma \Vdash A^+ \to B^-}\ {\to}R \qquad \frac{\Gamma \Vdash [A^+] \quad \Gamma, [B^-] \Vdash C}{\Gamma, [A^+ \to B^-] \Vdash C}\ {\to}L$$

$$\frac{\Gamma \Vdash A^- \quad \Gamma \Vdash B^-}{\Gamma \Vdash A^- \,\&\, B^-}\ \&R \qquad \frac{\Gamma, [A^-] \Vdash C}{\Gamma, [A^- \,\&\, B^-] \Vdash C}\ \&L_1 \qquad \frac{\Gamma, [B^-] \Vdash C}{\Gamma, [A^- \,\&\, B^-] \Vdash C}\ \&L_2$$

$$\frac{\Gamma \Vdash A^+}{\Gamma \Vdash \uparrow A^+}\ \uparrow R \qquad \frac{\Gamma, A^+ \Vdash C}{\Gamma, [\uparrow A^+] \Vdash C}\ \uparrow L$$

$$\frac{\Gamma \Vdash [A^+] \quad \Gamma \Vdash [B^+]}{\Gamma \Vdash [A^+ \times B^+]}\ {\times}R \qquad \frac{\Gamma, A^+, B^+ \Vdash C}{\Gamma, A^+ \times B^+ \Vdash C}\ {\times}L$$

$$\frac{}{\Gamma \Vdash [\mathbf{1}]}\ \mathbf{1}R \qquad \frac{\Gamma \Vdash C}{\Gamma, \mathbf{1} \Vdash C}\ \mathbf{1}L$$

$$\frac{\Gamma \Vdash [A^+]}{\Gamma \Vdash [A^+ + B^+]}\ {+}R_1 \qquad \frac{\Gamma \Vdash [B^+]}{\Gamma \Vdash [A^+ + B^+]}\ {+}R_2 \qquad \frac{\Gamma, A^+ \Vdash C \quad \Gamma, B^+ \Vdash C}{\Gamma, A^+ + B^+ \Vdash C}\ {+}L$$

$$\frac{\Gamma \Vdash A^-}{\Gamma \Vdash [\downarrow A^-]}\ \downarrow R \qquad \frac{\Gamma, A^- \Vdash C}{\Gamma, \downarrow A^- \Vdash C}\ \downarrow L$$

3 Quantifiers

The quantifiers follow the familiar patterns; we saw their basic structure in Lecture 14. We can deduce their polarity by seeing which rules are invertible, or which rule is applied first in the identity expansion. Clearly, they are $\forall R$ and $\exists L$.

$$\begin{array}{rcl} A^- & ::= & \ldots \mid \forall x{:}\tau.\ A^- \\ A^+ & ::= & \ldots \mid \exists x{:}\tau.\ A^+ \end{array}$$

We generalize the judgment to allow term variables τ to be declared in a signature Σ which is propagated to all premises in all rules: type declara-

tions for term variables are persistent.

$$\frac{\Sigma, a{:}\tau\ ;\ \Gamma \Vdash A(a)^-}{\Sigma\ ;\ \Gamma \Vdash \forall x{:}\tau.\, A(x)^-}\ \forall R^a \qquad \frac{\Sigma \vdash t : \tau \quad \Sigma\ ;\ \Gamma, [A(t)^-] \Vdash C}{\Sigma\ ;\ \Gamma, [\forall x{:}\tau.\, A(x)^-] \Vdash C}\ \forall L$$

$$\frac{\Sigma \vdash t : \tau \quad \Sigma\ ;\ \Gamma \Vdash [A(t)^+]}{\Sigma\ ;\ \Gamma \Vdash [\exists x{:}\tau.\, A(x)^+]}\ \exists R \qquad \frac{\Sigma, a{:}\tau\ ;\ \Gamma, A(a)^+ \Vdash C}{\Sigma\ ;\ \Gamma, \exists x{:}\tau.\, A(x)^+ \Vdash C}\ \exists L^a$$

4 Shifting Focus Between Logics

For the moment, we are interested in a logic that combines ordered with unrestricted propositions, with a very thin layer of unrestricted propositions.[1]

$$\begin{array}{lcl} A^-_{\mathsf{U}} & ::= & p^-_{\mathsf{U}} \mid {}^{\mathsf{U}}_{\mathsf{O}}\!\Downarrow A^-_{\mathsf{O}} \\ A^+_{\mathsf{U}} & ::= & p^+_{\mathsf{U}} \\ A^-_{\mathsf{O}} & ::= & p^-_{\mathsf{O}} \mid \ldots \mid \uparrow^{\mathsf{O}}_{\mathsf{O}} A^+ \\ A^+_{\mathsf{O}} & ::= & p^+_{\mathsf{O}} \mid \ldots \mid \downarrow^{\mathsf{O}}_{\mathsf{O}} A^- \mid {}^{\mathsf{U}}_{\mathsf{O}}\!\Uparrow A^+_{\mathsf{U}} \end{array}$$

We get the following additional rules

$$\frac{}{\Gamma, p^+_{\mathsf{U}} \Vdash [p^+_{\mathsf{U}}]}\ \mathsf{id}^+_{\mathsf{U}} \qquad \frac{}{\Gamma, [p^-_{\mathsf{U}}] \Vdash p^-_{\mathsf{U}}}\ \mathsf{id}^-_{\mathsf{U}}$$

$$\frac{(\Gamma\ \mathsf{stable}^-, \Omega\ \mathsf{stable}^-, C_{\mathsf{O}}\ \mathsf{stable}^+) \quad \Gamma, A^-_{\mathsf{U}}, [A^-_{\mathsf{U}}]\ ;\ \Omega \Vdash C_{\mathsf{O}}}{\Gamma, A^-_{\mathsf{U}}\ ;\ \Omega \Vdash C_{\mathsf{O}}}\ \mathsf{focus}^-_{\mathsf{UO}} \qquad \frac{(\Gamma\ \mathsf{stable}^-, C_{\mathsf{U}}\ \mathsf{stable}^+) \quad \Gamma, A^-_{\mathsf{U}}, [A^-_{\mathsf{U}}] \Vdash C_{\mathsf{U}}}{\Gamma, A^-_{\mathsf{U}} \Vdash C_{\mathsf{U}}}\ \mathsf{focus}^-_{\mathsf{UU}}$$

$$\frac{(\Gamma\ \mathsf{stable}^-) \quad \Gamma \Vdash [A^+_{\mathsf{U}}]}{\Gamma \Vdash A^+_{\mathsf{U}}}\ \mathsf{focus}^+_{\mathsf{U}}$$

$$\frac{\Gamma\ ;\ \cdot \Vdash A^-_{\mathsf{O}}}{\Gamma \Vdash {}^{\mathsf{U}}_{\mathsf{O}}\!\Downarrow A^-_{\mathsf{O}}}\ {}^{\mathsf{U}}_{\mathsf{O}}\!\Downarrow R \qquad \frac{\Gamma\ ;\ \Omega_1\ [A^-_{\mathsf{O}}]\ \Omega_2 \Vdash C}{\Gamma, [{}^{\mathsf{U}}_{\mathsf{O}}\!\Downarrow A^-_{\mathsf{O}}]\ ;\ \Omega_1\ \Omega_2 \Vdash C}\ {}^{\mathsf{U}}_{\mathsf{O}}\!\Downarrow L$$

$$\frac{\Gamma \Vdash [A^+_{\mathsf{O}}]}{\Gamma\ ;\ \cdot \Vdash [{}^{\mathsf{U}}_{\mathsf{O}}\!\Uparrow A^+_{\mathsf{O}}]}\ {}^{\mathsf{U}}_{\mathsf{O}}\!\Uparrow R \qquad \frac{\Gamma, A^+_{\mathsf{U}}\ ;\ \Omega_1\ \Omega_2 \Vdash C_{\mathsf{O}}}{\Gamma\ ;\ \Omega_1\ ({}^{\mathsf{U}}_{\mathsf{O}}\!\Uparrow A^+_{\mathsf{U}})\ \Omega_2 \Vdash C_{\mathsf{O}}}\ {}^{\mathsf{U}}_{\mathsf{O}}\!\Uparrow L$$

[1] As I am writing this, I am not at all sure that the polarity-preserving, mode-shifting modalities really work.

How do we now specify the ordered operational semantics of subsingleton logic? We consider a couple of rules.

$$\frac{\mathsf{proc}(P \mid Q)}{\mathsf{proc}(\mathsf{P}) \quad \mathsf{proc}(Q)}\ \mathsf{cmp}$$

This rule may be used many times, so we get

$$\mathsf{U}_{\mathsf{O}}\Uparrow(\forall P.\, \forall Q.\, \mathsf{proc}(P \mid Q) \setminus \uparrow(\mathsf{proc}(P) \bullet \mathsf{proc}(Q)))$$

Focusing on this unrestricted proposition will create the computation rule

$$\frac{\Omega_1\ \mathsf{proc}(P)\ \mathsf{proc}(Q)\ \Omega_2 \Vdash C}{\Omega_1\ \mathsf{proc}(P \mid Q)\ \Omega_2 \Vdash C}\ \mathsf{cmp}$$

which corresponds exactly the original ordered inference rule cmp just above. In the following, we omit the explicit $\mathsf{U}_{\mathsf{O}}\Uparrow$ and the outermost quantifiers, using the conventions that rules are persistent and that upper case variables are implicitly universally quantified.

Next, the rule for disjunction.

$$\frac{\mathsf{proc}(\mathsf{R}.l_k\ ;\ P) \quad \mathsf{proc}(\mathsf{caseL}\ (l_i \Rightarrow Q_i)_{i \in I})}{\mathsf{proc}(P) \quad \mathsf{proc}(Q_k)}\ \oplus C$$

Propositionally:

$$\mathsf{U}_{\mathsf{O}}\Uparrow \forall P{:}\mathsf{pexp}.\ \forall I{:}\mathsf{idx}.\ \forall Q{:}\Pi i{\in}I.\ \mathsf{pexp}.\ \forall k{\in}I.$$
$$\mathsf{proc}(\mathsf{R}.l_k\ ;\ P) \bullet \mathsf{proc}(\mathsf{caseL}\ (l_i \Rightarrow Q_i)_{i \in I}) \setminus \uparrow (\mathsf{proc}(P) \bullet \mathsf{proc}(Q_k))$$

We add the remaining rules, omitting the leading shifts and quantifiers which can easily be inferred. We also label each rule with a name, which we will eventually see as a dependent type declaration.

$$\begin{array}{lcl}
\mathsf{cmp} & : & \mathsf{proc}(P \mid Q) \setminus \uparrow(\mathsf{proc}(P) \bullet \mathsf{proc}(Q)) \\
\mathsf{fwd} & : & \mathsf{proc}(\leftrightarrow) \setminus \uparrow \mathbf{1} \\
\oplus C & : & \mathsf{proc}(\mathsf{R}.l_k\ ;\ P) \bullet \mathsf{proc}(\mathsf{caseL}\ (l_i \Rightarrow Q_i)_{i \in I}) \setminus \uparrow (\mathsf{proc}(P) \bullet \mathsf{proc}(Q_k)) \\
\& C & : & \mathsf{proc}(\mathsf{caseR}\ (l_i \Rightarrow P_i)_{i \in I}) \bullet \mathsf{proc}(\mathsf{L}.l_k\ ;\ Q) \setminus \uparrow (\mathsf{proc}(P_k) \bullet \mathsf{proc}(Q)) \\
\mathbf{1}C & : & \mathsf{proc}(\mathsf{closeR}) \bullet \mathsf{proc}(\mathsf{waitL}\ ;\ Q) \setminus \uparrow \mathsf{proc}(Q)
\end{array}$$

5 Example: Ordered Processes

When we generalize away from the subsingleton fragment, we needed to introduce channels because a process might communicate along any of the channels it uses. Consequently, we used *linear inference* rather than ordered inference to describe the operational semantics. The substructural operational semantics then uses the linear fragment rather than the ordered one. The principles of chaining, inversion, and focusing are completely analogous, so we will just use it without further formalities.

Processes are now captured with the predicate $\mathsf{proc}(x, P)$ which is process P offering a service along channel x. We begin with the rule of composition for spawning a new process, providing along a new channel z.

$$\frac{\mathsf{proc}(x, y \leftarrow P(y)\ ;\ Q(y))}{\mathsf{proc}(z, P(z)) \quad \mathsf{proc}(x, Q(z))}\ \mathsf{cmp}^z$$

Omitting quantifiers as in the previous example, we start with something like

$$\mathsf{cmp} \quad : \quad \mathsf{proc}(X, y \leftarrow P(y)\ ;\ Q(y)) \multimap \uparrow(\mathsf{proc}(z, P(z)) \otimes \mathsf{proc}(X, Q(z))) \quad ??$$

The problem here is the status of z. It it were a free variable in the rule and therefore implicitly universally quantified, we could choose any channel for z, including, say, X, which is obviously incorrect: z must be chosen fresh. The answer here is to *existentially quantify* over z.

$$\mathsf{cmp} \quad : \quad \mathsf{proc}(X, y \leftarrow P(y)\ ;\ Q(y)) \multimap \uparrow(\exists z{:}\mathsf{ch}.\ \mathsf{proc}(z, P(z)) \otimes \mathsf{proc}(X, Q(z)))$$

To see why this is correct, let us consider focusing on

$$a^+ \multimap \uparrow(\exists z.\, b^+(z) \otimes c^+(z))$$

which is a slightly abstracted version of the rule above. First, the chaining phase.

$$\frac{(\Delta = (\Delta_1, \Delta_2)) \quad \dfrac{(\Delta_1 = a^+)}{\Sigma\ ;\ \Delta_1 \Vdash [a^+]}\ \mathsf{id}^+ \quad \dfrac{\vdots}{\Sigma\ ;\ \Delta_2, \exists z.\, b^+(z) \otimes c^+(z) \Vdash G}}{\Sigma\ ;\ \Delta_2, [\uparrow(\exists z.\, b^+(z) \otimes c^+(z))] \Vdash G}\ \uparrow L \qquad \multimap L$$
$$\Sigma\ ;\ \Delta, [a^+ \multimap \uparrow(\exists z.\, b^+(z) \otimes c^+(z))] \Vdash G$$

In the only remaining subgoal, we need to complete the inversion phase.

$$\dfrac{(\Delta = (\Delta_1, \Delta_2)) \quad \dfrac{(\Delta_1 = a^+)}{\Sigma\ ;\ \Delta_1 \Vdash [a^+]}\ \mathsf{id}^+ \quad \dfrac{\dfrac{\dfrac{\Sigma, z{:}\tau\ ;\ \Delta_2, b^+(z), c^+(z) \Vdash G}{\Sigma, z{:}\tau\ ;\ \Delta_2, b^+(z) \otimes c^+(z) \Vdash G}\otimes L}{\Sigma\ ;\ \Delta_2, \exists z.\, b^+(z) \otimes c^+(z) \Vdash G}\exists L^z}{\Sigma\ ;\ \Delta_2, [\uparrow(\exists z.\, b^+(z) \otimes c^+(z))] \Vdash G}\uparrow L}{\Sigma\ ;\ \Delta, [a^+ \multimap \uparrow(\exists z.\, b^+(z) \otimes c^+(z))] \Vdash G}\multimap L$$

This gives us the following synthetic rule of inference:

$$\dfrac{\Sigma, z{:}\tau\ ;\ \Delta_2, b^+(z), c^+(z) \Vdash G}{\Sigma\ ;\ \Delta_2, a^+ \Vdash G}\ R^z$$

In our particular example, we get

$$\dfrac{\Sigma, z{:}\mathsf{ch}\ ;\ \Delta, \mathsf{proc}(z, P(z)), \mathsf{proc}(X, Q(z)) \Vdash G}{\Sigma\ ;\ \Delta, \mathsf{proc}(X, y \leftarrow P(y)\ ;\ Q(y)) \Vdash G}\ \mathsf{cmp}^z$$

which is exactly what we were hoping for, since it is the correct sequent calculus rendering of our original linear inference rule

$$\dfrac{\mathsf{proc}(x, y \leftarrow P(y)\ ;\ Q(y))}{\mathsf{proc}(z, P(z)) \quad \mathsf{proc}(x, Q(z))}\ \mathsf{cmp}^z$$

From this example, we can now write some of the other rules. For the sake

of brevity, we specify the synchronous versions.

$$
\begin{array}{lcl}
\mathsf{cmp} & : & \mathsf{proc}(X, y \leftarrow P(y)\ ;\ Q(y)) \\
& & \quad \multimap \uparrow (\exists z{:}\mathsf{ch}.\ \mathsf{proc}(z, P(z)) \otimes \mathsf{proc}(X, Q(z))) \\
\oplus C & : & \mathsf{proc}(X, X.l_k\ ;\ P) \otimes \mathsf{proc}(Z, \mathsf{case}\ X\ (l_i \Rightarrow Q_i)_{i \in I}) \\
& & \quad \multimap \uparrow (\mathsf{proc}(X, P) \otimes \mathsf{proc}(Z, Q_k)) \\
\& C & : & \mathsf{proc}(X, \mathsf{case}\ X\ (l_i \Rightarrow P_i)_{i \in I}) \otimes \mathsf{proc}(Z, X.l_k\ ;\ Q) \\
& & \quad \multimap \uparrow (\mathsf{proc}(X, P_k) \otimes \mathsf{proc}(Z, Q)) \\
\otimes C & : & \mathsf{proc}(X, \mathsf{send}\ X\ W\ ;\ P) \otimes \mathsf{proc}(Z, y \leftarrow \mathsf{recv}\ X\ ;\ Q(y)) \\
& & \quad \multimap \uparrow (\mathsf{proc}(X, P) \otimes \mathsf{proc}(Z, Q(W))) \\
\multimap C & : & \mathsf{proc}(X, y \leftarrow \mathsf{recv}\ X\ ;\ P(y)) \otimes \mathsf{proc}(Z, \mathsf{send}\ X\ W\ ;\ Q) \\
& & \quad \multimap \uparrow (\mathsf{proc}(X, P(W)) \otimes \mathsf{proc}(Z, Q)) \\
\mathbf{1} C & : & \mathsf{proc}(X, \mathsf{close}\ X) \otimes \mathsf{proc}(Z, \mathsf{wait}\ X\ ;\ Q) \\
& & \quad \multimap \uparrow \mathsf{proc}(Q) \\
\mathsf{fwd} & : & \mathsf{proc}(X, X \leftarrow Y) \multimap \uparrow X \doteq Y
\end{array}
$$

Only the last rule requires a new form of linear proposition, namely equality. We use it here only for parameters at type ι without any constant constructors to avoid a more extended development. Its right rule is just reflexivity; its left rules performs substitution.

$$\frac{}{\Sigma, x{:}\iota\ ;\ \cdot \vdash x \doteq x}\ \doteq R \qquad \frac{\Sigma, z{:}\iota\ ;\ \Delta(z,z) \vdash C(z,z)}{\Sigma, x{:}\iota, y{:}\iota\ ;\ \Delta(x,y), x \doteq y \vdash C(x,y)}\ \doteq L^z$$

This form of equality is *positive*: the left rule is invertible. This means that the focusing versions are

$$\frac{}{\Sigma, x{:}\iota\ ;\ \cdot \Vdash [x \doteq x]}\ \doteq R \qquad \frac{\Sigma, z{:}\iota\ ;\ \Delta(z,z) \Vdash C(z,z)}{\Sigma, x{:}\iota, y{:}\iota\ ;\ \Delta(x,y), x \doteq y \Vdash C(x,y)}\ \doteq L^z$$

Note that we could and perhaps should retain x and y in the signature in the premise, but they can no longer occur in $\Delta(z,z)$ or $G(z,z)$ so we have removed them.

Playing through focusing on the forwarding rule

$$\mathsf{fwd} : \mathsf{proc}(X, X \leftarrow Y) \multimap \uparrow X \doteq Y$$

we get the following synthetic rule

$$\frac{\Sigma, z{:}\mathsf{ch}\ ;\ \Delta(z,z) \Vdash C(z,z)}{\Sigma, x{:}\mathsf{ch}, y{:}\mathsf{ch}\ ;\ \Delta(x,y), \mathsf{proc}(x, x \leftarrow y) \Vdash C(x,y)}\ \mathsf{fwd}^z$$

which is the correct rendering of our (sketched) rule of linear inference: the channels x and y are identified. Since variables can be consistently renamed in a judgment, we could write equivalently $\Sigma, x{:}\mathsf{ch}\ ;\ \Delta(x,x) \Vdash C(x,x)$ or $\Sigma, y{:}\mathsf{ch}\ ;\ \Delta(y,y) \Vdash C(y,y)$. Parameters here are *not* names in the sense of nominal logic, since we cannot compare them for *disequality*. In fact, doing so would be wrong: the identity rule would then be unsound since it unifies two previously distinct parameters.

Exercises

Exercise 1 Write out the ordered SSOS rules for the asynchrononous semantics for subsingleton logic, using a msg predicate.

Exercise 2 Write out the linear SSOS rules for the asynchronous semantics for ordered logic, using a msg predicate.

Exercise 3 Integrate a structural layer into ordered logic using $\uparrow^{\mathsf{U}}_{\mathsf{O}} A_{\mathsf{O}}$ and $\downarrow^{\mathsf{U}}_{\mathsf{O}} A_{\mathsf{U}}$. Then use the modalities of Section 4 to extend the synchronous semantics of Section 5 to the new shift constructs.

References

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

[CPWW02] 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.

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

[Pfe04] Frank Pfenning. Substructural operational semantics and linear destination-passing style. In W.-N. Chin, editor, *Proceedings of the 2nd Asian Symposium on Programming Languages and Systems (APLAS'04)*, page 196, Taipei, Taiwan, November 2004. Springer-Verlag LNCS 3302. Abstract of invited talk.

[PS09] Frank Pfenning and Robert J. Simmons. Substructural operational semantics as ordered logic programming. In *Proceedings of the 24th Annual Symposium on Logic in Computer Science (LICS 2009)*, pages 101–110, Los Angeles, California, August 2009. IEEE Computer Society Press.

[Sim12] Robert J. Simmons. *Substructural Logical Specifications*. PhD thesis, Carnegie Mellon University, November 2012. Available as Technical Report CMU-CS-12-142.

[Sim14] Robert J. Simmons. Structural focalization. *ACM Transactions on Computational Logic*, 15(3):21:1–21:33, 2014.

[SP08] Robert J. Simmons and Frank Pfenning. Linear logical algorithms. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming (ICALP'08)*, pages 336–345, Reykjavik, Iceland, July 2008. Springer LNCS 5126.

[SP09] Robert J. Simmons and Frank Pfenning. Linear logical approximations. In G. Puebla and G. Vidal, editors, *Proceedings of the Workshop on Partial Evaluation and Program Manipulation*, pages 9–20, Savannah, Georgia, January 2009. ACM SIGPLAN.

[SP11] Robert J. Simmons and Frank Pfenning. Logical approximation for program analysis. *Higher-Order and Symbolic Computation*, 24(1–2):41–80, 2011.

[WCPW02] 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 Call-by-Push-Value

15-816: Substructural Logics
Frank Pfenning

Lecture 21
November 10, 2016

In this lecture we first present natural deduction in its pure form and then *polarize* it into negative and positive proposition in order to make it more directly suitable as the basis for a functional programming language. As we may surmise from earlier lectures, positive propositions type values, while negative propositions type computations. The resulting *call-by-push-value* system [Lev01] can compositionally embed both call-by-value and call-by-name.

1 Natural Deduction

Natural deduction was first introduced by Gentzen [Gen35] as a formalization of ordinary mathematical reasoning with connectives. In contrast, he considered his sequent calculus a technical device for proving consistency via his Hauptsatz. As such one might consider natural deduction as the most fundamental means to define the logical connectives. We will not dwell on this, but the key aspect of *harmony* that we formulated on the sequent calculus was first expressed in natural deduction [Dum91, ML83].

Instead of right and left rules that define the connectives, we have *introduction* and *elimination* rules. Roughly speaking, an introduction rule corresponds to a right rule: it shows how to prove a proposition. An elimination rule corresponds to a left rule and shows how to use a proposition. But rather than decomposing the proposition from the conclusion to the premise, the elimination rule should be read from the premise to the conclusion.

From a judgmental point of view, natural deduction is based on a single basic judgment $A\ \mathit{true}$ and a hypothetical judgment

$$A_1\ \mathit{true} \ldots A_n\ \mathit{true} \vdash A\ \mathit{true}$$

while the sequent calculus uses *two* judgments, A *ante* only as antecedents, and A *succ* only as a succedent

$$A_1\ \mathit{ante} \ldots A_n\ \mathit{ante} \vdash A\ \mathit{succ}$$

Consequently, in natural deduction there is only a hypothesis rule and no cut rule, although we have an admissible rule of substitution from the very nature of hypothetical judgments. Following Levy, we give here the structural form of all judgments, not the ordered or linear ones, but they of course exist as well.

$$\frac{}{\Gamma, A \vdash A}\ \mathsf{hyp} \qquad \frac{\Gamma \vdash A \quad \Gamma, A \vdash C}{\Gamma \vdash C}\ \mathsf{subst}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ {\to}I \qquad \frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}\ {\to}E$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \,\&\, B}\ \&I \qquad \frac{\Gamma \vdash A \,\&\, B}{\Gamma \vdash A}\ \&E_1 \qquad \frac{\Gamma \vdash A \,\&\, B}{\Gamma \vdash B}\ \&E_2$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \times B}\ {\times}I \qquad \frac{\Gamma \vdash A \times B \quad \Gamma, A, B \vdash C}{\Gamma \vdash C}\ {\times}E$$

$$\frac{}{\Gamma \vdash \mathbf{1}}\ \mathbf{1}I \qquad \frac{\Gamma \vdash \mathbf{1} \quad \Gamma \vdash C}{\Gamma \vdash C}\ \mathbf{1}E$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A + B}\ {+}I_1 \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A + B}\ {+}I_2 \qquad \frac{\Gamma \vdash A + B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}\ {+}E$$

Since the operational reading will differ, we have two forms of conjunction with the same introduction rule but different elimination rules. We see the characteristic form of eliminations for the negative connectives which turn a proof of a conclusion into hypotheses of its parts. That leads to some anomalies, like the $\mathbf{1}E$ rule in which the conclusion is equal to the second premise. Again, this can have some operational meaning so we include it here despite it apparent logical redundancy.

The name *substitution* expresses that we substitute the proof of A for uses of A in the proof of C. Martin-Löf [ML83] treats this as the very definition of what a hypothetical judgment means. When we come to the functional programming language, this will correspond to substitution of terms which is the engine that drives computation. This is very different from the sequent calculus, where cut reduction proceeds in much smaller steps.

2 Polarizing Natural Deduction

While polarization of propositions is entailed by their very nature and there is no choice, it is not immediately obvious how we should polarize the hypothetical judgments. With a view towards our goal of functional programming and the nature of implication as having the form $A^+ \to B^-$ we decide that *hypotheses should always be positive*, $\Gamma^+ \vdash C$. The conclusion C sometimes must be negative (for example, the $\to I$ and $\& I$) and sometimes positive (for example, $\times I$, $\mathbf{1}I$, and $\oplus I_i$). But let's recall the structural polarized propositions. Since the positive propositions will correspond to values we also call them *value types*, while negative propositions are *computation types*.

$$\begin{array}{lrcl} \text{Computation Types} & A^- & ::= & A^+ \to B^- \mid A^- \,\&\, B^- \mid \uparrow A^+ \\ \text{Value Types} & A^+ & ::= & A^+ \times B^+ \mid \mathbf{1} \mid A^+ + B^+ \mid \downarrow A^- \end{array}$$

The rules can be polarized straightforwardly, after some basic decisions. Besides requiring the hypotheses to be entirely positive (that is, ranging over *values*), we allow the elimination rules only when the conclusion C is

negative since elimination corresponds to some computation, not a value.

$$\frac{\Gamma \vdash A^+ \quad \Gamma, A^+ \vdash C^-}{\Gamma \vdash C^-}\ \mathsf{subst}^- \qquad \frac{\Gamma \vdash A^+ \quad \Gamma, A^+ \vdash C^+}{\Gamma \vdash C^+}\ \mathsf{subst}^+$$

$$\frac{}{\Gamma, A^+ \vdash A^+}\ \mathsf{hyp}$$

$$\frac{\Gamma, A^+ \vdash B^-}{\Gamma \vdash A^+ \to B^-} \to I \qquad \frac{\Gamma \vdash A^+ \to B^- \quad \Gamma \vdash A^+}{\Gamma \vdash B^-} \to E$$

$$\frac{\Gamma \vdash A^- \quad \Gamma \vdash B^-}{\Gamma \vdash A^- \,\&\, B^-} \,\&I \qquad \frac{\Gamma \vdash A^- \,\&\, B^-}{\Gamma \vdash A^-} \,\&E_1 \qquad \frac{\Gamma \vdash A^- \,\&\, B^-}{\Gamma \vdash B^-} \,\&E_2$$

$$\frac{\Gamma \vdash A^+ \quad \Gamma \vdash B^+}{\Gamma \vdash A^+ \times B^+} \times I \qquad \frac{\Gamma \vdash A^+ \times B^+ \quad \Gamma, A^+, B^+ \vdash C^-}{\Gamma \vdash C^-} \times E$$

$$\frac{}{\Gamma \vdash \mathbf{1}}\ \mathbf{1}I \qquad \frac{\Gamma \vdash \mathbf{1} \quad \Gamma \vdash C^-}{\Gamma \vdash C^-}\ \mathbf{1}E$$

$$\frac{\Gamma \vdash A^+}{\Gamma \vdash A^+ + B^+} +I_1 \qquad \frac{\Gamma \vdash B^+}{\Gamma \vdash A^+ + B^+} +I_2$$

$$\frac{\Gamma \vdash A^+ + B^+ \quad \Gamma, A^+ \vdash C^- \quad \Gamma, B^+ \vdash C^-}{\Gamma \vdash C^-} +E$$

The interesting part now pertains to the shifts. Using our restrictions on contexts and conclusion, we derive the following rules. The form of the elimination rule, either direct or with a side formula C, depends on the polarity of the proposition that is exposed, not the shift we eliminate.

$$\frac{\Gamma \vdash A^+}{\Gamma \vdash \uparrow A^+} \uparrow I \qquad \frac{\Gamma \vdash \uparrow A^+ \quad \Gamma, A^+ \vdash C^-}{\Gamma \vdash C^-} \uparrow E$$

$$\frac{\Gamma \vdash A^-}{\Gamma \vdash \downarrow A^-} \downarrow I \qquad \frac{\Gamma \vdash \downarrow A^-}{\Gamma \vdash A^-} \downarrow E$$

3 Term Assignments

We now assign program terms to the various typing rules. We have computations M of type A^- and values V of type A^+. Note that variables are always of positive type and therefore stand for values. This does *not* imply a call-by-value strategy, as we will see in the next lecture. First, the hypothesis rule is straightforward.

$$\frac{}{\Gamma, x{:}A^+ \vdash x : A^+}\ \mathsf{hyp}$$

The substitution principles, when annotated with proof terms, allow us to apply well-typed substitutions. Note that they are admissible rather than primitive typing rules. They tell us the result of the substitution is well-typed whenever we substitute a value of type A^+ for a variable x of type A^+. This is the only substitution we perform, since all variables range over values.

$$\frac{\Gamma \vdash V : A^+ \quad \Gamma, x{:}A^+ \vdash M : C^-}{\Gamma \vdash [V/x]M : C^-}\ \mathsf{subst}^- \qquad \frac{\Gamma \vdash V : A^+ \quad \Gamma, x{:}A^+ \vdash W : C^+}{\Gamma \vdash [V/x]W : C^+}\ \mathsf{subst}^+$$

We now complete the picture by giving all the introduction rules for positive propositions, omitting only the shift for now.

$$\frac{\Gamma \vdash V : A^+ \quad \Gamma \vdash W : B^+}{\Gamma \vdash (V, W) : A^+ \times B^+}\ \times I$$

$$\frac{\Gamma \vdash V : A^+ \times B^+ \quad \Gamma, x{:}A^+, y{:}B^+ \vdash N : C^-}{\Gamma \vdash \mathsf{match}\, V\, \mathsf{as}\, (x, y) \Rightarrow N : C^-}\ \times E$$

$$\frac{}{\Gamma \vdash ( ) : \mathbf{1}}\ \mathbf{1}I \qquad \frac{\Gamma \vdash V : \mathbf{1} \quad \Gamma \vdash N : C^-}{\Gamma \vdash \mathsf{match}\, V\, \mathsf{as}\, ( ) \Rightarrow N : C^-}\ \mathbf{1}E$$

$$\frac{\Gamma \vdash V : A^+}{\Gamma \vdash \mathsf{inl}(V) : A^+ + B^+}\ +I_1 \qquad \frac{\Gamma \vdash W : B^+}{\Gamma \vdash \mathsf{inr}(W) : A^+ + B^+}\ +I_2$$

$$\frac{\Gamma \vdash V : A^+ + B^+ \quad \Gamma, x{:}A^+ \vdash M : C^- \quad \Gamma, y{:}B^+ \vdash N : C^-}{\Gamma \vdash \mathsf{match}\, V\, \mathsf{as}\, (\mathsf{inl}(x) \Rightarrow M \mid \mathsf{inr}(y) \Rightarrow N) : C^-}\ +E$$

For functional programming, just as for session types, it is convenient to

generalize binary disjunction $A + B$ into a labeled sum $+\{l : A_l\}_{l \in L}$.

$$\frac{\Gamma \vdash V : A_k \quad (k \in L)}{\Gamma \vdash k(V) : +\{l : A_l\}_{l \in L}} \, +I_k$$

$$\frac{\Gamma \vdash V : +\{l : A_l\}_{l \in L} \quad \Gamma, x{:}A_l \vdash M_l : C^- \quad (\forall l \in L)}{\Gamma \vdash \mathsf{match}\, V\, \mathsf{as}\, (l(x) \Rightarrow M_l)_{l \in L} : C^-} \, +E$$

At this point we only need the shift to complete the value types. The positive type is $\downarrow A^-$. Since A^- represents a computation, the value of type $\downarrow A^-$ represents a suspended computation. Following tradition, we call this a *thunk*. The elimination rule forces a thunk to be evaluated.

$$\frac{\Gamma \vdash M : A^-}{\Gamma \vdash \mathsf{thunk}\, M : \downarrow A^-} \downarrow I \qquad \frac{\Gamma \vdash V : \downarrow A^-}{\Gamma \vdash \mathsf{force}\, V : A^-} \downarrow E$$

Let's take stock. At this point, we have the elimination rules for values as computations, and also the full set of values. Still missing are the computations for negative types. As we start on the negative types, we generalize $A^- \,\&\, B^-$ to the labeled version $\&\{l : A_l\}_{l \in L}$.

$$\frac{\Gamma, x{:}A^+ \vdash M : B^-}{\Gamma \vdash \lambda x.\, M : A^+ \to B^-} \to I \qquad \frac{\Gamma \vdash M : A^+ \to B^- \quad \Gamma \vdash V : A^+}{\Gamma \vdash M\, V : B^-} \to E$$

$$\frac{\Gamma \vdash M_l : A_l^- \quad (\forall l \in L)}{\Gamma \vdash \{l \Rightarrow M_l\}_{l \in L} : \&\{l : A_l^-\}_{l \in L}} \,\&I \qquad \frac{\Gamma \vdash M : \&\{l : A_l^-\}_{l \in L}}{\Gamma \vdash M.l_k : A_k^-} \,\&E_k$$

The computation type $\uparrow A^+$ just embeds a value, which we write as $\mathsf{return}\ V$. The elimination for decomposes this and substitutes the value for a bound variable.

$$\frac{\Gamma \vdash V : A^+}{\Gamma \vdash \mathsf{return}\, V : \uparrow A^+} \uparrow I \qquad \frac{\Gamma \vdash M : \uparrow A^+ \quad \Gamma, x{:}A^+ \vdash N : C^-}{\Gamma \vdash \mathsf{let\ val}\, x = M\, \mathsf{in}\, N : C^-} \uparrow E$$

In summary, including the type on which each construct operates, ei-

ther by introduction or elimination:

$$
\begin{array}{llll}
\text{Computations} & M & ::= & \mathsf{match}\, V\, \mathsf{as}\, (x,y) \Rightarrow M & A^+ \times B^+ \\
& & \mid & \mathsf{match}\, V\, \mathsf{as}\, (\,) \Rightarrow M & \mathbf{1} \\
& & \mid & \mathsf{match}\, V\, \mathsf{as}\, (l(x) \Rightarrow M_l)_{l\in L} & +\{l : A_l^+\}_{l\in L} \\
& & \mid & \mathsf{force}\, V & \downarrow A^- \\
& & \mid & \lambda x.M \mid M\, V & A^+ \to B^- \\
& & \mid & \{l \Rightarrow M_l\}_{l\in L} \mid M.k & \&\{l : A_l^-\}_{l\in L} \\
& & \mid & \mathsf{return}\, V \mid \mathsf{let\ val}\, x = M\, \mathsf{in}\, N & \uparrow A^+ \\
\text{Values} & V & ::= & x & \\
& & \mid & (V,W) & A^+ \times B^+ \\
& & \mid & (\,) & \mathbf{1} \\
& & \mid & l(V) & +\{l : A_l^+\}_{l\in L} \\
& & \mid & \mathsf{thunk}\, M & \downarrow A^-
\end{array}
$$

4 Local Reduction

The analogue of a cut reduction is a *local reduction*. It will call upon *substitution* $[V/x]M$. A local reduction arises when an introduction of a proposition is immediately followed by its elimination.

$$
\begin{array}{lcl}
\mathsf{match}\, (V,W)\, \mathsf{as}\, (x,y) \Rightarrow M & \longrightarrow & [V/x, W/y]M \\
\mathsf{match}\, (\,)\, \mathsf{as}\, (\,) \Rightarrow M & \longrightarrow & M \\
\mathsf{match}\, k(V)\, \mathsf{as}\, (l(x) \Rightarrow M_l)_{l\in L} & \longrightarrow & [V/x]M_k \\
\mathsf{force}\, (\mathsf{thunk}\, M) & \longrightarrow & M \\
(\lambda x.\, M)\, V & \longrightarrow & [V/x]M \\
\{l \Rightarrow M_l\}_{l\in L}.k & \longrightarrow & M_k \\
\mathsf{let\ val}\, x = \mathsf{return}\, V\, \mathsf{in}\, N & \longrightarrow & [V/x]N
\end{array}
$$

Local reductions can be also found in the operational semantics, which imposes a certain strategy on the reductions which give the call-by-push-value strategy its name.

5 Substructural Operational Semantics

Interestingly, we can specify the substructural operational semantics entirely on ordered logic. It takes the form of a stack machine and uses substitution, although other techniques are certainly possible.

As is typical for functional languages, we use three predicates:

- $\mathsf{eval}(M)$, which means computation M should be evaluated.
- $\mathsf{retn}(T)$, which means we return a *terminal computation* T.
- $\mathsf{cont}(K)$, which specifies continuation K waiting for a returned value.

Globally, we start with $\mathsf{eval}(M)$ for a closed term M and apply ordered inference to eventually obtain $\mathsf{retn}(T)$. Here, T stands for a terminal computation, that is, a computation that does not take any further steps. We have

$$\begin{array}{lrcll} \text{Terminal Computations} & T & ::= & \lambda x.\, M & A^+ \to B^- \\ & & \mid & \{l \Rightarrow M_l\}_{l \in L} & A^- \,\&\, B^- \\ & & \mid & \mathsf{return}\, V & \uparrow A \end{array}$$

During the computation, the configuration will always have one of the forms

$$\begin{array}{l} \mathsf{eval}(M)\ \mathsf{cont}(K_n)\ \ldots\ \mathsf{cont}(K_1) \\ \mathsf{retn}(T)\ \mathsf{cont}(K_n)\ \ldots\ \mathsf{cont}(K_1) \end{array}$$

which means that the continuations form a stack. Rather than pre-specify, we will write up the operational semantics and see which kind of continuations we need. It helps to make us aware how evaluations, returns, and continuations are typed. Note that all computations and values are closed, so no hypotheses are needed in their typing.

$$\frac{\cdot \vdash M : A^-}{\mathsf{eval}(M) : A^-} \qquad \frac{\cdot \vdash T : A^-}{\mathsf{retn}(T) : A^-} \qquad \frac{A^- \vdash K : B^-}{A^- \vdash \mathsf{cont}(K) : B^-}$$

The only interesting part here are the continuation stack frames. They expect a terminal computation on the left and return a terminal computation to the right, to the next stack frame.

The first key inside is that values of positive type do not need to be further evaluated. One can see that simply by looking at the typing rules.

So we have

$$\frac{\mathsf{eval}(\mathsf{match}\ (V, W)\ \mathsf{as}\ (x, y) \Rightarrow M)}{\mathsf{eval}([V/x, W/y]M)}\ \times C$$

$$\frac{\mathsf{eval}(\mathsf{match}\ (\,)\ \mathsf{as}\ (\,) \Rightarrow M)}{\mathsf{eval}(M)}\ \mathbf{1}C$$

$$\frac{\mathsf{eval}(\mathsf{match}\ k(V)\ \mathsf{as}\ (l(x) \Rightarrow M_l)_{l \in L})}{\mathsf{eval}([V/x]M_k)}\ +C$$

$$\frac{\mathsf{eval}(\mathsf{force}\ (\mathsf{thunk}\ M))}{\mathsf{eval}(M)}\ \downarrow C$$

The cases for negative types are more complicated, since they require continuations. Fortunately, there are only 3. We start with functions.

$$\frac{\mathsf{eval}(M\ V)}{\mathsf{eval}(M) \quad \mathsf{cont}(\_\ V)}\ \rightarrow C_1 \qquad \frac{\mathsf{eval}(\lambda x.\, M)}{\mathsf{retn}(\lambda x.\, M)}\ \rightarrow C_2 \qquad \frac{\mathsf{retn}(\lambda x.\, M) \quad \mathsf{cont}(\_\ V)}{\mathsf{eval}([V/x]M)}\ \rightarrow C_3$$

We can see the continuation must accept a function from the left and pass its return value to the right. So we have

$$\frac{\cdot \vdash V : A^+}{A^+ \rightarrow B^- \vdash \mathsf{cont}(\_\ V) : B^-}$$

Now we can appreciate why this scheme is called *call-by-push-value*. In the purely functional fragment (only type $A^+ \rightarrow B^-$) the configuration will have one of the two forms

$$\begin{array}{l} \mathsf{eval}(M)\ \mathsf{cont}(\_\ V_n)\ \ldots\ \mathsf{cont}(\_\ V_1) \\ \mathsf{retn}(T)\ \mathsf{cont}(\_\ V_n)\ \ldots\ \mathsf{cont}(\_\ V_1) \end{array}$$

that is, the continuations form a stack of values, and function application will push a value onto the stack. The reason we can still easily represent call-by-name, by the way, is because $\mathsf{thunk}\, M$ is a possible value. We will see that in the next lecture.

Next, products $\&\{l : A_l\}_{l \in L}$. We see that they correspond to *lazy pairs*, because the components are not evaluated until the value of that compo-

nent is requested.

$$\frac{\mathsf{eval}(M.k)}{\mathsf{eval}(M) \quad \mathsf{cont}(\_.k)}\ \&C_1$$

$$\frac{\mathsf{eval}(\{l \Rightarrow M_l\}_{l \in L})}{\mathsf{retn}(\{l \Rightarrow M_l\}_{l \in L})}\ \&C_2$$

$$\frac{\mathsf{retn}(\{l \Rightarrow M_l\}_l \in L) \quad \mathsf{cont}(\_.k)}{\mathsf{eval}(M_k)}\ \&C_3$$

Again, we can derive the typing of the new form of continuation from the computation rules.

$$\frac{}{\&\{l : A_l\}_{l \in L} \vdash \mathsf{cont}(\_.k) : A_k}$$

There is a strong analogy between the typing of this continuation and the typing of a message in SILL.

Finally, values as they are included in computations.

$$\frac{\mathsf{eval}(\mathsf{let\ val}\, x = M\ \mathsf{in}\ N)}{\mathsf{eval}(M) \quad \mathsf{cont}(\mathsf{let\ val}\, x = \_\ \mathsf{in}\ N)}\ \uparrow C_1$$

$$\frac{\mathsf{eval}(\mathsf{return}\, V)}{\mathsf{retn}(\mathsf{return}\, V)}\ \uparrow C_2$$

$$\frac{\mathsf{retn}(\mathsf{return}\, V) \quad \mathsf{cont}(\mathsf{let\ val}\, x = \_\ \mathsf{in}\ N)}{\mathsf{eval}([V/x]N)}\ \uparrow C_3$$

The requisit typing:

$$\frac{x{:}A^+ \vdash N : C^-}{\uparrow A^+ \vdash \mathsf{cont}(\mathsf{let\ val}\, x = \_\ \mathsf{in}\ N) : C^-}$$

6 Example: A Map Function

As a simple example we consider a higher-order function map that applies a function to each element of a given list to construct a new list. The interesting aspect of this exercise it the polarization.

Lists are entirely positive. Not unexpected, given we already encounter this property in the concurrent setting. Of course, we could have *lazy lists*, in which case the type would be quite different (see Exercise 3).

$$\mathsf{list}\, A^+ = +\{\mathsf{cons} : A^+ \times \mathsf{list}\, A^+, \mathsf{nil} : 1\}$$

The map function has to account for this particular polarization.

$$\mathsf{map} : \downarrow(A^+ \to \uparrow B^+) \to \mathsf{list}\, A^+ \to \uparrow \mathsf{list}\, B^+$$

The we define

$$\begin{array}{l}
\mathsf{map} = \lambda f.\, \lambda l. \\
\quad \mathsf{match}\ l\ \mathsf{as}\ (\ \mathsf{cons}(p) \Rightarrow \mathsf{match}\ p\ \mathsf{as}\ (x, l') \Rightarrow \\
\qquad\qquad \mathsf{let\,val}\ y = (\mathsf{force}\ f)\ x\ \mathsf{in} \\
\qquad\qquad \mathsf{let\,val}\ k = (\mathsf{map}\ f)\ l'\ \mathsf{in} \\
\qquad\qquad\quad \mathsf{return}\ \mathsf{cons}(y, l') \\
\qquad\quad \mid \mathsf{nil}(p) \Rightarrow \mathsf{match}\ p\ \mathsf{as}\ (\,) \Rightarrow \\
\qquad\qquad \mathsf{return}\ \mathsf{nil}(\,))
\end{array}$$

Exercises

Exercise 1 The operational semantics uses substitution of values for variables. We can instead *bind* variables to values using a persistent predicate $\underline{\mathsf{bind}}(x, V)$ in operational semantics.

Rewrite the substructural operational semantics using $\underline{\mathsf{bind}}$ so that only (fresh) parameters are substituted for variables, not arbitrary values.

Exercise 2 We can refactor the syntax using *patterns* which are defined as

$$\text{Patterns} \quad p, q \quad ::= x \mid (p, q) \mid () \mid l(p)$$

where variables x always have type $\downarrow A^-$. Then, there is only a single matching construct

$$\mathsf{match}\ V\ \mathsf{as}\ (p_i \Rightarrow M_i)_i$$

Rewrite the typing rules and the operational semantics using patterns.

Exercise 3 Define *lazy lists* as the negative type

$$\mathsf{list}\, A^- = \uparrow + \{\mathsf{cons} : \downarrow \& \{\mathsf{hd} : A^-, \mathsf{tl} : \mathsf{list}\, A^-\}, \mathsf{nil} : \downarrow \& \{\,\}\}$$

Rewrite the map function to work on lazy lists.

References

[Dum91] Michael Dummett. *The Logical Basis of Metaphysics*. Harvard University Press, Cambridge, Massachusetts, 1991. The William James Lectures, 1976.

[Gen35] 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.

[Lev01] Paul Blain Levy. *Call-by-Push-Value*. PhD thesis, University of London, 2001.

[ML83] 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.

Lecture Notes on
Call-by-Value and Call-by-Name

15-816: Substructural Logics
Frank Pfenning

Lecture 22
November 16, 2017

In this lecture we consider two different polarization strategies for structural intuitionistic natural deduction. If we decide to translate propositions *positively* we obtain, under the computational interpretation of proofs as computations, a call-by-value language. If we translate propositions *negatively* we obtain call-by-name. These embeddings are compositional, supporting Levy's claim [Lev01] that call-by-push-value (CBPV) is a unifying approach to functional programming. With CPBV we can easily choose, at a fine-grained level, which computations are eager and which are lazy.

1 Call-by-Value as Positive Polarization

Let's assume we have a source language

Types	A, B, C	$::=$	$A \supset B \mid A \wedge B \mid \top \mid \vee\{l : A_l\}_{l \in L}$	
Expressions	E	$::=$	$x \mid \lambda x.\, E \mid E_1\, E_2$	$A \supset B$
		$\mid$	$\langle E_1, E_2 \rangle \mid \pi_1\, E \mid \pi_2 E$	$A \wedge B$
		$\mid$	$\langle\,\rangle$	$\top$
		$\mid$	$l(E) \mid \mathsf{case}\ E\ (l(x) \Rightarrow E_l)_{l \in L}$	$A \vee B$

We write $(A)^+$ for the positive polarization of A, which is defined inductively as follows:

$$
\begin{array}{lcl}
(A \supset B)^+ & = & \downarrow((A)^+ \rightarrow \uparrow(B)^+) \\
(A \wedge B)^+ & = & (A)^+ \times (B)^+ \\
(\top)^+ & = & \mathbf{1} \\
(\vee\{l : A_l\}_{l \in L})^+ & = & +\{l : (A_l)^+\}_{l \in L}
\end{array}
$$

This would seem to be a minimal positive polarization. Based on this, we can now write translations of expressions. The theorem we are aiming for is

$$\textit{If } \Gamma \vdash E : A \textit{ then } (\Gamma)^+ \vdash (E)^+ : \uparrow (A)^+$$

Note that the translation of an expression should be a computation, so we have to coerce $(A)^+$ to be a negative type in this judgment. This principle and the translation of types leaves very little leeway. Let's work through this carefully for functions. Assume

$$\Gamma \vdash \lambda x.\, E : A \supset B$$

Then

$$\Gamma^+ \vdash (\lambda x.\, E)^+ : \uparrow (A \supset B)^+$$

which works out to

$$\Gamma^+ \vdash (\lambda x.\, E)^+ : \uparrow\downarrow ((A)^+ \to \uparrow (B)^+)$$

We also know

$$\Gamma, x{:}(A)^+ \vdash (E)^+ : \uparrow (B)^+$$

From that, we can fill in

$$(\lambda x.\, E)^+ = \mathsf{return}\ \mathsf{thunk}\, (\lambda x.\, (E)^+)$$

What about application $(E_1\ E_2)^+$? We know

$$\begin{array}{lcl} (\Gamma)^+ \vdash (E_1)^+ & : & \uparrow\downarrow ((A)^+ \to \uparrow (B)^+) \\ (\Gamma)^+ \vdash (E_2)^+ & : & \uparrow (A)^+ \\ (\Gamma)^+ \vdash (E_1\ E_2)^+ & : & \uparrow (B)^+ \end{array}$$

From this we can see that the types almost force:

$$(E_1\ E_2)^+ = \mathsf{let\ val}\ f = (E_1)^+\ \mathsf{in\ let\ val}\ x = (E_2)^+\ \mathsf{in}\ (\mathsf{force}\ f)\, x$$

Also well-typed would be the result of swapping the two lets, or performing one more result binding at the end:

$$\begin{array}{lcl} (E_1\ E_2)^+ & = & \mathsf{let\ val}\ f = (E_1)^+\ \mathsf{in} \\ & & \mathsf{let\ val}\ x = (E_2)^+\ \mathsf{in} \\ & & \mathsf{let\ val}\ y = (\mathsf{force}\ f)\, x\ \mathsf{in} \\ & & \quad \mathsf{return}\ y \end{array}$$

In summary, for functions:

$$\begin{array}{lcl}(x)^+ & = & \mathsf{return}\, x \\ (\lambda\, x.\, E)^+ & = & \mathsf{return\ thunk}\, (\lambda x.(E)^+) \\ (E_1\, E_2)^+ & = & \mathsf{let\ val}\ f = (E_1)^+\ \mathsf{in\ let\ val}\ x = (E_2)^+\ \mathsf{in}\ (\mathsf{force}\ f)\, x\end{array}$$

Translations of pairs is simpler, since we can arrange to use positive (eager) pairs in the target.

$$\begin{array}{lcl}(\langle E_1, E_2\rangle)^+ & = & \mathsf{let\ val}\ x_1 = (E_1)^+\ \mathsf{in\ let\ val}\ x_2 = (E_2)^+\ \mathsf{in\ return}\, (x_1, x_2) \\ (\pi_1\, E)^+ & = & \mathsf{let\ val}\ x = (E)^+\ \mathsf{in\ match}\ x\ \mathsf{as}\ (y, z) \Rightarrow \mathsf{return}\ y \\ (\pi_2\, E)^+ & = & \mathsf{let\ val}\ x = (E)^+\ \mathsf{in\ match}\ x\ \mathsf{as}\ (y, z) \Rightarrow \mathsf{return}\ z\end{array}$$

Similarly for $\top$ and $\vee\{l : A_l\}_{l\in L}$

$$\begin{array}{lcl}(\langle\,\rangle)^+ & = & \mathsf{return}\ (\,) \\ (l(E))^+ & = & \mathsf{let\ val}\ x = (E)^+\ \mathsf{in\ return}\ l(x) \\ (\mathsf{case}\ E\ (l(x) \Rightarrow E_l)_{l\in L})^+ & = & \mathsf{let\ val}\ x = (E)^+\ \mathsf{in\ match}\ x\ \mathsf{as}\ (l(x) \Rightarrow (E_l)^+)_{l\in L}\end{array}$$

At this point we have completed our embedding. We could, for example, give a call-by-value operational semantics on unpolarized expressions and then show that this particular translation is operationally adequate. We are more inclined to think of this translation as the definition of call-by-value and move on.

2 Call-by-Name as Negative Polarization

We now consider the *negative polarization* of an unpolarized type. For the conjunction, we clearly should choose the negative conjunction, corresponding to lazy pairs consistent with call-by-name.

$$\begin{array}{lcl}(A \supset B)^- & = & \downarrow (A)^- \to (B)^- \\ (A \wedge B)^- & = & \&\{\pi_1 : (A)^-, \pi_2 : (B)^-\} \\ (\top)^- & = & \&\{\,\} \\ (\vee\{l : A_l\}_{l\in L}) & = & \uparrow +\{l : \downarrow (A_l)^-\}_{l\in L}\end{array}$$

Now the judgment $\Gamma \vdash E : A$ will be translated to $(\Gamma)^+ \vdash (E)^- : (A)^-$, where for the hypotheses with have

$$(x_1{:}A_1, \ldots, x_n{:}A_n)^+ = x_1{:}\downarrow(A_1)^-, \ldots, x_n{:}\downarrow (A_n)^-$$

The extra down shift for the context is forced since call-by-push-value allows only positively typed variables, while for the translation of an expression to a computation, no additional shift is necessary.

As we design the translation for expressions, let the types be or guide as usual.

$$\begin{array}{lcl} (\lambda x.\, E)^- & : & \downarrow (A)^- \to (B)^- \\ (x)^- & : & \downarrow (A)^- \\ (E)^- & : & (B)^- \end{array}$$

Clearly, we have

$$\begin{array}{lcl} (\lambda x.\, E)^- & = & \lambda x.\, (E)^- \\ (x)^- & = & \mathsf{force}\, x \end{array}$$

In the same style of reasoning:

$$\begin{array}{lcl} (E_1\, E_2)^- & : & (B)^- \\ (E_1)^- & : & \downarrow (A)^- \to (B)^- \\ (E_2)^- & : & (A)^- \end{array}$$

Again, it seems our hand is forced:

$$(E_1\, E_2)^- \quad = \quad (E_1)^-\, (\mathsf{thunk}\, (E_2)^-)$$

In summary, for functions:

$$\begin{array}{lcl} (x)^- & = & \mathsf{force}\, x \\ (\lambda x.\, E)^- & = & \lambda x.\, (E)^- \\ (E_1\, E_2)^- & = & (E_1)^-\, (\mathsf{thunk}\, (E_2)^-) \end{array}$$

This clearly represent call-by-name. We pass a computation, packaged as a thunk, and force that thunk where the variable is used. In all *call-by-need* language such as Haskell, the value of this forced expression is memoized so that future evaluations of $\mathsf{force}\, x$ do not evaluate the thunk again but retrieve its value.

For conjunction, we abbreviate the process. Recall that $(A \wedge B)^- = \&\{\pi_1 : (A)^-, \pi_2 : (B)^-\}$

$$\begin{array}{lcl} (\langle E_1, E_2 \rangle)^- & = & \{\pi_1 \Rightarrow (E_1)^-, \pi_2 \Rightarrow (E_2)^-\} \\ (\pi_1\, E) & = & (E)^-.\pi_1 \\ (\pi_2\, E) & = & (E)^-.\pi_2 \end{array}$$

Truth $\top$ is the nullary case of conjunction and consequently becomes $\&\{\,\}$. We the translate

$$(\langle\rangle)^- \quad = \quad \{\,\}$$

It seems implication and conjunction translates more directly for call-by-name than for call-by-value. However, disjunction has two shifts and is therefore more complicated.

$$\begin{array}{lcl} (l(E))^- & : & \uparrow + \{l : \downarrow (A_l)^-\}_{l \in L} \quad \text{for } l \in L \\ E^- & : & (A_l)^- \end{array}$$

so

$$(l(E))^- \quad = \quad \mathsf{return}\, l(\mathsf{thunk}\, (E)^-)$$

The elimination form is our most complex case

$$\begin{array}{lcl} (\mathsf{case}\ E\ (l(x) \Rightarrow E_l)_{l \in L})^- & : & (C)^- \\ (E)^- & : & \uparrow + \{l : \downarrow (A_l)^-\}_{l \in L} \\ (E_l)^- & : & (C)^- \\ (x)^- & : & \downarrow (A_l)^- \end{array}$$

but if want to respect all these types, the following suggests itself

$$(\mathsf{case}\ E\ (l(x) \Rightarrow E_l)_{l \in L})^- \quad = \quad \mathsf{let\ val}\ y = (E)^- \mathsf{match}\ y\ \mathsf{as}\ (l(x) \Rightarrow (E_l)^-)_{l \in L}$$

Summarizing whole call-by-name translation (which is to say, the negative translation)

$$\begin{array}{lcl} (x)^- & = & \mathsf{force}\, x \\ (\lambda x.\, E)^- & = & \lambda x.\, (E)^- \\ (E_1\, E_2)^- & = & (E_1)^-\, (\mathsf{thunk}\, (E_2)^-) \\ \\ (\langle E_1, E_2 \rangle)^- & = & \{\pi_1 \Rightarrow (E_1)^-, \pi_2 \Rightarrow (E_2)^-\} \\ (\pi_1\, E) & = & (E)^-.\pi_1 \\ (\pi_2\, E) & = & (E)^-.\pi_2 \\ \\ (\langle\,\rangle)^- & = & \{\,\} \\ \\ (l(E))^- & = & \mathsf{return}\, l(\mathsf{thunk}\, (E)^-) \\ (\mathsf{case}\ E\ (l(x) \Rightarrow E_l)_{l \in L})^- & = & \mathsf{let\ val}\ y = (E)^- \mathsf{match}\ y\ \mathsf{as}\ (l(x) \Rightarrow (E_l)^-)_{l \in L} \end{array}$$

3 Destinations

The operational semantics of call-by-push-value is very direct using ordered inference. In the next lecture we will introduce the Concurrent Logical Framework (CLF) which, unfortunately, is linear and does not support

ordered specifications. One idea, not very elegant, is to create explicit sequences of assumptions. But there is a different way, namely to use *destinations* to tie the propositions together. In general, the ordered context

$$A_1 \ \ldots \ A_n$$

is represented by

$$A_1(d_0, d_1) \ A_2(d_1, d_2) \ \ldots \ A_n(d_{n-1}, d_n)$$

where all of $d_0, d_1, d_2, \ldots, d_{n-1}, d_n$ are distinct parameters, and d_0 and d_n represent the left and right endpoints [SP11]. You can think of them just as if they were *channels* in our previous linear specifications, but used in a disciplined way since the context *is* actually ordered.

In our particular example, the configuration would look like

$$\begin{array}{l}\mathsf{eval}(M, d_{n+1}, d_n) \ \mathsf{cont}(K_n, d_n, d_{n-1}) \ \ldots \ \mathsf{cont}(K_1, d_1, d_0) \\ \mathsf{retn}(T, d_{n+1}, d_n) \ \mathsf{cont}(K_n, d_n, d_{n-1}) \ \ldots \ \mathsf{cont}(K_1, d_1, d_0)\end{array}$$

We can rearrange and optimize slightly, noting, for example, that we never need d_{n+1} and we use

$$\begin{array}{l}\mathsf{eval}(M, d) \\ \mathsf{retn}(T, d) \\ \mathsf{cont}(d, K, d')\end{array}$$

As a sample, we given the rules for functions, first in their ordered form and then in destination passing style. Note that the rules for applications must introduce a fresh destination.

Ordered	Destination-Passing
$\dfrac{\mathsf{eval}(M\ V)}{\mathsf{eval}(M) \quad \mathsf{cont}(\_\ V)} \to C_1$	$\dfrac{\mathsf{eval}(M\ V, d)}{\mathsf{eval}(M, d') \quad \mathsf{cont}(d', \_\ V, d)} \to C_1^{d'}$
$\dfrac{\mathsf{eval}(\lambda x.\ M)}{\mathsf{retn}(\lambda x.\ M)} \to C_2$	$\dfrac{\mathsf{eval}(\lambda x.\ M, d)}{\mathsf{retn}(\lambda x.\ M, d)} \to C_2$
$\dfrac{\mathsf{retn}(\lambda x.\ M) \quad \mathsf{cont}(\_\ V)}{\mathsf{eval}([V/x]M)} \to C_3$	$\dfrac{\mathsf{retn}(\lambda x.\ M, d') \quad \mathsf{cont}(d', \_\ V, d)}{\mathsf{eval}([V/x]M, d)} \to C_3$

As a preview of CLF [WCPW02, CPWW02, WCPW04, SNS08, SN11], we show the relevant part of the file cbpv.clf which implements the above idea.

There are a few things we have not discussed, such as the *indexing* of values and computations by their positive or negative types. We first highlight the three rules on the right.

```
eval/app : eval (app M V) D
            -o {Exists d'. eval M d' * cont d' (app1 V) D}.
eval/lam : eval (lam (\!x. M !x)) D -o {retn (lam (\!x. M !x)) D}.
eval/app1 : retn (lam (\!x. M !x)) D' * cont D' (app1 V) D
             -o {eval (M !V) D}.
```

Because we index values and computations, the code below is not only an operational specification but also a type checker for call-by-push-value. We have limited ourselves to the *binary* forms of $\&$ and $+$ since label sets are not so easily represented.

```
% Call-by-push-value in CLF
% Pure function fragment with shifts

neg : type.
pos : type.

arrow : pos -> neg -> neg.   % A -> B
up : pos -> neg.             % up A
down : neg -> pos.           % down A

% values and computations, indexed by their type
val : pos -> type.
comp : neg -> type.

% negative types
% A -> B
lam : (val A -> comp B) -> comp (arrow A B).
app : comp (arrow A B) -> val A -> comp B.

% up A
return : val A -> comp (up A).
letval : comp (up A) -> (val A -> comp C) -> comp C.

% down A
thunk : comp A -> val (down A).
force : val (down A) -> comp A.
```

```
% runtime artefacts
dest : neg -> type.
frame : neg -> neg -> type.
app1 : val A -> frame (arrow A B) B.
letval1: (val A -> comp C) -> frame (up A) C.

% ssos predicates
eval : comp A -> dest A -> type.
retn : comp A -> dest A -> type.
cont : dest A -> frame A B -> dest B -> type.

% A -> B
eval/lam : eval (lam (\!x. M !x)) D -o {retn (lam (\!x. M !x)) D}.
eval/app : eval (app M V) D
            -o {Exists d'. eval M d' * cont d' (app1 V) D}.
eval/app1 : retn (lam (\!x. M !x)) D' * cont D' (app1 V) D
             -o {eval (M !V) D}.

% up A
eval/return : eval (return V) D -o {retn (return V) D}.
eval/letval : eval (letval M (\!x. N !x)) D
               -o {Exists d'. eval M d'
                   * cont d' (letval1 (\!x. N !x)) D}.
eval/letval1 : retn (return V) D' * cont D' (letval1 (\!x. N !x)) D
                -o {eval (N !V) D}.

% down A
eval/force : eval (force (thunk M)) D
             -o {eval M D}.

#query * 1 * 1
Pi d0. eval (lam (\!x. return x)) d0 -o {retn M d0}.

#query * 1 * 1
Pi d0. eval (app (lam (\!x. return x)) (thunk (lam (\!y. return y)))) d0
 -o {retn M d0}.
```

Exercises

Exercise 1 Both call-by-value and call-by-name lead to code that is considerably more complex than it needs to be, including, for example, patterns such as $\mathsf{let\ val}\ x' = \mathsf{return}\ x\ \mathsf{in}\ M$. These spurious introduction/elimination forms are called *administrative redices*. Begin by showing an example of an expression whose call-by-value translation contains an administrative redex.

If possible, rewrite the call-by-value translation using two different forms, one resulting directly in a value the other in a computation.

$$\Gamma^+ \vdash (E)^p : (A)^+$$
$$\Gamma^+ \vdash (E)^n : \uparrow(A)^+$$

calling upon the appropriate translation form. Try to write the refined translation so that no administrative redices arise. If this does not work, do you see another approach to avoiding administrative redices?

Exercise 2 Carry out Exercise 1 for call-by-name.

Exercise 3 Investigate a *linear call-by-push-value* combined with Levy's by an adjunction with two shifts. Explore the expressive power of the result. Does linearity describe and interesting and useful properties of functional computation?

Exercise 4 Using a substitution-free operational semantics as in Exercise L21.1, specify a *call-by-need* operational semantics. Can you do this on call-by-push-value in general, or should it be integrated somehow (or described directly) on call-by-name?

References

[CPWW02] 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.

[Lev01] Paul Blain Levy. *Call-by-Push-Value*. PhD thesis, University of London, 2001.

[SN11] Anders Schack-Nielsen. *Implementing Substructural Logical Frameworks*. PhD thesis, IT University of Copenhagen, January 2011.

[SNS08] 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.

[SP11] Robert J. Simmons and Frank Pfenning. Logical approximation for program analysis. *Higher-Order and Symbolic Computation*, 24(1–2):41–80, 2011.

[WCPW02] 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.

[WCPW04] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. Specifying properties of concurrent computations in CLF. In C.Schürmann, editor, *Proceedings of the 4th International Workshop on Logical Frameworks and Meta-Languages (LFM'04)*, Cork, Ireland, July 2004. Electronic Notes in Theoretical Computer Science (ENTCS), vol 199, pp. 133–145, 2008.

Lecture Notes on
The Concurrent Logical Framework

15-816: Substructural Logics
Frank Pfenning

Lecture 23
November 17, 2016

A *logical framework* is a metalanguage for the representation and analysis of deductive systems such as logics, type systems, specifications of operational semantics, etc. The goal is to distill the essence of deductive systems so that encodings are as direct and natural as possible. In many ways one can consider them *normative* in that they embody the judgmental principles upon which the design of logics and programming languages are (or ought to be) based on.

An early logical framework was LF [HHP87, HHP93], implemented in the Twelf system [PS99] which is based on a minimal structural dependent type system λ^{Π}. It elucidated and crystallized the notions of bound variable, capture-avoiding substitution, hypothetical judgment, and generic judgment. The high level nature of the encodings allowed automatic and programmatic theorem proving [Sch00] as well as execution of some specifications as backward-chaining logic programs [MN86, Pfe91].

It was recognized early on that substructural logics and related programming languages could not be represented as directly in LF and related frameworks such as λProlog [MN86] as one might hope. Essentially, early frameworks did not support linear hypothetical judgments directly, which hampered encodings. This was addressed in a line of research on substructural linear [HM94, Mil94, Chi95, CP96, CP02] and ordered [PP99b, PP99a, Pol01] logical frameworks, eventually culminating in the Concurrent Logical Framework (CLF) [WCPW02, CPWW02, WCPW04] and its implementation in Celf [SNS08].

CLF is expressive and robust enough to allow logic programming [LPPW05] but metatheoretic reasoning in the style of Twelf remains elusive (see [Ree09]

for one approach). In this lecture we focus on the *positive fragment* of CLF, applying a bit of hindsight to polarize the original presentation. This fragment is of particular interest since its forward-chaining operational semantics allows us to represent the deductive systems we have analyzed in this course in a high-level and executable manner.

1 The Positive Fragment of CLF

CLF arises from the polarized adjoint formulation of intuitionistic linear logic by admitting dependent types. We will largely downplay this aspect of the CLF, since it is rich enough to be the subject of its own course [Pfe92]. Instead we emphasize the substructural aspects of the framework. Before launching into its description, we should emphasize that we are interested almost exlusively in focused, cut-free proofs. It is terms representing these proofs that end up being in bijective correspondence with the objects we would like to represent.

By default, layers of the syntax are *linear*, so we will only annotate types that are structural as A_{U}.

$$
\begin{array}{lcll}
A_{\mathsf{U}}^+ & ::= & p_{\mathsf{U}}^+ \mid \ldots & \\
A^- & ::= & A^+ \multimap B^- \mid \Pi x{:}A_{\mathsf{U}}^+ . B^- \mid A^- \mathbin{\&} B^- \mid \uparrow A^+ & \text{(but not } p^-) \\
A^+ & ::= & p^+ \mid A^+ \otimes B^+ \mid \mathbf{1} \mid \exists x{:}A_{\mathsf{U}}^+ . B^+ & \text{(but not } \downarrow A^-)
\end{array}
$$

A few remarks on these types. We do not include negative atoms (p^-) or $\downarrow A^-$, which constitutes our restriction to the negative fragment. We omitted disjunction $A^+ \oplus B^+$ because we have not carried out the theory to understand what true concurrency would mean, something we discuss in the next lecture. We have left open what kinds of propositions we would have in the structural layer. Positive atoms p_{U}^+ are useful because they correspond to the persistent propositions we have used in various representations.

Note that universal ($\Pi x{:}A_{\mathsf{U}}^+ . B^+$) and existential ($\exists x{:}A_{\mathsf{U}}^+ . B^+$) are constructs of mixed mode, combining structural and linear types into a linear type. This appears to be necessary: while one can give formalistic constructions of a linear dependent function space, there is to date no fully satisfactory account of it. The reason lies in the question of what constitutes a "linear use" of x in a hypothetical linear Π, as compared to simply a "mention" of x in the type. In practice, we have developed a number of techniques to circumvent the need for linear dependent functions, mostly by splitting the name x (which is persistent) from a linear capability which

explictly marks uses x. The complications, by the way, are not specific to linear logic but appear in the literature of modal logic in various forms even just for first-order modal logic.

The proof term assignment to this calculus turns out to be quite a bit different for call-by-push-value or for SILL, both of which were similarly polarized. Here, we are interested in representing only cut-free, focused proofs because these are used for representation. For starters, in call-by-push-value we had two forms of terms: computations (of negative type) and values (of positive type). Here we will have five different forms of terms, corresponding to right inversion and left focusing (negative types), left inversion and right focusing (positive types) and one for neutral sequents, before a focusing phase is started. We introduce the terms in stages, but first all five judgments. We use Γ for positive structural contexts, Δ for linear antecedents, and Ω^+ for linear antecedents presented in an ordered fashion so that inversion is deterministic.

$\Gamma\,;\,\Delta \vdash M : A^-$	right inversion
$\Gamma\,;\,\Delta\,;\,\Omega^+ \vdash J$	left inversion
$\Gamma\,;\,\Delta, [R : A^-] \vdash E : C^+$	left focusing
$\Gamma\,;\,\Delta \vdash [V : C^+]$	right focusing
$\Gamma\,;\,\Delta \vdash E : C^+$	stable sequent
$\Gamma \vdash [V_{\mathsf{U}} : A^+_{\mathsf{U}}]$	structural right focus

In left inversion, the judgment J on the right could be either $M : A^-$ or $E : C^+$.

Right Inversion. For right inversion, the assignment is straightforward, consistent with our call-by-push-value functional language, even though we are operating in a sequent calculus here. The judgment for right inversion is $\Gamma\,;\,\Delta \vdash M : A^-$.

$$\frac{\Gamma\,;\,\Delta\,;\,p : A^+ \vdash M : B^-}{\Gamma\,;\,\Delta \vdash \lambda p.\, M : A^+ \multimap B^-}\ \multimap R \qquad \frac{\Gamma, x{:}A^+_{\mathsf{U}}\,;\,\Delta \vdash M : B^-}{\Gamma\,;\,\Delta \vdash \lambda x.\, M : \Pi x{:}A.\, B^-}\ \Pi R$$

$$\frac{\Gamma\,;\,\Delta \vdash M : A^- \quad \Gamma\,;\,\Delta \vdash N : B^-}{\Gamma\,;\,\Delta \vdash \langle M, N\rangle : A^- \,\&\, B^-}\ \&R \qquad \frac{\Gamma\,;\,\Delta \vdash E : A^+}{\Gamma\,;\,\Delta \vdash \{E\} : \uparrow A^+}\ \uparrow R$$

In the final rule we transition to the stable judgment, where all declarations in Δ are either $x : A^-$ or $x : p^+$. For Γ, we only consider $x^+_{\mathsf{U}} : p^+_{\mathsf{U}}$, which is also stable.

Left Inversion. Left inversion operates on an ordered context Ω with propositions $p : A^+$ where p is a *pattern* (not an atomic type). When the context is empty and all inversion steps have been applied, we return to the judgment J.

$$\frac{\Gamma\,;\,\Delta, x{:}p^+\,;\,\Omega \vdash J}{\Gamma\,;\,\Delta\,;\,(x : p^+)\,\Omega \vdash J}\ \mathsf{atm}^+ \qquad \frac{\Gamma\,;\,\Delta\,;\,(p : A^+)\,(q : B^+)\,\Omega \vdash J}{\Gamma\,;\,\Delta\,;\,([p, q] : A^+ \otimes B^+)\,\Omega \vdash J}\ \otimes L$$

$$\frac{\Gamma\,;\,\Delta\,;\,\Omega \vdash J}{\Gamma\,;\,\Delta\,;\,([\,] : \mathbf{1})\,\Omega \vdash J}\ \mathbf{1}L \qquad \frac{\Gamma, x{:}A^+_{\mathsf{U}}\,;\,\Delta\,;\,(q : B^+)\,\Omega \vdash J}{\Gamma\,;\,\Delta\,;\,([x_{\mathsf{U}}, q] : \exists x_{\mathsf{U}} : A^+_{\mathsf{U}}.\, B^+)\,\Omega \vdash J}\ \exists L$$

$$\frac{\Gamma\,;\,\Delta \vdash J}{\Gamma\,;\,\Delta\,;\,\cdot \vdash J}\ \mathsf{empty}$$

Left Focus. When thinking about left focus, we have to think about the *signature* Σ which contains (in our case) constants $c : A^-_{\mathsf{L}}$, arbitrarily reusable. Strictly speaking, there should be a shift here, but we dispense with that due to the special case of the global signature.

$$\frac{(c{:}A^- \in \Sigma) \quad \Gamma\,;\,\Delta, [c : A^-] \vdash E : C^+}{\Gamma\,;\,\Delta \vdash E : C^+}\ \mathsf{foc}^-_{\mathsf{U}} \qquad \frac{\Gamma\,;\,\Delta, [x : A^-] \vdash E : C^+}{\Gamma\,;\,\Delta, x{:}A^- \vdash E : C^+}\ \mathsf{foc}^-_{\mathsf{L}}$$

$$\frac{\Gamma\,;\,\Delta \vdash [V : A^+] \quad \Gamma\,;\,\Delta, [R\,V : B] \vdash E : C^+}{\Gamma\,;\,\Delta, \Delta', [R : A^+ \multimap B^-] \vdash E : C^+}\ \multimap L$$

$$\frac{\Gamma \vdash [V_{\mathsf{U}} : A^+_{\mathsf{U}}] \quad \Gamma\,;\,\Delta, [R\,V_{\mathsf{U}} : [V_{\mathsf{U}}/x]B^-] \vdash E : C^+}{\Gamma\,;\,\Delta, [R : \Pi x{:}A^+_{\mathsf{U}}.\, B^+] \vdash E : C^+}\ \Pi L$$

$$\frac{\Gamma\,;\,\Delta, [\pi_1\,R : A^-] \vdash E : C^+}{\Gamma\,;\,\Delta, [R : A^- \,\&\, B^-] \vdash E : C^+}\ \&L_1 \qquad \frac{\Gamma\,;\,\Delta, [\pi_2\,R : B^-] \vdash E : C^+}{\Gamma\,;\,\Delta, [R : A^- \,\&\, B^-] \vdash E : C^+}\ \&L_2$$

$$\frac{\Gamma\,;\,\Delta\,;\,p : A^+ \vdash E : C^+}{\Gamma\,;\,\Delta, [R : \uparrow A^+] \vdash \mathsf{let}\ \{p\} = R\ \mathsf{in}\ E : C^+}\ \uparrow L$$

The last rule here represents a transition to the left inversion judgment.

Right Focus. Finally, we come to right focus which, in the positive fragment, will always either succeed and finish the proof or fail. Since the pos-

itive fragment lacks $\downarrow A^-$ we cannot lose focus.

$$\frac{\Gamma\ ;\ \Delta \vdash [E : C^+]}{\Gamma\ ;\ \Delta \vdash E : C^+}\ \mathsf{foc}^+ \qquad \frac{}{\Gamma\ ;\ x{:}p^+ \vdash [x : p^+]}\ \mathsf{id}^+$$

$$\frac{\Gamma\ ;\ \Delta \vdash [V : A] \quad \Gamma\ ;\ \Delta \vdash [W : B]}{\Gamma\ ;\ \Delta, \Delta' \vdash [[V, W] : A \otimes B]}\ \otimes R \qquad \frac{}{\Gamma\ ;\ \cdot \vdash [[\,] : \mathbf{1}]}\ \mathbf{1}R$$

$$\frac{\Gamma \vdash [V_{\mathsf{U}} : A^+_{\mathsf{U}}] \quad \Gamma\ ;\ \Delta \vdash [W : [V_{\mathsf{U}}/x]B^+]}{\Gamma\ ;\ \Delta \vdash [[V_{\mathsf{U}}, W] : \exists x{:}A^+_{\mathsf{U}}.\, B^+]}\ \exists R$$

Structural Right Focus. Since in our language at the moment we only consider atomic structural propositions, we only have one rule.

$$\frac{}{\Gamma, x{:}p^+_{\mathsf{U}}\ ;\ \cdot \vdash [x : p^+_{\mathsf{U}}]}\ \mathsf{id}^+_{\mathsf{U}}$$

2 Summary of Proof Terms

We obtain the following language of terms, where we indicate in each line the corresponding proposition and the concrete Celf syntax for the con-

struct.

		Abstract Syntax		Concrete Syntax	
		Term	Type	Term	Type
M	$::=$	$\lambda p.\, M$	$A^+ \multimap B^-$	`\p. M`	`A -o B`
	$\mid$	$\lambda x_{\mathsf{U}}.\, M$	$\Pi x_{\mathsf{U}}{:}A^-_{\mathsf{U}}.\, B^-$	`\!x. M`	`Pi x:A. B`
	$\mid$	$\langle M, N\rangle$	$A^- \,\&\, B^-$	`< M , N >`	`A & B`
	$\mid$	$\{E\}$	$\uparrow A^+$	`{ E }`	`{ A }`
p	$::=$	x	p^+	`x`	
	$\mid$	$[p, q]$	$A^+ \otimes B^+$	`[p,q]`	`A * B`
	$\mid$	$[\,]$	$\mathbf{1}$	`1`	`1`
	$\mid$	$[x_{\mathsf{U}}, p]$	$\exists x_{\mathsf{U}}{:}A^+_{\mathsf{U}}.\, B^+$	`[!x,p]`	`Exists x:A. B`
R	$::=$	c	$c{:}A^- \in \Sigma$	`c`	
	$\mid$	x	$x{:}A^- \in \Delta$	`x`	
	$\mid$	$R\, V$	$A^+ \multimap B^-$	`R V`	`A -o B`
	$\mid$	$\pi_1\, R \mid \pi_2\, R$	$A \,\&\, B$	`#1 R  #2 R`	`A & B`
	$\mid$	$R\, V_{\mathsf{U}}$	$\Pi x_{\mathsf{U}}{:}A^+_{\mathsf{U}}.\, B^-$	`R !V`	`Pi x:A. B`
V	$::=$	x	p^+	`x`	
	$\mid$	$[V, W]$	$A^+ \otimes B^+$	`[V,W]`	`A * B`
	$\mid$	$[\,]$	$\mathbf{1}$	`1`	`1`
	$\mid$	$[V_{\mathsf{U}}, W]$	$\exists x_{\mathsf{U}}{:}A^+_{\mathsf{U}}.\, B^+$	`[!V,W]`	`Exists x{:}A. B`
E	$::=$	let $\{p\} = R$ in E	left focus	`let {p} = R in E`	`{ A }`
	$\mid$	V	right focus	`V`	

3 Example: Coin Exchange

We have already seen a significant transcription of inference rules into Celf in Lecture 22 on call-by-value and call-by-name.

Let's see CLF in action on a simpler example: the old coin exchange.[1]

```
q : type.
d : type.
n : type.

d2q : d * d * n -o { q }.
q2d : q -o { d * d * n }.
```

[1] Source at http://www.cs.cmu.edu/~fp/courses/15816-f16/misc/exchange.clf

```
n2d : n * n -o { d }.
d2n : d -o { n * n }.
```

We can now perform type-checking by using the form `c : A = M.` which verifies that term M has type A. Moreover, c stands for M in the remainder of the file. The first example is just one step, where we convert three nickels to a dime and a nickel.

```
example1 : n * n * n -o {d * n} =
  \[n1, [n2, n3]]. {             % n1:n, n2:n, n3:n |- _ : d * n
    let {d1} = n2d [n1, n3] in   % d1:n, n2:n       |- _ : d * n
     [d1, n2]                    % d1:n, n2:n       |- _ : d * n
  }.
```

We used, rather arbitrarily, the first and the third nickel to convert to a dime, leaving the last one in our possession. We showed, after each line, the antecedent and the succedent, omitting the proof terms.

We can also see if our forwarding chaining engine would find this proof. Actually it does not, because our forward chaining engine applies rules until quiescence. But since we can exchange coins back and forth, this specification (when viewed as a program) will never terminate. Once we put a bound on the number of steps to take, it depends on luck. In this case, with a bound of 11, it happens to succeed.

```
Query (11, 1, *, 1) (n * (n * n)) -o {d * n}.
Solution: \[X1, [X2, X3]]. {
    let {X4} = n2d [X1, X3] in
    let {[X5, X6]} = d2n X4 in
    let {X7} = n2d [X2, X5] in
    let {[X8, X9]} = d2n X7 in
    let {X10} = n2d [X8, X9] in
    let {[X11, X12]} = d2n X10 in
    let {X13} = n2d [X6, X11] in
    let {[X14, X15]} = d2n X13 in
    let {X16} = n2d [X12, X14] in
    let {[X17, X18]} = d2n X16 in
    let {X19} = n2d [X15, X17] in [X19, X18]}
Query ok.
```

We can clearly see in the proof that it displays, that it changed back-and-forth between two nickels and a time and stops forward chaining to

examine the goal after 11 iterations. It so happens that we do have one dime and one nickel at that point. Here is one more example, this time using the more reliable type-checking.

```
example2 : n * n * n * n * n -o { q } =
  \[n1, [n2, [n3, [n4, n5]]]]. {
    let {d1} = n2d [n1, n2] in
    let {d2} = n2d [n3, n4] in
    let {q0} = d2q [d1, [d2, n5]] in
    q0
  }.
```

Exercises

Exercise 1 Implement your choice of a finite state transducer like binary increment or compressing runs of b's as a forward-chaining concurrent logic program. You should use the technique of destinations to represent the ordered context linearly so that, for example, the character `a` might be represented as `msg L a R` where `L` and `R` represent destinations that tie this character to its left and right neighbors of the predicate representing the state of a transducer.

Exercise 2 In the style of Exercise 1, implement a pushdown automaton that recognizes a string of properly matched parentheses.

References

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

[CP96] 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.

[CP02] 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.

[CPWW02] 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.

[HHP87] 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.

[HHP93] 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.

[HM94] 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.

[LPPW05] 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.

[Mil94] Dale Miller. A multiple-conclusion meta-logic. In S. Abramsky, editor, *Ninth Annual Symposium on Logic in Computer Science*, pages 272–281, Paris, France, July 1994. IEEE Computer Society Press.

[MN86] Dale Miller and Gopalan Nadathur. Higher-order logic programming. In Ehud Shapiro, editor, *Proceedings of the Third International Logic Programming Conference*, pages 448–462, London, June 1986.

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[Pfe92] Frank Pfenning. Computation and deduction. Unpublished lecture notes, 277 pp. Revised May 1994, April 1996, May 1992.

[Pol01] Jeff Polakow. *Ordered Linear Logic and Applications*. PhD thesis, Department of Computer Science, Carnegie Mellon University, August 2001.

[PP99a] Jeff Polakow and Frank Pfenning. Natural deduction for intuitionistic non-commutative linear logic. In J.-Y. Girard, editor, *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications (TLCA'99)*, pages 295–309, L'Aquila, Italy, April 1999. Springer-Verlag LNCS 1581.

[PP99b] Jeff Polakow and Frank Pfenning. Relating natural deduction and sequent calculus for intuitionistic non-commutative linear logic. In Andre Scedrov and Achim Jung, editors, *Proceedings of the 15th Conference on Mathematical Foundations of Programming Semantics*, pages 449–466, New Orleans, Louisiana, April 1999. Electronic Notes in Theoretical Computer Science, Volume 20.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[Ree09] Jason C. Reed. *A Hybrid Logical Framework*. PhD thesis, Carnegie Mellon University, September 2009. Available as Technical Report CMU-CS-09-155.

[Sch00] 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.

[SNS08] 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.

[WCPW02] 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.

[WCPW04] 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
Concurrent Cost Semantics

15-816: Substructural Logics
Frank Pfenning

Lecture 24
November 29, 2016

In the last lecture we finally formally introduced the Concurrent Logical Framework (CLF) [WCPW02, CPWW02] and its implementation in Celf [SN11]. In this lecture we will use CLF to develop a high level implementation of SILL, the core of a session-typed programming language. As we will see, with some thought, it turns out the CLF is an almost perfect vehicle for specifying SILL. This approach exhibits a perfect isomorphism with the CLF specification of the sequent calculus for linear logic [BBMS16] that has previously been presented on purely logical grounds [Pfe94, Ree09].

We then proceed to instrument our semantics with costs, to compute the work and span of concurrent computations which together can be seen as measuring the "parallel complexity" of the computation. In our setting, we count the total number of communication steps that have to be performed, but other cost measures can be derived in a similar manner.

1 Representing Channels and Process Expressions

At the outset, we assume the following types (to be revised later):

```
ch : type.
exp : type.
```

where `ch` represents channels and `exp` represents process expressions. Channels should remain abstract, as in our previous encoding, which means the type `ch` is inhabited only by parameters that are introduced during the computation. Expressions represent concurrent programs.

At the level of our linear inference semantics, we write $\mathsf{proc}(c, P)$ for the state of a process computing P and providing a service along c. This will be represented here as `proc P C` for an expression `P` and channel `C`, so we have

```
proc : exp -> ch -> type.
```

Note that we "curry" our propositions, so there are no explicit parentheses around the arguments of a predicate. The other point to note is the `proc P C` is a *type*, rather than a proposition. This is because we are working in a type theory, so our process configuration will look like

```
p1 : proc P1 C1, p2 : proc P2 C2, ..., pn : proc Pn Cn
```

where each `p` represents a means to reference the process when describing the computation. For now, it is perfectly sensible to just think of it as a proposition.

Next, we consider a simple program

$$w{:}A \vdash \mathsf{send}\ c\ w\ ;\ \mathsf{close}\ c :: (c : A \otimes \mathbf{1})$$

where A is arbitrary and therefore a propositional variable. The first idea of representation would be

$$\ulcorner \mathsf{send}\ c\ w\ ;\ \mathsf{close}\ c \urcorner = \texttt{send C W (close C)}$$

which would give us the straightforward types

```
send : ch -> exp -> exp.
close : ch -> exp.
```

While these types are workable, they are not fully satisfactory as we will see, and do not fully exploit the expressive power of the framework.

On the receiving side, we might have a matching process

$$c{:}A \otimes 1 \vdash y \leftarrow \mathsf{recv}\ c\ ;\ \mathsf{wait}\ c\ ;\ P_y :: (d : D)$$

The receive construct binds the variable y with scope $\mathsf{wait}\ c\ ;\ P_y$. We represent is by a corresponding binder with corresponding scope in the logical framework. Recall that in CLF, the binders are represented with a λ-abstraction and written as `\x. M`. Using this idea we obtain

$$\ulcorner y \leftarrow \mathsf{recv}\ c\ ;\ \mathsf{wait}\ c\ ;\ P_y \urcorner = \texttt{recv C (\y. wait C (P y))}$$

An interesting part of this representation is that we indicate the possible dependence of P_y on y by writing `P y`, which means that `P` will have type `ch -> exp`. Overall, this gives us

```
recv : ch -> (ch -> exp) -> exp.
wait : ch -> exp -> exp.
```

Again, this is servicable, but uninspired. Why? Note that our process language is *linearly typed* and in fact we have seen it as being in a Curry-Howard correspondence with the linear sequent calculus! But our representation above is *not* linearly typed. In both the sender and the receiver example expressions

```
send C W (close C)          % sender
recv C (\y. wait C (P y))   % receiver
```

the channel `C` is apparently not linear. This means that we can write bogus expressions, namely programs that use their channels not linearly and they will type-check in the framework, which is unfortunate.

We will sharpen our representation so that *only* process expression that are properly linear will be well-typed in the framework. We use different techniques for the provider and the client of a channel, although other choices are certainly possible. On the provider side, we note that an executing process

```
proc (send C W (close C)) C
```

has a lot of redundancy, because the channel `C` along which we communicate is mentioned multiple times. What we do instead is for the provider expressions to leave the provider channel (here `C`) *implicit*, so the above becomes

```
proc (send_ W (close_)) C
```

where the underscore suffix in the name of the `send_` and `close_` are there to remind us that they implicitly refer to the channel that is provided. With that, we can then use linear typing:

```
send_ : ch -o exp -o exp.
close_ : exp.
```

To be formal, the representation function now takes a parameter c (the channel along which the process provides) so we can recognize the appropriate syntactic form.

$$\begin{array}{lcl} \ulcorner \mathsf{send}\ c\ w\ P \urcorner^c & = & \texttt{send\_ W } \ulcorner P \urcorner^c \\ \ulcorner \mathsf{close}\ c \urcorner^c & = & \texttt{close\_} \end{array}$$

Unfortunately, on the client side this particular device fails. This is because a client can use many different processes, and the name of a channel is critical to identify which channel we want to communicate with. So how do we deal with the apparent non-linearity of `C` in an expression such as

```
recv C (\y. wait C (P y))
```

which uses `C` twice? A clue to the answer was provided many lectures ago when we gave the *asynchronous semantics*. Recall that, for example

$$\frac{\mathsf{proc}(c, \mathsf{send}\ c\ w; P)}{\mathsf{proc}(c', P) \qquad \mathsf{msg}(c, \mathsf{send}\ c\ w\ ;\ c \leftarrow c')}\ \otimes C^{c'}$$

where c' is a freshly chosen continuation channel. So we'll use this idea here, even though for now our semantics is synchronous: sending will create a fresh continuation channel for further communication. We bake this into our representation, rather than using it only as a feature of our semantics. So we define

$$\ulcorner y \leftarrow \mathsf{recv}\ c\ ;\ Q_y \urcorner^d \quad = \quad \texttt{recv C (\y. \c.} \ulcorner Q_y \urcorner^d \texttt{ y c)}$$

Of course, $\mathsf{wait}\ c$ does not receive a continuation channel, since the associated process has terminated. So we get

$$\ulcorner \mathsf{wait}\ c\ ;\ Q \urcorner^d \quad = \quad \texttt{wait C } \ulcorner Q \urcorner^d$$

These constructs can now be linearly typed

```
recv : ch -o (ch -o ch -o exp) -o exp.
wait : ch -o exp -o exp.
```

Let's think about forwarding and spawning. Within a process, forwarding is

$$\mathsf{proc}(c, c \leftarrow d)$$

which means that forwarding has c as an implicit argument. Similarly, spawning

$$\mathsf{proc}(d, x \leftarrow P_x\ ;\ Q_x)$$

creates a new channel c, which is provided by P_c (and therefore implicit in P_c and used by Q_c, where it is explicit. This means we have

$$\begin{array}{lcl} \ulcorner c \leftarrow d \urcorner^c & = & \texttt{fwd\_ D} \\ \ulcorner x \leftarrow P_x\ ;\ Q_x \urcorner^d & = & \texttt{spawn\_} \ulcorner P_x \urcorner^x \texttt{ (\x.} \ulcorner Q_x \urcorner^d \texttt{ x)} \end{array}$$

from which we can read off the following linear types

```
fwd : ch -o exp.
spawn : exp -o (ch -o exp) -o exp.
```

In summary, for the constructs implementing forward, spawn, $A \otimes B$, and $\mathbf{1}$, we have the following representation function where we assume that channels c in the programming notation are translated to variables C with the same name in the logical framework:

$\ulcorner c \leftarrow d \urcorner^c$	$=$	`fwd_ D`	
$\ulcorner x \leftarrow P_x \,;\, Q_x \urcorner^c$	$=$	`spawn_ P (\x. Q x)`	where $\ulcorner P_x \urcorner^x = \mathtt{P}, \ulcorner Q_x \urcorner^c = \mathtt{Q}$
$\ulcorner \mathsf{send}\ c\ w\ P \urcorner^c$	$=$	`send_ W P`	where $\ulcorner P \urcorner^c = \mathtt{P}$
$\ulcorner y \leftarrow \mathsf{recv}\ c \,;\, Q_y \urcorner^d$	$=$	`recv C (\y. \c. Q y c)`	where $\ulcorner Q_y \urcorner^d = \mathtt{Q}$
$\ulcorner \mathsf{close}\ c \urcorner^c$	$=$	`close_`	
$\ulcorner \mathsf{wait}\ c \,;\, Q \urcorner^d$	$=$	`wait C Q`	where $\ulcorner Q \urcorner^d = \mathtt{Q}$

This gives rise to the following *linear* types for the process constructors:

```
ch : type.
exp : type.

fwd_ : ch -o exp.
spawn_ : exp -o (ch -o exp) -o exp.

send_ : ch -o exp -o exp.
recv : ch -o (ch -o ch -o exp) -o exp.

close_ : exp.
wait : ch -o exp -o exp.
```

2 Intrinsic Typing

At this point we achieved a partial victory: only process expressions which treat channels linearly will be well-typed in the framework, providing a modicum of correctness checking. However, many meaningless process expressions can still be represented. For example,

$$x \leftarrow \mathsf{close}\ x \,;\, y \leftarrow \mathsf{recv}\ x \,;\, \mathsf{wait}\ x \,;\, d \leftarrow y$$

does not make much sense since close x is matched up with a recv x instead of a wait. And yet, its translation

```
example1 : exp =
spawn_ (close_) (\x. recv x (\y. \x. wait x (fwd_ y))).
```

type-checks perfectly.

Of course, the problem is that the translation *enforces linearity* but does *not enforce types*. If we want to achieve this additional amount of precision, we need to *index* both channels and expressions with their session types. Fortunately, this can be done quite easily. We don't even need to change the representation function, just make their CLF types more precise. To start with we define (on our small fragment so far):

```
tp : type.
tensor : tp -> tp -> tp.
one : tp.

ch : tp -> type.
exp : tp -> type.
```

Processes `proc P C` define a process of type A that offers a channel of type A, so we have

```
proc : exp A -> proc A -> type.
```

This declaration is schematic over A and Celf type reconstruction will determine A wherever it sees `proc P C` from `P` and `C`.

For the language constructs, we just need to read the type indices off the typing rules. We show a few examples, starting with identity.

$$\frac{}{d{:}A \vdash c \leftarrow d :: (c : A)}\ \mathsf{id}$$

Recall that $\ulcorner c \leftarrow d \urcorner^c =$ `fwd_ D` which means that

```
fwd : ch A -o exp A.
```

Here, the first `A` comes from the type of the channel d, while the expression comes from the type of the (implicit) channel c.

Similarly

$$\frac{\Delta \vdash P_x :: (x : A) \quad \Delta', x{:}A \vdash Q_x :: (c : C)}{\Delta, \Delta' \vdash x \leftarrow P_x\ ;\ Q_x :: (c : C)}\ \mathsf{cut}$$

Since

$$\ulcorner x \leftarrow P_x\ ;\ Q_x \urcorner^c = \texttt{spawn\_}\ \ulcorner P_x \urcorner^x\ \texttt{(\x.}\ \ulcorner Q_x \urcorner^c\ \ \texttt{x)}$$

we obtain

```
spawn_ : exp A -o (ch A -o exp C) -o exp C.
```

As a last detailed example, let's look at receiving by the client.

$$\frac{\Delta, y{:}A, x{:}B \vdash Q_y :: (c : C)}{\Delta, x{:}A \otimes B \vdash y \leftarrow \mathsf{recv}\ x\ ;\ Q_y :: (c : C)}\ \otimes L$$

with

$$\ulcorner y \leftarrow \mathsf{recv}\ x\ ;\ Q_y \urcorner^c = \texttt{recv X (\y. \x.} \ulcorner Q_y \urcorner^c \texttt{ y x)}$$

```
recv : ch (tensor A B) -o (ch A -o ch B -o exp C) -o exp C.
```

Summarizing all the types so far, we have

```
tp : type.
tensor : tp -> tp -> tp.
one : tp.

ch : tp -> type.
exp : tp -> type.

fwd_ : ch A -o exp A.
spawn_ : exp A -o (ch A -o exp C) -o exp C.

send_ : ch A -o exp B -o exp (tensor A B).
recv : ch (tensor A B) -o (ch A -o ch B -o exp C) -o exp C.

close_ : exp one.
wait : ch one -o exp C -o exp C.

proc : exp A -> ch A -> type.
```

Now our previous example

```
example1 : exp =
spawn_ (close_) (\x. recv x (\y. \x. wait x (fwd_ y))).
```

will no longer type-check, but fail with the (slightly edited, replacing two variables with underscores) message

```
Type-checking failed in declaration of example1 on line 17:
Unification failed: Constants tensor and one differ
Object 1 has type:
ch one
but expected:
ch (tensor _ _)
```

3 Choice

Choice, whether external or internal is not significant, reveals a new challenge. We decide to only implement binary choice just to keep the encoding as straightforward as possible. We use internal choice as our example. We have the following constructs

$$\begin{array}{l} c.\pi_1 \,;\, P \\ c.\pi_2 \,;\, P \\ \mathsf{case}\ c\ (\pi_1 \Rightarrow Q_1 \mid \pi_2 \Rightarrow Q_2) \end{array}$$

Since $A \oplus B$ is also positive, the sending constructs use the provider channel and therefore have an implicit argument. We use `select1_` and `select2_` as our concrete names for the two sending constructs.

$$\begin{array}{lcl} \ulcorner c.\pi_1 \,;\, P \urcorner^c & = & \texttt{select1\_}\ \ulcorner P \urcorner^c \\ \ulcorner c.\pi_2 \,;\, P \urcorner^c & = & \texttt{select2\_}\ \ulcorner P \urcorner^c \\ \ulcorner \mathsf{case}\ c\ (\pi_1 \Rightarrow Q_1 \mid \pi_2 \Rightarrow Q_2) \urcorner^d & = & \texttt{case C (\textbackslash c.}\ \ulcorner Q_1 \urcorner^d\ \texttt{c) (\textbackslash c.}\ \ulcorner Q_2 \urcorner^d\ \texttt{c)} \quad \texttt{??} \end{array}$$

The problem here is the last line. Let's examine the typing rule.

$$\frac{\Delta, c{:}A \vdash Q_1 :: (d : D) \quad \Delta, c{:}B \vdash Q_2 :: (d : D)}{\Delta, c{:}A \oplus B \vdash \mathsf{case}\ c\ (\pi_1 \Rightarrow Q_1 \mid \pi_2 \Rightarrow Q_2) :: (d : D)}\ \oplus L$$

Since $\oplus$ is an additive connective, all channels are propagated into both branches of the case construct. In the first attempted encoding above, however, the context will be split between $\ulcorner Q_1 \urcorner^d$ and $\ulcorner Q_2 \urcorner^d$. So we need to exploit that the framework also has an additive connective, namely external choice $A^- \mathbin{\&} B^-$, with the proof term being a pair $\langle M, N \rangle$.

$$\begin{array}{lcl} \ulcorner c.\pi_1 \,;\, P \urcorner^c & = & \texttt{select1\_}\ \ulcorner P \urcorner^c \\ \ulcorner c.\pi_2 \,;\, P \urcorner^c & = & \texttt{select2\_}\ \ulcorner P \urcorner^c \\ \ulcorner \mathsf{case}\ c\ (\pi_1 \Rightarrow Q_1 \mid \pi_2 \Rightarrow Q_2) \urcorner^d & = & \texttt{case C <(\textbackslash c.}\ \ulcorner Q_1 \urcorner^d\ \texttt{c),(\textbackslash c.}\ \ulcorner Q_2 \urcorner^d\ \ \texttt{c)>} \end{array}$$

which yields the types

```
select1_ : exp A -o exp (plus A B).
select2_ : exp B -o exp (plus A B).
case : ch (plus A B) -o (ch A -o exp C) & (ch B -o exp C) -o exp C.
```

4 Operational Semantics: Forward and Spawn

Ideally, at the highest level of abstraction, we would like

```
c/fwd : proc (fwd_ D) C -o { C = D }.
```

that is, `C` and `D` are globally identified. Unfortunately, we omitted equality as a type constructor in CLF, so we need a different idea. The the simplest seems to be to actually synchronize with the provider of `D` and relabel it to become the provider of `C`. Since our calculus is linear, there will be exactly one provider of `D`, so this does not create any ambiguity or race conditions.

```
c/fwd : proc P D * proc (fwd_ D) C -o { proc P C }.
```

The forwarding process itself of course terminates in this step. We can see how the decision to leave the providing channel implicit helps here: if `P` referred to the channel `D` multiple times as the concrete process syntax does, then we would actually have to substitute `C` for `D` throughout `P`, which is a somewhat complex operation. In the rule above, all implicit references are now to `C`, where they previously referenced `D`.

For process spawn we need to create a fresh channel and start a new process. As usual, we use existential quantification in the framework to create a fresh parameter.

```
c/spawn : proc (spawn_ P (\x. Q x)) C
           -o { Exists a. proc P a * proc (Q a) C }.
```

We see that `P` provides along the new channel `a`, while `(Q a)` is a client of `a`.

5 Operational Semantics: Communication

The synchronous semantics is now a straightforward transcription of our inference rule, taking care of creating and using continuation channels. As we did for the description of the operational semantics we use the notation `c'` for the continuation channel of `c`.

```
c/tensor : proc (send_ W P) C * proc (recv C (\x. \c'. Q x c')) D
        -o { Exists c'. proc P c' * proc (Q W c') D }.
```

In the right-hand side of this rule, we write `Q W c'` to substitute `W` for `x` and the continuation channel `c'` for the bound variable `c'`. This takes advantage of β-reduction at the framework level to implement the name-for-name substitution at the process level.

For **1**, there will be no continuation channel so we just synchronize the `close_` with the matching `wait`.

```
c/one : proc (close_) C * proc (wait C Q) D
        -o { proc Q D }.
```

There are two rules for $A \oplus B$, depending on whether the first or the second branch is selected.

```
c/plus1 : proc (select1_ P) C * proc (case C <(\c'. Q1 c'),(\c'. Q2 c')>) D
         -o { Exists c'. proc P c' * proc (Q1 c') D }.
c/plus2 : proc (select2_ P) C * proc (case C <(\c'. Q1 c'),(\c'. Q2 c')>) D
         -o { Exists c'. proc P c' * proc (Q2 c') D }.
```

Note that even though no types are mentioned, these rules are type-checked, so both linearity and session-typing must at least be consistent. Of course, we can still make mistakes that will pass this test. For example, if we replace `Q2` by `Q1` in the `c/plus2` rule, CLF type reconstruction will still succeed by giving both branches the same type. We would need formal metatheory to catch such an error, but there is currently no tool to support such an activity.

We can try our semantics on some simple example: spawning a process of type **1** which just closes, in parallel with one that just waits for that to finish.

```
#query * 1 * 1
Pi c0. proc (spawn_ (close_) (\c1. wait c1 (close_))) c0 -o {proc P c0}.
```

The Celf directive `#query * 1 * 1` means that we run without bound, expecting 1 solution, looking for arbitrarily many, and running the query only once. We write `Pi c0.` to create a fresh initial channel `c0`. On the right hand side of this negative type, we have `proc P c0` which will test if the final configuration (that is, the one where we have quiescence) consists of a single process offering along `c0`. It will show this process, which we expect to be `close_`. We get:

```
Solution: \!c0. \X1. {
    let {[!a, [X2, X3]]} = c/spawn X1 in
    let {X4} = c/one [X2, X3] in X4}
 #P = close_
Query ok.
```

The resulting proof term is a representation of the computation of the process expression above, that is, a sequent of configurations. Let's write in the state of the configuration at each line as a comment with the types. We mark persistent variables with an exclamation mark `!`.

```
Solution: \!c0. \X1. {                  % !c0:ch one, X1:proc (spawn_ ...)
    let {[!a, [X2, X3]]} = c/spawn X1 in % !c0:ch one, !a:ch one,
                                        %    X2:proc (close_) a, X3:proc (wait a ...) c0
    let {X4} = c/one [X2, X3]           % !c0:ch one, !a:ch one, X4:proc (close_) c0
    in X4}
 #P = close_
Query ok.
```

We see that while expressions are typed linearly, channels are *not* linear in configurations. This is not directly possibly, since channels appear in index positions of processes that provide or use them. However, the process that provides each channel is treated as a linear resource.

Let's write one more example, which represents a kind of negation, where we think of $\mathsf{bool} = \mathbf{1} \oplus \mathbf{1}$. The for $c : \mathsf{bool}$ we think of $c.\pi_1$; $\mathsf{close}\ c$ as true and $c.\pi_2$; $\mathsf{close}\ c$ as false. The following program spawns a process which behaves like false, which is then negated by its client.

$$
\begin{array}{l}
c_1 \leftarrow (c_1.\pi_2\ ;\ \mathsf{close}\ c_1)\ ; \\
\mathsf{case}\ c_1\ \ (\pi_1 \Rightarrow c_0.\pi_2\ ;\ \mathsf{close}\ c_0 \\
\qquad\quad \mid \pi_2 \Rightarrow c_0.\pi_1\ ;\ \mathsf{close}\ c_0)
\end{array}
$$

In the CLF encoding:

```
Pi c0. proc (spawn_ (select2_ close_)
            (\c1. case c1 <(\c2. wait c2 (select2_ close_)) ,
                          (\c3. wait c3 (select1_ close_))>)) c0
```

As expected, this will execute in 3 steps: one spawn, one select, and one close, and end up as a process wishing to send π_1 and then closing the channel c_0.

```
Solution: \!c0. \X1. {
    let {[!a, [X2, X3]]} = c/spawn X1 in
    let {[!c', [X4, X5]]} = c/plus2 [X2, X3] in
    let {X6} = c/one [X4, X5] in X6}
 #P = select1_ close_
Query ok.
```

The signature and queries from this section can be found with the course materials[1]. The completion with the remaining connectives $A \multimap B$ and $A \mathbin{\&} B$ is also available online[2].

6 An Asynchronous Semantics

The representation and typing of all channels and process expressions remains the same when we want to give an asynchronous semantics, but we have two propositions $\mathsf{proc}(c, P)$ and $\mathsf{msg}(c, P)$ to define the operational semantics. Only certain kinds of messages are permitted, but we will not formalize this within the judgments.

First, spawn does not change, but a forwarding process interacts now with a message rather than a process. Because our language fragment has only positive connectives so far ($A \otimes B$, $\mathbf{1}$, $A \oplus B$) all messages come *from* the provider of `D`, so the rule can essentially stay the same. If we add negative propositions, forwarding may also need to interact with messages coming along `C` (see Exercise 2).

```
proc : exp A -> ch A -> type.
msg : exp A -> ch A -> type.

c/fwd : msg P D * proc (fwd_ D) C
         -o { msg P C }.
c/spawn : proc (spawn_ P (\x. Q x)) C
        -o { Exists a. proc P a * proc (Q a) C }.
```

The simple send now decomposes into two. As a reminder, we show the formulation using linear inference.

$$\frac{\mathsf{proc}(c, \mathsf{send}\ c\ w\ ;\ P)}{\mathsf{proc}(c', P) \quad \mathsf{msg}(c, \mathsf{send}\ c\ w\ ;\ c \leftarrow c')}\ \otimes C_s^{c'}$$

$$\frac{\mathsf{msg}(c, \mathsf{send}\ c\ w\ ;\ c \leftarrow c') \quad \mathsf{proc}(d, y \leftarrow \mathsf{recv}\ c\ ;\ Q_y)}{\mathsf{proc}(d, [c'/c]Q_w)}\ \otimes C_r$$

In CLF syntax, these become

[1] http://www.cs.cmu.edu/~fp/courses/15816-f16/misc/session/session-sync.clf

[2] http://www.cs.cmu.edu/~fp/courses/15816-f16/misc/session/session-complete-sync.clf

```
s/tensor : proc (send_ W P) C
        -o { Exists c'. proc P c' * msg (send_ W (fwd_ c')) C }.
r/tensor : msg (send_ W (fwd_ C')) C * proc (recv C (\x. \c'. Q x c')) D
        -o { proc (Q W C') D }.
```

We see that the decision to parameterize by a continuation channel works out well. Note that send creates the continuation channel, which is then received together with the channel w.

The remainder of the rules are divided analogously into send and receive rules.

```
s/one : proc (close_) C
        -o { msg (close_) C }.
r/one : msg (close_) C * proc (wait C Q) D
        -o { proc Q D }.

s/plus1 : proc (select1_ P) C
         -o { Exists c'. proc P c' * msg (select1_ (fwd_ c')) C }.
r/plus1 : msg (select1_ (fwd_ C')) C * proc (case C <(\c'. Q1 c'),(\c'. Q2 c')>) D
         -o { proc (Q1 C') D }.

s/plus2 : proc (select2_ P) C
         -o { Exists c'. proc P c' * msg (select2_ (fwd_ c')) C }.
r/plus2 : msg (select2_ (fwd_ C')) C * proc (case C <(\c'. Q1 c'),(\c'. Q2 c')>) D
         -o { proc (Q2 C') D }.
```

Now, for example, the second query (slightly modified to send true instead of false) ends in a configuration with no remaining process but two messages, transmitting π_2 followed by a close. We capture this when we examine the final configuration by expecting it to consist of two messages, one with a new destination (we couldn't predict what it is called, so we quantify over it) and a second one with the original destination.

```
#query * 1 * 1
Pi c0. proc (spawn_ (select1_ close_)
            (\c1. case c1 <(\c2. wait c2 (select2_ close_)) ,
                           (\c3. wait c3 (select1_ close_))>)) c0
-o { Exists c1. msg P c1 * msg (Q c1) c0 }.
```

Indeed, we obtain the expected answer and a proof term representing the computation.

```
Solution: \!c0. \X1. {
```

```
    let {[!a, [X2, X3]]} = c/spawn X1 in
    let {[!c', [X4, X5]]} = s/plus1 X2 in
    let {X6} = s/one X4 in
    let {X7} = r/plus1 [X5, X3] in
    let {X8} = r/one [X6, X7] in
    let {[!c'_1, [X9, X10]]} = s/plus2 X8 in
    let {X11} = s/one X9 in [!c'_1, [X11, X10]]}
 #P = close_
 #Q = \X1. select2_ (fwd_ X1)
Query ok.
```

Due to the increased parallelism afforded by asynchronous communication, some steps here could be carried out in parallel. For example, the sending of close and the receiving of π_1 are independent and could happen in either order. We can see that because there is no dependencies between these two lines: `X6` does not occur in `r/plus1 [X5, X3]` (nor does `X7` occur in `s/one X4`).

```
    let {X6} = s/one X4 in
    let {X7} = r/plus1 [X5, X3] in
```

By *true concurrency*, the computation where these two lines are swapped are indistinguishable from the given one.

The signature and queries from this section can be found with the course materials[3]. The completion with the remaining connectives $A \multimap B$ and $A \mathbin{\&} B$ is also available[4].

7 A Cost Semantics Tracking Total Work

We now want to instrument our semantics to compute the *total work* performed by a concurrent computation. We define this here as the total number of communication steps that take place, and for simplicity we restrict ourselves to the synchronous semantics (see Exercise 4).

For every process, we keep track of the total work that it has performed. We count here the number of *send* operations. Since every message that is sent is also received, counting the number of receives just doubles this cost.

[3] http://www.cs.cmu.edu/~fp/courses/15816-f16/misc/session/session-async.clf

[4] http://www.cs.cmu.edu/~fp/courses/15816-f16/misc/session/session-complete-async.clf

Our basic predicate for linear inference is now $\mathsf{proc}(c, w, P)$, where w tracks the amount of work performed by this process so far. We begin with the rules for tensor, which is straightfoward.

$$\frac{\mathsf{proc}(c, w, \mathsf{send}\ c\ e\ ;\ P) \quad \mathsf{proc}(d, w', y \leftarrow \mathsf{recv}\ c\ ;\ Q_y)}{\mathsf{proc}(c, w+1, P) \quad \mathsf{proc}(d, w', Q_e)}\ \otimes C$$

Spawning makes sure the new process starts at work 0.

$$\frac{\mathsf{proc}(d, w, x \leftarrow P_x\ ;\ Q_x)}{\mathsf{proc}(c, 0, P_c) \quad \mathsf{proc}(d, w, Q_c)}\ \mathsf{spawn}^c$$

Forwarding is interesting, because the work performed by the forwarding process must be accounted for. So we have to add it into the process that it notifies of the forwarding.

$$\frac{\mathsf{proc}(d, w, P) \quad \mathsf{proc}(c, w', c \leftarrow d)}{\mathsf{proc}(c, w + w', P)}\ \mathsf{fwd}$$

If we decided to count forwarding as communication, we would send the cost of the resulting process to $w + w' + 1$. Similar reasoning applies to process of type $\mathbf{1}$.

$$\frac{\mathsf{proc}(c, w, \mathsf{close}\ c) \quad \mathsf{proc}(d, w', \mathsf{wait}\ c\ ;\ Q)}{\mathsf{proc}(c, w + w' + 1, Q)}\ \mathbf{1}C$$

The remaining rules and their transcription into Celf follows the pattern of what we have done before.

For a work semantics for asynchronously communicating processes, see Exercise 4.

8 A Cost Semantics for Span

The *span* of a concurrent computation can be defined in different ways. We can say that the span consists of the number of communication steps where everything that can happen in parallel, does. Another way to define it is by looking at the dependency graph induced by the truly concurrent semantics and define it as the longest path from the root (where the computation starts) to the leaf (where the computation ends).

We specify this through a predicate $\mathsf{proc}(c, t, P)$, where t is *the earliest time* that this process could be at the given stage of the computation. Again, we only count explicit communication steps as costing time, but more complex measures are certainly possible.

We begin again with sending a channel along a channel. The earliest the two processes can synchronize is at time $\mathsf{max}(t, t')$, and then we have to add 1 to be able to continue.

$$\frac{\mathsf{proc}(c, t, \mathsf{send}\ c\ e\ ;\ P) \quad \mathsf{proc}(d, t', y \leftarrow \mathsf{recv}\ c\ ;\ Q_y)}{\mathsf{proc}(c, \mathsf{max}(t, t') + 1, P) \quad \mathsf{proc}(d, \mathsf{max}(t, t') + 1, Q_e)}\ \otimes C$$

Spawning makes sure the new process starts with the the clock of the parent process, because that is the earliest time it could have been spawned. If we like, we could also count the spawn itself; here we do not.

$$\frac{\mathsf{proc}(d, t, x \leftarrow P_x\ ;\ Q_x)}{\mathsf{proc}(c, t, P_c) \quad \mathsf{proc}(d, t, Q_c)}\ \mathsf{spawn}^c$$

Forwarding can take place at the earliest that either process could have gotten to this point.

$$\frac{\mathsf{proc}(d, t, P) \quad \mathsf{proc}(c, t', c \leftarrow d)}{\mathsf{proc}(c, \mathsf{max}(t, t'), P)}\ \mathsf{fwd}$$

If we decided to count forwarding as communication, we would set the cost of the resulting process to $w + w' + 1$. Similar reasoning applies to process of type $\mathbf{1}$.

$$\frac{\mathsf{proc}(c, t, \mathsf{close}\ c) \quad \mathsf{proc}(d, t', \mathsf{wait}\ c\ ;\ Q)}{\mathsf{proc}(c, \mathsf{max}(t, t') + 1, Q)}\ \mathbf{1}C$$

One reason we are counting communication steps that advance the type, but not spawns or forwards is that this allows us to use the type as a guide for the number of communications that must happen, even if we do not necessarily know their timing.

Again, transcription into CLF does not pose any particular challenges, except perhaps implementing the arithmetic.

Exercises

Exercise 1 Our encoding takes advantage of the asymmetric nature of intuitionistic sequents so we can leave the offering channel implicit. Revise this implementation so that the offering process is abstracted over the offering channel, which would give the type

```
proc : (ch A -o proc A) -> ch A -> type.
```

Of course, the encoding of process expressions has to change accordingly write. Encode this approach in Celf, rewrite the examples in the new syntax, and compare.

Exercise 2 In the case of the asynchronous semantics, the simple rule

```
c/fwd : msg P D * proc (fwd_ D) C -o {msg P C}.
```

is no longer sufficient to implement forwarding. Exhibit a concrete, well-typed process that will get stuck with only this rule and extend the implementation of forwarding that it works for all the connectives.

Exercise 3 Write a cost semantics that counts the total number of processes that will be created during an execution.

Exercise 4 Give a cost semantics counting total work for *asynchronous* communication. As before, only count sending of messages (not receipt) and exclude forward and spawn.

Exercise 5 Give a cost semantics for span for *asynchronous* communication. As before, only count the sending of message (not receipt) and exclude forward and spawn.

Exercise 6 Extend the representation of SILL with recursively defined types and recursively defined processes so that you can encode programs such as the queue or stack. Discuss some of the options and obstacles, and implement your extension, with example, in Celf.

References

[BBMS16] Peter Brottveit Bock, Alessandro Bruni, Agata Murawska, and Carsten Schürmann. Representing session types. Unpublished manuscript, presented at the seminar in honor of Dale Miller's 60th birthday, December 2016.

[CPWW02] 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.

[Pfe94] Frank Pfenning. Structural cut elimination in linear logic. Technical Report CMU-CS-94-222, Department of Computer Science, Carnegie Mellon University, December 1994.

[Ree09] Jason C. Reed. *A Hybrid Logical Framework*. PhD thesis, Carnegie Mellon University, September 2009. Available as Technical Report CMU-CS-09-155.

[SN11] Anders Schack-Nielsen. *Implementing Substructural Logical Frameworks*. PhD thesis, IT University of Copenhagen, January 2011.

[WCPW02] 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 Resource Semantics

15-816: Substructural Logics
Frank Pfenning

Lecture 25
December 1, 2016

In this lecture we explore a new presentation of substructural logics, one where resources are explicitly tracked in the judgments. It is a new form of semantics, given by intuitionistic means, while generally semantic investigations take a classical point of view even if the studied subject is intuitionistic.

In the present lecture we change the judgments, but we try to disturb the nature of proofs as little as possible. In the following lecture, we will take a less restrictive view, which leads to new ways to reason linearly.

One of the important reasons to investigate a resource semantics is that it allows us to express new properties and relations, beyond what is possible in linear logic itself. Further materials and properties on resource semantics are given by Reed [Ree09].

1 Resource-Aware Judgments

In order to give a Kripke-like resource semantics for ordered logic we label all the resources with unique labels representing that resource. In the succedent we record all the resources that may be used, which may be a subset of the resources listed in the antecedent. So a sequent has the form

$$A_1[\alpha_1], \ldots, A_n[\alpha_n] \vdash C[p]$$

where p is formed from $\alpha_1, \ldots, \alpha_n$ with a binary resource combinator '$\cdot$'. In addition we have the empty resource label ϵ, which is the unit of '$\cdot$'.

Resource combination is associative, so we have the laws

$$\begin{array}{rcl} p \cdot \epsilon & = & p \\ \epsilon \cdot q & = & q \\ (p \cdot q) \cdot r & = & p \cdot (q \cdot r) \end{array}$$

We will apply these equations silently, just as we, for example, silently reorder hypotheses.

By labeling resources we recover the property of weakening.

> **Weakening:** If $\Gamma \vdash C[p]$ then $\Gamma, A[\alpha] \vdash C[p]$. Here α must be new in order to maintain the invariant on seqents that all antecedents are labeled with distinct resource parameters.

Contraction in general cannot be quite formulated, since we cannot contract $A[\alpha], A[\beta]$ to $A[\alpha \cdot \beta]$ because, at least for the moment, hypotheses can only be labeled with resource parameters and not combinations of them. Nevertheless, we treat antecedents *persistently* by propagating them to all premises of the rules.

The identity is quite straightforward:

$$\frac{}{\Gamma, A[\alpha] \vdash A[\alpha]}\ \mathsf{id}$$

Cut is a bit more complicated, because resource labels in succedents are more general than in antecedents. But we have already seen this in substitution principles with proof terms, so we imitate this solution, substituting resources p for resource parameter α:

$$\frac{\Gamma \vdash A[p] \quad \Gamma, A[\alpha] \vdash C[l \cdot \alpha \cdot r]}{\Gamma \vdash C[l \cdot p \cdot r]}\ \mathsf{cut}$$

We now revisit each of the connectives so far in turn, deriving the appropriate rules. The goal is to achieve an exact isomorphism between the linear logic inference rules based on hypothetical judgments and the inference rules based on resources. Hidden behind the isomorphism is the equational reasoning in the resource algebra.

Fuse. The resources available to achieve the goals are split between the two premises. Previously, this was achieved by splitting the context itself.

Note that we use Γ here to stand for a context in which all assumptions are labeled with unique resource parameters.

$$\frac{\Gamma \vdash A[p] \quad \Gamma \vdash B[q]}{\Gamma \vdash A \bullet B[p \cdot q]}\ \bullet R$$

In order to apply a left rule to a given assumption $A[\alpha]$, the resource α must actually be available, which is recorded in the succedent. Upon application of the rule the resource is no longer available, but new resources may now be available (depending on the connective).

$$\frac{\Gamma, (A \bullet B)[\alpha], A[\beta], B[\gamma] \vdash C[l \cdot (\beta \cdot \gamma) \cdot r]}{\Gamma, (A \bullet B)[\alpha] \vdash C[l \cdot \alpha \cdot r]}\ \bullet L^{\beta,\gamma}$$

In this rule, α is consumed and new resources β and γ are introduced. By associativity, we could omit the parentheses in the resources of the premise.

Under and Over. The intuitions above give us enough information to write out these rules directly, modeling the linear sequent calculus.

$$\frac{\Gamma, A[\alpha] \vdash B[\alpha \cdot p]}{\Gamma \vdash (A \setminus B)[p]}\ \setminus R^{\alpha} \qquad \frac{\Gamma, A[\alpha] \vdash B[p \cdot \alpha]}{\Gamma \vdash (B \,/\, A)[p]}\ / R^{\alpha}$$

In the elimination rule we see again how a split between the antecedents is represented as a split between the resources.

$$\frac{\Gamma, (A \setminus B)[\alpha] \vdash A[p] \quad \Gamma, (A \setminus B)[\alpha], B[\beta] \vdash C[l \cdot \beta \cdot r]}{\Gamma, (A \setminus B)[\alpha] \vdash C[l \cdot p \cdot \alpha \cdot r]}\ \setminus L^{\beta}$$

By strengthening, we can see that the antecedent $(A \setminus B)[\alpha]$ can not be used in either premise. At this point we can easily see how $B \,/\, A$ should work, just reversion $\alpha \cdot p$ in the succedent of the conclusion.

$$\frac{\Gamma, (B \,/\, A)[\alpha] \vdash A[p] \quad \Gamma, (B \,/\, A)[\alpha], B[\beta] \vdash C[l \cdot \beta \cdot r]}{\Gamma, (B \,/\, A)[\alpha] \vdash C[l \cdot \alpha \cdot p \cdot r]}\ / L^{\beta}$$

Unit. Here, we just have to enforce the emptiness of the resources.

$$\frac{}{\Gamma \vdash \mathbf{1}[\epsilon]}\ \mathbf{1}R \qquad \frac{\Gamma, \mathbf{1}[\alpha] \vdash C[l \cdot r]}{\Gamma, \mathbf{1}[\alpha] \vdash C[l \cdot \alpha \cdot r]}\ \mathbf{1}L$$

In the $\mathbf{1}L$ rule we replace α by ϵ and then use its unit property to obtain $l \cdot r$.

2 Exponentials

Our representation technique for the sequent calculus using explicit resources is already rich enough to handle persistence. We just allow antecedents $A_{\mathsf{U}}[\epsilon]$ together with resource-bound $A_{\mathsf{O}}[\alpha]$. In $\downarrow R$, we effectively "check" the emptiness of resource-bound assumptions. All propositions annotated here with resources are ordered.

$$\frac{\Gamma \vdash A_{\mathsf{U}}[\epsilon]}{\Gamma \vdash (\downarrow^{\mathsf{U}}_{\mathsf{O}} A_{\mathsf{U}})[\epsilon]}\ \downarrow R \qquad \frac{\Gamma, (\downarrow^{\mathsf{U}}_{\mathsf{O}} A_{\mathsf{U}})[\alpha], A_{\mathsf{U}}[\epsilon] \vdash C[l \cdot r]}{\Gamma, (\downarrow^{\mathsf{U}}_{\mathsf{O}} A_{\mathsf{U}})[\alpha] \vdash C[l \cdot \alpha \cdot r]}\ \downarrow L$$

$$\frac{\Gamma \vdash A[\epsilon]}{\Gamma \vdash (\uparrow^{\mathsf{U}}_{\mathsf{O}} A)[\epsilon]}\ \uparrow R \qquad \frac{\Gamma, (\uparrow^{\mathsf{U}}_{\mathsf{O}} A)[\epsilon], A[\alpha] \vdash C[l \cdot \alpha \cdot r]}{\Gamma, (\uparrow^{\mathsf{U}}_{\mathsf{O}} A)[\epsilon] \vdash C[l \cdot r]}\ \uparrow L^{\alpha}$$

The rules for the remaining connectives are easy to fill in, including the expected rules for structural proposition A_{U}.

One can also write A_{U} instead of the more verbose $A_{\mathsf{U}}[\epsilon]$.

3 Correspondence

It is now easy to establish that the resource calculus is in bijective correspondence with the ordered sequent calculus. Moroever, it satisfies the expected properties of cut and identity. Key is the crucial strengthening property.

For the remainder of this lecture we assume that a resource context has hypotheses of the form $A_{\mathsf{U}}[\epsilon]$ and $A_{\mathsf{O}}[\alpha]$, where all resource parameters α are distinct, and the succedent has the form $C[p]$, where p is a product of distinct resource parameters. We write $\alpha \notin p$ if α does not occur in p. The equational theory for resources remains associativity for '$\cdot$' with unit ϵ.

Theorem 1 (Strengthening for Resource Semantics) *If $\Gamma, A[\alpha] \vdash C[p]$ and $\alpha \notin p$ then $\Gamma \vdash C[p]$ with the same proof.*

Proof: By induction on the structure of the given proof. □

Theorem 2 (Identity) *In the system where identity is restricted to atomic propositions, general identity is admissible. That is, $A[\alpha] \vdash A[\alpha]$ for any proposition A.*

Proof: By induction on the structure of A. □

Theorem 3 (Cut) *In the system without the* cut *and* cut! *rules, they are admissible. That is, for $p = q = \epsilon$ or $q = \alpha$, we have*

$$\frac{\Gamma \vdash A[p] \quad \Gamma, A[q] \vdash C[l \cdot q \cdot r]}{\Gamma \vdash C[l \cdot p \cdot r]}\ \mathsf{cut}$$

Proof: By nested induction, first on the cut formula A, then on the structure of the proofs in the two premises (one must become smaller while the other remains the same). □

In order to formulate a correspondence theorem, we need to express relationships between assumptions. We write $(A_1, \ldots, A_n)[\epsilon] = A_1[\epsilon], \ldots, A_n[\epsilon]$ and $(A_1, \ldots, A_n)[\vec{\alpha}] = A_1[\alpha_1], \ldots, A_n[\alpha_n]$. Furthermore, we need to construct a pair of contexts $\Gamma_\mathsf{U}; \Omega_\mathsf{O}$ from given a resource context. For ease of definition, we do not require a separation of zones but generate a mixed context with linear and structural antecedents that can then be separated.

$$\begin{array}{lcl}
(\cdot)|_\epsilon & = & (\cdot) \\
(\Gamma, \Gamma')|_{p \cdot q} & = & \Gamma|_p, \Gamma'|_q \\
(A_\mathsf{O}[\alpha])|_\alpha & = & A_\mathsf{O} \\
(A_\mathsf{O}[\alpha])|_\epsilon & = & (\cdot) \\
(A_\mathsf{U}[\epsilon])|_\epsilon & = & A_\mathsf{U}
\end{array}$$

Because of the equational theory, this definition has some nondeterminism. Under the general assumptions of this section, $\Gamma|_p$ will be defined and unique.

Theorem 4 (Correspondence)

(i) *If $\Gamma_\mathsf{U}; \Omega_\mathsf{O} \vdash C$ then $\Gamma_\mathsf{U}[\epsilon], \Omega_\mathsf{O}[\vec{\alpha}] \vdash C[\alpha_1 \cdots \alpha_n]$.*

(ii) *If $\Gamma \vdash C[p]$ then $\Gamma|_p \vdash C$.*

Moreover, the correspondence between linear and resource proofs is a bijection.

Proof: By straightforward inductions, exploiting strengthening. □

4 Linear Resource Semantics

So far have presented the resource semantics for an ordered logic. How do we get one for linear logic? Actually, this is quite easy: we just add the equation

$$p \cdot q = q \cdot p$$

and we get linear logic! Under and over now collapse, because $\alpha \cdot p = p \cdot \alpha$. In other ways, the rules remain exactly the same.

What about structural logic? We get this by identifying *all* terms

$$p = \epsilon$$

which have already done implicitly by writing structural antecedents as $A[\epsilon]$.

An interesting intermediate point is affine logic. For this, it seems best to postulate a resource inequality, defined here (in the presence of symmetry, that is, linear logic) as

$$p \leq q \quad \text{iff} \quad \exists r. p \cdot r = q$$

and then changing the rule to allow subset at some critical junctures, such as

$$\frac{\alpha \leq p}{\Gamma, A[\alpha] \vdash A[p]}\ \mathsf{id}$$

References

[Ree09] Jason C. Reed. *A Hybrid Logical Framework*. PhD thesis, Carnegie Mellon University, September 2009. Available as Technical Report CMU-CS-09-155.

Lecture Notes on Classical Linear Logic

15-816: Substructural Logics
Frank Pfenning

Lecture 25
December 8, 2016

Originally, linear logic was conceived by Girard [Gir87] as a *classical* system, with one-sided sequents, an involutive negation, and an appropriate law of excluded middle. For a number of the applications, such as functional computation, logic programming, and implicit computational complexity the intuitionistic version is more suitable. In the case of concurrent computation, both classical and intuitionistic systems may be used, although the additional expressiveness afforded by the intuitionistic system seems to have some advantages even in that setting.

In this lecture we present classical linear logic and then show that we can easily interpret it intuitionistically. Briefly, classical linear logic can be modeled intuitionistically as deriving a contradiction from linear assumptions. This is shown via a so-called *double-negation translation*. Its parametric nature allows a number of additional variants of classical linear logic to be explained intuitionistically, in particular the so-called *mix* rules.

These lecture notes do *not* present the operational semantics of classical linear logic as a basis for concurrency which we presented in lecture. The one we presented can be found in Section 5 of [CPT16] in a somewhat different notation, another semantics is given by Wadler [Wad12].

1 Classical Linear Sequents

A sequent in classical linear logic just has the form

$$\vdash A_1, \ldots, A_n$$

where $A_1, \ldots, A_n$ are propositions. The comma separator can be read as a form of disjunction, which does not exist in intuitionistic linear logic.

An important aspect of the system is a negation operator, written as $A^\perp$, which is defined for all propositions except that atomic ones. As we define the rules for the classical connectives, we will also have to define negation. We already note that

$$(A^\perp)^\perp = A$$

is one of the basic laws.

The identity emphasizes the fact that reasoning in classical linear logic is akin to deriving a contradiction.

$$\frac{}{\vdash A, A^\perp}\ \mathsf{id}$$

Cut is somehow the dual—we do not cut a proposition, but a proposition and its negation.

$$\frac{\vdash \Sigma, A \quad \vdash \Sigma', A^\perp}{\vdash \Sigma, \Sigma'}\ \mathsf{cut}$$

2 Multiplicative Connectives

The connectives now are no longer defined by left and right rules, but by right rules, negation, and right rules for the negated proposition. We can see that this must be the case by looking at the cut rule.

The multiplicative conjunction $A \otimes B$ is quite similar to the intuitionistic version.

$$\frac{\vdash \Sigma, A \quad \vdash \Sigma', B}{\vdash \Sigma, \Sigma', A \otimes B}\ \otimes$$

The negation $(A \otimes B)^\perp = A^\perp \parr B^\perp$ introduces a new connective $\parr$ which does not exist in intuitionistic linear logic.

$$\frac{\vdash \Sigma, A, B}{\vdash \Sigma, A \parr B}\ \parr$$

It is a multiplicative form of disjunction, and clearly satisfies the law of excluded middle $A \parr A^\perp$. We can check the cut reduction and identity

expansion, just as we did in the intuitionistic case. First, the cut reduction:

$$\dfrac{\dfrac{\vdash \Sigma, A \quad \vdash \Sigma', B}{\vdash \Sigma, \Sigma', A \otimes B}\otimes \quad \dfrac{\vdash \Sigma'', A^\perp, B^\perp}{\vdash \Sigma'', A^\perp \parr B^\perp}\parr}{\vdash \Sigma, \Sigma', \Sigma''}\,\mathsf{cut}$$

$$\longrightarrow_R$$

$$\dfrac{\vdash \Sigma', B \qquad \dfrac{\vdash \Sigma, A \quad \vdash \Sigma'', A^\perp, B^\perp}{\vdash \Sigma, \Sigma'', B^\perp}\,\mathsf{cut}}{\vdash \Sigma, \Sigma', \Sigma''}\,\mathsf{cut}$$

Second, the identity expansion:

$$\dfrac{}{\vdash A \otimes B, A^\perp \parr B^\perp}\,\mathsf{id}_{A\otimes B} \quad \longrightarrow_R \quad \dfrac{\dfrac{}{\vdash A, A^\perp}\,\mathsf{id}_A \quad \dfrac{}{\vdash B, B^\perp}\,\mathsf{id}_B}{\vdash A \otimes B, A^\perp, B^\perp}\otimes}{\vdash A \otimes B, A^\perp \parr B^\perp}\parr$$

We will not continue to do so, but leave it as an exercise to check cut reduction and identity expansion.

The multiplicative units do not present surprises. Note that unlike $A \parr B$, $\bot$ can actually be given meaning intuitionistically.

$$\dfrac{}{\vdash \mathbf{1}}\,\mathbf{1} \qquad \mathbf{1}^\perp = \bot \qquad \dfrac{\vdash \Sigma}{\vdash \Sigma, \bot}\,\bot$$

Not surprisingly, $\bot$ is the identity for $\parr$.

3 Additive Connectives

The additives do not differ much in their intuitionistic and classical versions.

$$\dfrac{\vdash \Sigma, A}{\vdash \Sigma, A \oplus B}\,\oplus_1 \qquad \dfrac{\vdash \Sigma, B}{\vdash \Sigma, A \oplus B}\,\oplus_2$$

In a classical calculus, $\oplus$ and $\&$ are duals

$$(A \oplus B)^\perp = A^\perp \,\&\, B^\perp$$

and the rule for $\&$ are as expected, copying the context Σ to both premises.

$$\frac{\vdash \Sigma, A \quad \vdash \Sigma, B}{\vdash \Sigma, A \,\&\, B}\ \&$$

The units present no particular surprises or difficulties.

$$\frac{}{\vdash \Sigma, \top}\ \top \qquad \top^\perp = \mathbf{0} \qquad \text{no rule for } \mathbf{0}$$

4 Exponential Modalities

Girard's formulation of the modalities was in terms of explicit rules for weakening, contraction, and dereliction. However, it is also possible to present classical linear logic using two judgments, truth and possibility. This is what Andreoli [And92] calls the *dyadic* formulation of linear logic. We show here the original rules for reference; other two-sided formulations can be found in [CCP03]. Note that Girard's formulation does not lend itself to a structural proof of cut elimination, which Andreoli did not present but can be found in [CCP03] and goes back to an another unpublished technical report [Pfe94].

In order to explain the rules for $!A$ we have to define its dual,

$$(!A)^\perp = ?A^\perp$$

Persistent resources become formulas $?A$, because we are working just on the right of the sequent. The ! rule requires there to be no linear resources, but permits persistent ones. These are now marked with $?$, so we obtain

$$\frac{\vdash ?\Sigma, A}{\vdash ?\Sigma, !A}\ !$$

Conversely, a persistent formula is true, which becomes

$$\frac{\vdash \Sigma, A}{\vdash \Sigma, ?A}\ ?$$

Why do not retain a copy of $?A$ in the premise, because we have explicit rules for weakening and contraction of persistent propositions.

$$\frac{\vdash \Sigma}{\vdash \Sigma, ?A}\ \mathsf{Weaken} \qquad \frac{\vdash \Sigma, ?A, ?A}{\vdash \Sigma, ?A}\ \mathsf{Contract}$$

5 Double-Negation Interpretation

We now follow [CCP03], interpreting classical linear logic in intuitionistic linear logic. The technique of a *double-negation translation* is quite common in logics [Fri78] and is related to conversion to continuation-passing style in programming languages.

Roughly, we think of classical $\vdash \Sigma$ as intuitionistic $\neg[\![\Sigma]\!] \vdash \bot$, that is, deriving a contradiction from the negation of the translation of Σ. It is not immediately clear what should play the role of negation on the intuitionistic side, however. Instead of using $\bot$ and $\neg A$ (which we have yet to define intuitionistically), we use a new atomic proposition p and translate $\vdash \Sigma$ to $[\![\Sigma]\!]_p \multimap p \vdash p$. We will later exploit the fact that the translation is parametric in p by considering some choices for what p might be. We write

$$\sim_p A = A \multimap p$$

to emphasize the interpretation of the translation as a form of negation. We usually omit the p, since it is never changed throughout a translation.

The theorem we are striving for is

$$\vdash \Sigma \quad \text{iff} \quad \sim_p [\![\Sigma]\!]_p \vdash p$$

Instead of just presenting the translation, we consider various cases to see what it should be. For example, what happens with atoms? Could we just translate atoms to themselves?

$$\frac{}{\vdash P, P^\perp}\ \mathsf{id}$$

If we set

$$\begin{array}{lcl} [\![P]\!] & = & P \\ [\![P^\perp]\!] & = & \sim P \end{array}$$

This means we would have to prove

$$\sim P, \sim(\sim P) \vdash p$$

which is

$$P \multimap p, (P \multimap p) \multimap p \vdash p$$

and easy to show.

Let's try $A \otimes B$.

$$\frac{\vdash \Sigma, A \quad \vdash \Sigma', B}{\vdash \Sigma, \Sigma', A \otimes B}\ \otimes$$

If we generate a tensor, but double-negate the subformulas,

$$[\![A \otimes B]\!] = (\sim\sim[\![A]\!]) \otimes (\sim\sim[\![B]\!])$$

then the sequent we have to show after translation would be

$$\sim[\![\Sigma]\!], \sim[\![\Sigma']\!], \sim((\sim\sim[\![A]\!]) \otimes (\sim\sim[\![B]\!])) \vdash p$$

After applying $\multimap L$ and closing the subgoals with the identity, we are looking at

$$\dfrac{\begin{array}{c}\vdots\\ \sim[\![\Sigma]\!], \sim[\![\Sigma']\!] \vdash (\sim\sim[\![A]\!]) \otimes (\sim\sim[\![B]\!])\end{array} \quad \dfrac{}{p \vdash p}\,\mathsf{id}}{\sim[\![\Sigma]\!], \sim[\![\Sigma']\!], \sim((\sim\sim[\![A]\!]) \otimes (\sim\sim[\![B]\!])) \vdash p}\,\multimap L$$

Now we can apply the $\otimes R$ rule and then $\multimap R$ to bring $\sim[\![A]\!]$ back to the left-hand side.

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ \sim[\![\Sigma]\!], \sim[\![A]\!] \vdash p\end{array}}{\sim[\![\Sigma]\!] \vdash \sim\sim[\![A]\!]}\,\multimap R \quad \dfrac{\begin{array}{c}\vdots\\ \sim[\![\Sigma']\!], \sim[\![B]\!] \vdash p\end{array}}{\sim[\![\Sigma']\!] \vdash \sim\sim[\![B]\!]}\,\multimap R}{\sim[\![\Sigma]\!], \sim[\![\Sigma']\!] \vdash (\sim\sim[\![A]\!]) \otimes (\sim\sim[\![B]\!])}\,\otimes R \quad \dfrac{}{p \vdash p}\,\mathsf{id}}{\sim[\![\Sigma]\!], \sim[\![\Sigma']\!], \sim((\sim\sim[\![A]\!]) \otimes (\sim\sim[\![B]\!])) \vdash p}\,\multimap L$$

At this point we can apply the "induction hypothesis" of the translation, asserting that the open premises follow since $\vdash \Sigma, A$ and $\vdash \Sigma, B$.

For $A \parr B$, matters are a bit more complicated.

$$\dfrac{\vdash \Sigma, A, B}{\vdash \Sigma, A \parr B}\,\parr$$

Since there is no $\parr$ connective on the intuitionistic side, we have to translate uses of the $\parr$ rule into application of the $\otimes L$ rule. This makes sense, since $\parr$ was justified as the formal dual of $\otimes$. This means we have to distribute the negations a bit differently.

$$[\![A \parr B]\!] = \sim(\sim[\![A]\!] \otimes \sim[\![B]\!])$$

Then we get (in somewhat abbreviated form)

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \sim[\![\Sigma]\!], \sim[\![A]\!], \sim[\![B]\!]\end{array}}{\sim[\![\Sigma]\!], \sim[\![A]\!] \otimes \sim[\![B]\!]}\,\otimes L}{\sim[\![\Sigma]\!], \sim\sim(\sim[\![A]\!] \otimes \sim[\![B]\!])}\,\multimap L, \multimap R$$

where the open subproof follows again inductively, from the translation of the premise of the classical $\parr$ rule.

We can continue to reason along these lines. For connectives where there is an intuitionistic counterpart, we just double-negate the subformulas. For those where there is not, we negate once, use the intuitionistic dual, and then negate once again. This leads us to the following table.

$$\begin{array}{lcl}
[\![P]\!] & = & P \\
[\![P^\perp]\!] & = & {\sim}P \\
[\![A \otimes B]\!] & = & {\sim}{\sim}[\![A]\!] \otimes {\sim}{\sim}[\![B]\!] \\
[\![A \parr B]\!] & = & {\sim}({\sim}[\![A]\!] \otimes {\sim}[\![B]\!]) \\
[\![\mathbf{1}]\!] & = & \mathbf{1} \\
[\![\bot]\!] & = & {\sim}\mathbf{1} \\
[\![A \oplus B]\!] & = & {\sim}{\sim}[\![A]\!] \oplus {\sim}{\sim}[\![B]\!] \\
[\![A \mathbin{\&} B]\!] & = & {\sim}{\sim}[\![A]\!] \mathbin{\&} {\sim}{\sim}[\![B]\!] \\
[\![\mathbf{0}]\!] & = & \mathbf{0} \\
[\![\top]\!] & = & \top \\
[\![!A]\!] & = & !{\sim}{\sim}[\![A]\!] \\
[\![?A]\!] & = & {\sim}!{\sim}[\![A]\!]
\end{array}$$

There are more economical translations where some double negations are omitted, but the one shown above seems most systematic.

6 Correctness of the Translation

From our little derivation, it is easy to see the following:

Theorem 1 (From CLL to ILL) *If* $\vdash \Sigma$ *then* ${\sim}[\![\Sigma]\!]_p \vdash p$.

Proof: By induction on the structure of the given derivation. A few lemmas are needed for the exponentials (see [CCP03]), to bridge the gap between the monadic and dyadic presentations of the logic. □

The converse requires an entirely different technique. First we observe that intuitionistic linear logic makes some finer distinctions (especially in the treatment of linear implication). If these distinctions are ignored, we can prove the result classically. In this translation, we think of $A \multimap B = A^\perp \parr B$, on the classical side.

Lemma 2 *If* $\Gamma\,;\,\Delta \to A$ *then* $\vdash (!\Gamma)^\perp, \Delta^\perp, A$

Proof: By induction on the given derivation, using some lemmas regarding classical provability. □

The second lemma we need is that, classically, the translation is essentially the identity, if we use $\perp$.

Lemma 3 *For any proposition* A, $[\![A]\!]_\perp \overset{CLL}{\equiv} A$.

Proof: A simple induction on the structure of A, mostly exploiting that $\sim_\perp\sim_\perp A \overset{CLL}{\equiv} A$. □

Theorem 4 (From ILL to CLL) *If* $\sim[\![\Sigma]\!]_p \vdash p$ *then* $\vdash \Sigma$.

Proof: If $\Sigma = A_1, \ldots, A_n$, we have

$$\sim_p[\![A_1]\!]_p, \ldots, \sim_p[\![A_n]\!]_p \vdash p$$

Since classical logic proves *more* (Lemma 2), we get

$$\vdash (\sim_p[\![A_1]\!]_p)^\perp, \ldots, (\sim_p[\![A_n]\!]_p)^\perp, p$$

This proof is parametric in p, so we can substitute $p = \perp$ throughout the proof and obtain

$$\vdash (\sim_\perp[\![A_1]\!]_\perp)^\perp, \ldots, (\sim_\perp[\![A_n]\!]_\perp)^\perp, \perp$$

Now we recall that $(\sim_\perp A)^\perp = (A^\perp \parr \perp)^\perp = (A \otimes \mathbf{1})$. Since $A \otimes \mathbf{1} \overset{CLL}{\equiv} A$ we can use cut multiple times and arrive at

$$\vdash [\![A_1]\!]_\perp, \ldots, [\![A_n]\!]_\perp, \perp$$

Then we recall that $\perp^\perp = \mathbf{1}$ so we can cut this with $\vdash \mathbf{1}$ to get

$$\vdash [\![A_1]\!]_\perp, \ldots, [\![A_n]\!]_\perp$$

Finally recall that $[\![A]\!]_\perp \overset{CLL}{\equiv} A$. Using cut A number of times we get

$$\vdash A_1, \ldots, A_n$$

which is what we needed to show. □

7 Mix and Other Variations

In his original paper [Gir87], Girard also discussed a variant of linear logic with the rules of mix. They are

$$\frac{}{\vdash \cdot}\ \mathsf{mix}_0 \qquad \frac{\vdash \Sigma \quad \vdash \Sigma'}{\vdash \Sigma, \Sigma'}\ \mathsf{mix}_2$$

It turns out that the logic with these roles (and good proof-theoretic properties), can also be characterized with axioms postulating that $\bot \overset{CLL}{\equiv} \mathbf{1}$. Why is that? If $\bot$ and $\mathbf{1}$ are equivalent, this means we have $\vdash \mathbf{1}, \mathbf{1}$ and $\vdash \bot, \bot$ since $\mathbf{1}^\bot = \bot$ and $\bot^\bot = \mathbf{1}$.

Then we can derive mix_0 as

$$\dfrac{\dfrac{\dfrac{}{\vdash \mathbf{1}}\,\mathbf{1} \qquad \vdash \bot, \bot}{\vdash \bot}\,\mathsf{cut} \qquad \dfrac{}{\vdash \mathbf{1}}\,\mathbf{1}}{\vdash \cdot}\,\mathsf{cut}$$

and mix_2 as

$$\dfrac{\dfrac{\vdash \Sigma}{\vdash \Sigma, \bot}\,\bot \qquad \dfrac{\dfrac{\vdash \Sigma'}{\vdash \Sigma', \bot}\,\bot \qquad \vdash \mathbf{1}, \mathbf{1}}{\vdash \Sigma', \mathbf{1}}\,\mathsf{cut}}{\vdash \Sigma, \Sigma'}\,\mathsf{cut}$$

Now we can proceed as in the previous section, and exploit the parametricity of the translation by using

$$p = \mathbf{1}$$

In the crucial step, we use

$$(\sim_{\mathbf{1}} A)^\bot = (A^\bot \parr \mathbf{1})^\bot = A \otimes \bot \overset{MIX}{\equiv} A \otimes \mathbf{1}$$

In this way we come to the conclusion that using the mix rules in the classical setting is just like trying consume all linear resources (proving $\mathbf{1}$), rather than trying to derive a contradiction (proving $\bot$). Since the process calculus admits such as interpretation, it seems reasonable that in encodings of concurrent computation the mix rule is difficult to deny. In the intuitionistic case, we can derive a counterpart as follows.

A process that does not offer any external services, has the form

$$\Gamma\ ;\ \Delta \vdash P :: z : \mathbf{1}$$

Two such processes can be combined as follows:

$$\dfrac{\Gamma\ ;\ \Delta \vdash P :: z : \mathbf{1} \quad \dfrac{\Gamma\ ;\ \Delta' \vdash Q :: w : \mathbf{1}}{\Gamma\ ;\ \Delta', z{:}\mathbf{1} \vdash z().Q :: w : \mathbf{1}}\ \mathbf{1}L}{\Gamma\ ;\ \Delta, \Delta' \vdash (\nu z)(P \mid z().Q) :: w : \mathbf{1}}\ \mathsf{cut}$$

This, however, does not quite have the desired effect, because Q cannot reduce until P has completed its computation. It is, in effect, a sequential composition. This is why, in most recent incarnations of the proof term assignment, we have separated the input prefix from its scope. In the earliest paper [CP10], the $\mathbf{1}L$ rule was entirely silent, but that created some small discord between the proof theory and the process reductions, as attested by the relative complexity of the bisimulation theorems in that paper. With the newer process assignment we obtain:

$$\dfrac{\Gamma\ ;\ \Delta \vdash P :: z : \mathbf{1} \quad \dfrac{\Gamma\ ;\ \Delta' \vdash Q :: w : \mathbf{1}}{\Gamma\ ;\ \Delta', z{:}\mathbf{1} \vdash z().\mathbf{0} \mid Q :: w : \mathbf{1}}\ \mathbf{1}L}{\Gamma\ ;\ \Delta, \Delta' \vdash (\nu z)(P \mid z().\mathbf{0} \mid Q) :: w : \mathbf{1}}\ \mathsf{cut}$$

This now permits P and Q to proceed in parallel.

We can replace p by other constants and obtain other interpretations. At this point, one of them is still open, in the sense that we have not found a good independent proof-theoretically satisfying characterization (see [CCP03]).

Exercises

Exercise 1 (Classical Harmony) Give the missing cut reductions and identity expansions for classical linear logic.

Exercise 2 (Dyadic Classical Linear Logic) Give the rules for a one-sided, two-zone sequent calculus based on the same ideas as separating persistent resources from linear ones. Show that derivable sequents are the same as the ones for the one-sided, one-zone sequent calculus presented in this lecture.

Exercise 3 (Mix) Prove that in the presence of the mix rules, $\vdash \mathbf{1}, \mathbf{1}$ and $\vdash \bot, \bot$ are derivable.

References

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):197–347, 1992.

[CCP03] 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.

[CP10] 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.

[CPT16] Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 26(3):367–423, 2016.

[Fri78] Harvey Friedman. Classically and intuitionistically provably recursive functions. In D.S. Scott and G.H. Muller, editors, *Higher Set Theory*, pages 21–27. Springer-Verlag LNM 699, 1978.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Pfe94] Frank Pfenning. Structural cut elimination in linear logic. Technical Report CMU-CS-94-222, Department of Computer Science, Carnegie Mellon University, December 1994.

[Wad12] 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.