Lecture Notes on
The Lambda Calculus

15-814: Types and Programming Languages
Frank Pfenning

Lecture 1
Tuesday, September 4, 2018

1 Introduction

This course is about the principles of programming language design, many of which derive from the notion of *type*. Nevertheless, we will start by studying an exceedingly pure notion of computation based only on the notion of function, that is, Church's λ-calculus [CR36]. There are several reasons to do so.

- We will see a number of important concepts in their simplest possible form, which means we can discuss them in full detail. We will then reuse these notions frequently throughout the course without the same level of detail.
- The λ-calculus is of great historical and foundational significance. The independent and nearly simultaneous development of Turing Machines [Tur36] and the λ-Calculus [CR36] as universal computational mechanisms led to the *Church-Turing Thesis*, which states that the effectively computable (partial) functions are exactly those that can be implemented by Turing Machines or, equivalently, in the λ-calculus.
- The notion of function is the most basic abstraction present in nearly all programming languages. If we are to study programming languages, we therefore must strive to understand the notion of function.
- It's cool!

2 The λ-Calculus

In ordinary mathematical practice, functions are ubiquitous. For example, we might define

$$\begin{aligned} f(x) &= x + 5 \\ g(y) &= 2 * y + 7 \end{aligned}$$

Oddly, we never state what f or g actually are, we only state what happens when we apply them to arbitrary arguments such as x or y. The λ-calculus starts with the simple idea that we should have notation for the function itself, the so-called λ-abstraction.

$$\begin{aligned} f &= \lambda x.\, x + 5 \\ g &= \lambda y.\, 2 * y + 7 \end{aligned}$$

In general, $\lambda x.\, e$ for some arbitrary expression e stands for the function which, when applied to some e' becomes $[e'/x]e$, that is, the result of *substituting* or *plugging in* e' for occurrences of the variable x in e. For now, we will use this notion of substitution informally—in the next lecture we will define it formally.

We can already see that in a pure calculus of functions we will need at least three different kinds of expressions: λ-abstractions $\lambda x.\, e$ to form function, *application* $e_1\, e_2$ to apply a function e_1 to an argument e_2, and *variables* x, y, z, etc. We summarize this in the following form

$$\begin{array}{lll} \text{Variables} & x & \\ \text{Expressions} & e & ::= \quad \lambda x.\, e \mid e_1\, e_2 \mid x \end{array}$$

This is not the definition of the *concrete syntax* of a programming language, but a slightly more abstract form called *abstract syntax*. When we write down concrete expressions there are additional conventions and notations such as parentheses to avoid ambiguity.

1. Juxtaposition (which expresses application) is *left-associative* so that $x\, y\, z$ is read as $(x\, y)\, z$

2. $\lambda x.$ is a prefix whose scope extends as far as possible while remaining consistent with the parentheses that are present. For example, $\lambda x.\, (\lambda y.\, x\, y\, z)\, x$ is read as $\lambda x.\, ((\lambda y.\, (x\, y)\, z)\, x)$.

We say $\lambda x.\, e$ *binds* the variable x with scope e. Variables that occur in e but are not bound are called *free variables*, and we say that a variable x may occur free in an expression e. For example, y is free in $\lambda x.\, x\, y$ but not x. Bound variables can be renamed consistently in a term So $\lambda x.x + 5 = \lambda y.y + 5 = \lambda whatever.\,\mathsf{whatever} + 5$. Generally, we rename variables *silently* because we identify terms that differ only in the names of λ-bound variables. But, if we want to make the step explicit, we call it α-conversion.

$$\lambda x.\, e =_\alpha \lambda y.[y/x]e \quad \text{provided } y \text{ not free in } e$$

The proviso is necessary, for example, because $\lambda x.x\, y \neq \lambda y.y\, y$.

We capture the rule for function application with

$$(\lambda x.\, e_2)\, e_1 =_\beta [e_1/x]e_2$$

and call it *β-conversion*. Some care has to be taken for the substitution to be carried our correctly—we will return to this point later.

If we think beyond mere equality at *computation*, we see that β-conversion has a definitive direction: we apply is from left to right. We call this *β-reduction* and it is the engine of computation in the λ-calculus.

$$(\lambda x.\, e_2)\, e_1 \longrightarrow_\beta [e_1/x]e_2$$

3 Function Composition

One the most fundamental operation on functions in mathematics is to compose them. We might write

$$(f \circ g)(x) = f(g(x))$$

Having λ-notation we can first explicitly denote the result of composition (with some redundant parentheses)

$$f \circ g = \lambda x.\, f(g(x))$$

As a second step, we realize that $\circ$ itself is a function, taking two functions as arguments and returning another function. Ignoring the fact that it is usually written in infix notation, we define

$$\circ = \lambda f.\, \lambda g.\, \lambda x.\, f(g(x))$$

Now we can calculate, for example, the composition of the two functions we had at the beginning of the lecture. We note the steps where we apply

β-conversion.

$$\begin{array}{rl}
& (\circ\,(\lambda x.\, x+5))\,(\lambda y.\, 2*y+7) \\
= & ((\lambda f.\, \lambda g.\, \lambda x.\, f(g(x)))(\lambda x.\, x+5))\,(\lambda y.\, 2*y+7) \\
=_\beta & (\lambda g.\, \lambda x.\, (\lambda x.\, x+5)(g(x)))\,(\lambda y.\, 2*y+7) \\
=_\beta & \lambda x.\, (\lambda x.\, x+5)\,((\lambda y.\, 2*y+7)(x)) \\
=_\beta & \lambda x.\, (\lambda x.\, x+5)\,(2*x+7) \\
=_\beta & \lambda x.\, (2*x+7)+5 \\
= & \lambda x.\, 2*x+12
\end{array}$$

While this appears to go beyond the pure λ-calculus, we will see in Section 7 that we can actually encode natural numbers, addition, and multiplication. We can see that $\circ$ as an operator is not *commutative* because, in general, $\circ\, f\, g \neq \circ\, g\, f$. You may test your understanding by calculating $(\circ\,(\lambda y.\, 2*y+7))\,(\lambda x.\, x+5)$ and observing that it is different.

4 Identity

The simplest function is the identity function

$$I = \lambda x.\, x$$

We would expect that in general, $\circ\, I\, f = f = \circ\, f\, I$. Let's calculate one of these:

$$\begin{array}{rl}
& \circ\, I\, f \\
= & ((\lambda f.\, \lambda g.\, \lambda x.\, f(g(x)))\,(\lambda x.\, x))\, f \\
=_\beta & (\lambda g.\, \lambda x.\, (\lambda x.\, x)(g(x)))\, f \\
=_\beta & \lambda x.\, ((\lambda x.\, x)(f(x))) \\
=_\beta & \lambda x.\, f(x)
\end{array}$$

We see $\circ\, I\, f = \lambda x.\, f\, x$ but it does not appear to be equal to f. However, $\lambda x.\, f\, x$ and f would seem to be equal in the following sense: if we apply both sides to an arbitray expression e we get $(\lambda x.\, f\, x)\, e = f\, e$ on the left and $f\, e$ on the right. In other words, the two functions appear to be *extensionally* equal. We capture this by adding another rule to the calculus call η.

$$e =_\eta \lambda x.\, e\, x \quad \text{provided } x \text{ not free in } e$$

The proviso is necessary—can you find a counterexample to the equality if it is violated?

5 Summary of λ-Calculus

λ-Expressions.

$$\begin{array}{lll} \text{Variables} & x & \\ \text{Expressions} & e & ::= \quad \lambda x.\, e \mid e_1\, e_2 \mid x \end{array}$$

$\lambda x.\, e$ binds x with scope e, which is as large as possible while remaining consistent with the given parentheses. Juxtaposition $e_1\, e_2$ is left-associative.

Equality.

Substitution	$[e_1/x]e_2$			*(capture-avoiding, see Lecture 2)*
α-conversion	$\lambda x.\, e$	$=_\alpha$	$\lambda y.[y/x]e$	provided y not free in e
β-conversion	$(\lambda x.\, e_2)\, e_1$	$=_\beta$	$[e_1/x]e_2$	
η-conversion	$\lambda x.\, e\, x$	$=_\eta$	e	provided x not free in e

We generally apply α-conversion silently, identifying terms that differ only in the names of the bound variables. When we write $e = e'$ we allow $\alpha\beta\eta$-equality and the usual mathematical operations such as expanding a definition.

Reduction.

$$\beta\text{-reduction} \quad (\lambda x.\, e_2)\, e_1 \quad \longrightarrow_\beta \quad [e_1/x]e_2$$

6 Representing Booleans

Before we can claim the λ-calculus as a universal language for computation, we need to be able to represent *data*. The simplest nontrivial data type are the Booleans, a type with two elements: *true* and *false*. The general technique is to represent the values of a given type by *normal forms*, that is, expressions that cannot be reduced. Furthermore, they should be *closed*, that is, not contain any free variables. We need to be able to distinguish between two values, and in a closed expression that suggest introducing two bound variables. We then define rather arbitrarily one to be *true* and the other to be *false*

$$\begin{array}{rcl} \mathit{true} & = & \lambda x.\, \lambda y.\, x \\ \mathit{false} & = & \lambda x.\, \lambda y.\, y \end{array}$$

The next step will be to define *functions* on values of the type. Let's start with negation: we are trying to define a λ-expression *not* such that

$$\begin{array}{rcl} \textit{not true} & = & \textit{false} \\ \textit{not false} & = & \textit{true} \end{array}$$

We start with the obvious:

$$\textit{not} = \lambda b.\ \ldots$$

Now there are two possibilities: we could either try to apply b to some arguments, or we could build some λ-abstractions. In lecture, we followed the first path—you may want try the second as an exercise.

$$\textit{not} = \lambda b.\, b\,(\ldots)\,(\ldots)$$

We suggest two arguments to b, because b stands for a Boolean, and Booleans *true* and *false* both take two arguments. $\textit{true} = \lambda x.\, \lambda y.\, x$ will pick out the first of these two arguments and discard the second, so since we specified $\textit{not true} = \textit{false}$, the first argument to b should be *false*!

$$\textit{not} = \lambda b.\, b\, \textit{false}\ (\ldots)$$

Since $\textit{false} = \lambda x.\, \lambda y.\, y$ picks out the second argument and $\textit{not false} = \textit{true}$, the second argument to b should be *true*.

$$\textit{not} = \lambda b.\, b\, \textit{false true}$$

Now it is a simple matter to calculate that the computation of *not* applied to *true* or *false* completes in three steps and obtain the correct result.

$$\begin{array}{rcl} \textit{not true} & \longrightarrow^3_\beta & \textit{false} \\ \textit{not false} & \longrightarrow^3_\beta & \textit{true} \end{array}$$

We write $\longrightarrow^n_\beta$ for reduction in n steps, and $\longrightarrow^*_\beta$ for reduction in an arbitrary number of steps, including zero steps. In other words, $\longrightarrow^*_\beta$ is the reflexive and transitive closure of $\longrightarrow_\beta$.

As a next exercise we try conjuction. We want to define a λ-expression *and* such that

$$\begin{array}{rcl} \textit{and true true} & = & \textit{true} \\ \textit{and true false} & = & \textit{false} \\ \textit{and false true} & = & \textit{false} \\ \textit{and false false} & = & \textit{false} \end{array}$$

Learning from the negation, we start by guessing

$$\mathit{and} = \lambda b.\, \lambda c.\, b\,(\ldots)\,(\ldots)$$

where we arbitrarily put b first. If b is *true*, this will return the first argument. Looking at the equations we see that this should always be equal to the second argument.

$$\mathit{and} = \lambda b.\, \lambda c.\, b\, c\,(\ldots)$$

If b is *false* the result is always false, no matter what c is, so the second argument to b is just *false*.

$$\mathit{and} = \lambda b.\, \lambda c.\, b\, c\, \mathit{false}$$

Again, it is now a simple matter to verify the desired equations and that, in fact, the right-hand side of these equations is obtained by reduction.

We know we can represent all functions on Booleans returning Booleans once we have negation and conjunction. But we can also represent the more general conditional *if* with the requirements

$$\begin{array}{rcl} \mathit{if}\ \mathit{true}\ u\ w & = & u \\ \mathit{if}\ \mathit{false}\ u\ w & = & w \end{array}$$

Note here that the variable u and w stand for arbitrary λ-expressions and not just Booleans. From what we have seen before, the conditional is now easy to define:

$$\mathit{if} = \lambda b.\, \lambda u.\, \lambda w.\, b\, u\, w$$

Looking at the innermost abstraction, we have $\lambda w.\, (b\, u)\, w$ which is actually η-convertible to $b\, u$! Taking another step we arrive at

$$\begin{array}{rcl} \mathit{if} & = & \lambda b.\, \lambda u.\, \lambda w.\, b\, u\, w \\ & =_\eta & \lambda b.\, \lambda u.\, b\, u \\ & =_\eta & \lambda b.\, b \\ & = & I \end{array}$$

In other words, the conditional is just the identity function!

7 Representing Natural Numbers

Finite types such as Booleans are not particularly interesting. When we think about the computational power of a calculus we generally consider the *natural numbers* $0, 1, 2, \ldots$. We would like a representation $\overline{n}$ such that they are all distinct. We obtain this by thinking of the natural numbers are generated from zero by repeated application of the successor function. Since we want our representations to be closed we start with two abstractions: one (z) that stands for zero, and one (s) that stands for the successor function.

$$\begin{array}{lcl}
\overline{0} & = & \lambda z.\, \lambda s.\, z \\
\overline{1} & = & \lambda z.\, \lambda s.\, s\, z \\
\overline{2} & = & \lambda z.\, \lambda s.\, s\, (s\, z) \\
\overline{3} & = & \lambda z.\, \lambda s.\, s\, (s\, (s\, z)) \\
\ldots & & \\
\overline{n} & = & \lambda z.\, \lambda s.\, \underbrace{s\, (\ldots (s}_{n \text{ times}}\ z))
\end{array}$$

In other words, the representation $\overline{n}$ iterates its second argument n times over its first argument

$$\overline{n}\, x\, f = f^n(x)$$

where $f^n(x) = \underbrace{f(\ldots(f}_{n \text{ times}}(x)))$

The first order of business now is to define a successor function that satisfies *succ* $\overline{n} = \overline{n+1}$. As usual, there is more than one way to define it, here is one (throwing in the definition of *zero* for uniformity):

$$\begin{array}{lclcl}
\mathit{zero} & = & \overline{0} & = & \lambda z.\, \lambda s.\, z \\
\mathit{succ} & = & \lambda n.\, \overline{n+1} & = & \lambda n.\, \lambda z.\, \lambda s.\, s\, (n\, z\, s)
\end{array}$$

We cannot carry out the correctness proof in closed form as we did for the Booleans since there would be infinitely many cases to consider. Instead we calculate generically (using mathmetical notation and properties)

$$\begin{array}{cl}
 & \mathit{succ}\ \overline{n} \\
= & \lambda z.\, \lambda s.\, s\, (\overline{n}\, z\, s) \\
= & \lambda z.\, \lambda s.\, s\, (s^n(z)) \\
= & \lambda z.\, \lambda s.\, s^{n+1}(z) \\
= & \overline{n+1}
\end{array}$$

A more formal argument might use mathematical induction over n.

Using the iteration property we can now define other mathematical functions over the natural numbers. For example, addition of n and k iterates the successor function n times on k.

$$\mathit{plus} = \lambda n.\, \lambda k.\, n\ k\ \mathit{succ}$$

You are invited to verify the correctness of this definition by calculation. Similarly:

$$\begin{array}{lcl} \mathit{times} & = & \lambda n.\, \lambda k.\, n\ (\mathit{plus}\ k)\ \mathit{zero} \\ \mathit{exp} & = & \lambda b.\, \lambda e.\, e\ (\mathit{times}\ b)\ (\mathit{succ}\ \mathit{zero}) \end{array}$$

Everything appears to be going swimmingly until we hit the predecessor function defined by

$$\begin{array}{lcl} \mathit{pred}\ \overline{0} & = & \overline{0} \\ \mathit{pred}\ \overline{n+1} & = & \overline{n} \end{array}$$

You may try for a while to see if you can define the predecessor function, but it is very difficult. The problem seems to be that it is easy to define functions f using the schema of *iteration*

$$\begin{array}{lcl} f\ 0 & = & c \\ f\ (n+1) & = & g\ (f\ n) \end{array}$$

(namely: $f = \lambda n.\, n\, c\, g$), but not the so-called schema of *primitive recursion*

$$\begin{array}{lcl} f\ 0 & = & c \\ f\ (n+1) & = & g\ n\ (f\ n) \end{array}$$

because it is difficult to get access to n.

More about this and other properties and examples of the λ-calculus in Lecture 2.

References

[CR36] Alonzo Church and J.B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, May 1936.

[Tur36] Alan Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936. Published 1937.

Lecture Notes on Recursion, Binding, Substitution, and Computation

15-814: Types and Programming Languages
Ryan Kavanagh

Lecture 2
September 6, 2018

1 Introduction

Last time we saw that the set Λ of lambda terms was generated by the grammar:

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

We worked out some programming examples involving Booleans and natural numbers. We reasoned informally $\alpha\beta\eta$-equivalence and saw that we could go wrong if we were not careful about binding and substitution.

Today we will make the notions of equivalence and substitution precise. We will also see how to capture recursion.

1.1 Warm up

To make sure we remember how to use the untyped λ-calculus, let us do a few warm-up exercises. You can find the solutions on the next page.

Exercise 1 *Define the constant function* $\mathbf{K}$ *(also known as the* $\mathbf{K}$ *combinator) that satisfies* $\mathbf{K}xy = x$ *for all* x *and* y*.*

Exercise 2 *Define a test to see if a Church numeral is zero:*

$$\mathsf{isZero}\ \overline{0} = \mathsf{true} = \lambda x.\lambda y.x$$
$$\mathsf{isZero}\ \overline{n+1} = \mathsf{false} = \lambda x.\lambda y.y$$

It is interesting to consider what happens when we apply a λ-term to itself. Self-application is captured by the term $\omega = \lambda x.xx$. This may look odd at first sight, but it is a perfectly acceptable term. For example, $\omega\mathbf{K} = \lambda y.\mathbf{K}$ and $\omega\mathbf{I} = \mathbf{I}$. More interesting is $\Omega = \omega\omega$:

$$\Omega = (\lambda x.xx)(\lambda x.xx) = [(\lambda x.xx)/x](xx) = (\lambda x.xx)(\lambda x.xx).$$

The term Ω behaves exactly like an infinite loop!

Solutions Take $\mathbf{K} = \lambda x.\lambda y.x$ and $\mathsf{isZero} = \lambda n.n\ \mathsf{true}\ (\mathbf{K}\ \mathsf{false})$. The intuition for isZero is that if $\overline{n}$ is zero, then we should return true, and otherwise, $\overline{n}$'s "successor" parameter should be a function that constantly returns false.

2 Recursion

We would like to implement the factorial function

$$\mathsf{fact}\ \overline{n} = \mathsf{if}\ (\mathsf{isZero}\ \overline{n})(\overline{1})(\mathsf{mult}\ \overline{n}\ (\mathsf{fact}\ (\mathsf{pred}\ \overline{n})))$$

assuming we already have a predecessor function pred, which you will implement on Homework 0. We might start off with

$$\mathsf{fact} = \lambda n.\mathsf{if}(\mathsf{isZero}\ n)(\overline{1})(\mathsf{mult}\ n\ (\mathsf{fact}(\mathsf{pred}\ n))),$$

but we get stuck because we have an instance of fact on both sides. Let us consider what would happen if we factored out a fact on the right:

$$\mathsf{fact} = (\lambda f.\lambda n.\mathsf{if}\ (\mathsf{isZero}\ n)(\overline{1})(\mathsf{mult}\ n\ (f(\mathsf{pred}\ n))))\mathsf{fact}.$$

Letting

$$\Phi = \lambda f.\lambda n.\mathsf{if}\ (\mathsf{isZero}\ n)(\overline{1})(\mathsf{mult}\ n\ (f(\mathsf{pred}\ n))),$$

we see that fact can be expressed as a fixed point of Φ, that is, $\Phi(\mathsf{fact}) = \mathsf{fact}$. Can we find such a fixed point?

Theorem 1 (Fixed point theorem) *For all $F \in \Lambda$ there exists an $X \in \Lambda$ such that $FX = X$.*

Proof: Earlier we encountered the divergent term $\Omega = (\lambda x.xx)(\lambda x.xx)$, where applying the β rule gave Ω again:

$$(\lambda x.xx)(\lambda x.xx) = [(\lambda x.xx)/x](xx) = (\lambda x.xx)(\lambda x.xx).$$

This infinite unfolding behaviour is similar to what we want in a fixed point: if X is a fixed point of F, then $X = FX = F(FX) = \cdots$. Suppose we inserted an F at the beginning of each of the function bodies:

$$(\lambda x.F(xx))(\lambda x.F(xx)) = [\lambda x.F(xx)x](F(xx)) = F((\lambda x.F(xx))(\lambda x.F(xx))).$$

Take $X = (\lambda x.F(xx))(\lambda x.F(xx))$ and we are done. □

We can abstract over the F in the above proof to get the $\mathbf{Y}$ *combinator*[1] that constructs the fixed point of any term to which it is applied:

Corollary 2 *Let* $\mathbf{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$*. Then* $\mathbf{Y}F = F(\mathbf{Y}F)$ *for all* $F \in \Lambda$*.*

We can now define our factorial function: $\mathsf{fact} = \mathbf{Y}\Phi$. Unfolding the definition and using Corollary 2, we see that this is actually what we wanted:

$$\begin{aligned}
\mathsf{fact}\ \overline{n} &= \mathbf{Y}\Phi\overline{n} \\
&= \Phi(\mathbf{Y}\Phi)\overline{n} \\
&= \mathsf{if}\ (\mathsf{isZero}\ \overline{n})(\overline{1})(\mathsf{mult}\ \overline{n}\ (\mathbf{Y}\Phi(\mathsf{pred}\ \overline{n}))) \\
&= \mathsf{if}\ (\mathsf{isZero}\ \overline{n})(\overline{1})(\mathsf{mult}\ \overline{n}\ (\mathsf{fact}(\mathsf{pred}\ \overline{n}))).
\end{aligned}$$

3 Binding and substitution

We need to be careful when we substitute to ensure that we do not accidentally bind (or *capture*) free variables. We say that an occurrence of the variable x is *bound* if it is in the scope of an abstractor λx, otherwise it is *free*. The set of free variables $\mathsf{fv}(e)$ in a term e is recursively defined on the structure of the term:

$$\begin{aligned}
\mathsf{fv}(x) &= \{x\} \\
\mathsf{fv}(\lambda x.e) &= \mathsf{fv}(e) \setminus \{x\} \\
\mathsf{fv}(e_1 e_2) &= \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2).
\end{aligned}$$

We say that a term is *closed* or a *combinator* if it has no free variables.

For the rest of this lecture, we will use the symbol $\equiv$ to mean that two terms are syntactically equal.

[1] The startup incubator was named after this combinator.

Given a term $\lambda x.e$, a *change of bound variable* is the result of $\lambda y.[y/x]e$ when y does not appear in e. Because y does not appear in e, we do not need to worry about capture. In this case, we say that $\lambda x.e$ and $\lambda y.[y/x]e$ are α-congruent: $\lambda x.e =_\alpha \lambda y.[y/x]e$. More generally, we say that two terms e_1 and e_2 are α-congruent, $e_1 =_\alpha e_2$, if e_1 can be obtained from e_2 through a sequence changes of bound variable. For example,

$$\lambda x.x(\lambda y.yx)z =_\alpha \lambda w.w(\lambda y.yw)z \neq_\alpha \lambda z.z(\lambda y.yz)z.$$

Changing bound variables is sometimes called *α-varying*. We identify α-congruent terms, that is, we treat them as though they were syntactically equal. Thanks to this identification, we can adopt the *variable convention*: we always assume the bound variables are chosen to be different from all of the free variables. With the variable convention, we can safely substitute in the naïve manner. (We implicitly assumed the variable convention in the proof of theorem 1. Where?[2])

We can now make the definition of substitution explicit:

$$\begin{aligned}
[e/x]x &\equiv e \\
[e/x]y &\equiv y && (\text{if } x \not\equiv y) \\
[e_1/x](\lambda y.e_2) &\equiv \lambda y.[e_1/x]e_2 && (\text{if } x \not\equiv y \text{ and } y \notin \mathsf{fv}(e_1))^3 \\
[e_1/x](e_2 e_3) &\equiv ([e_1/x]e_2)([e_1/x]e_3)
\end{aligned}$$

It is a good exercise to think carefully about this definition and how it interacts with the variable convention.

4 Reduction and computation

So far we have treated the λ-calculus as an equational theory. This is not a satisfactory notion of computation, because we have no notion of making progress or of termination (of knowing when we have reached a "value").

To capture the idea of making some form of directed "progress", we use *reductions*. *β-reduction* is the least relation $\rightarrow_\beta$ on Λ satisfying for all $e_1, e_2 \in \Lambda$:

- $(\lambda x.e_1)e_2 \rightarrow_\beta [e_2/x]e_1$,
- if $e_1 \rightarrow_\beta e_1'$, then $e_1 e_2 \rightarrow_\beta e_1' e_2$, $e_2 e_1 \rightarrow_\beta e_2 e_1'$, and $\lambda x.e_1 \rightarrow_\beta \lambda x.e_1'$.

[2] We implicitly assumed $x \notin \mathsf{fv}(F)$.

[3] These conditions are redundant by the variable convention.

Let $\to_\beta^*$ be the reflexive, transitive closure of $\to_\beta$, i.e., the least relation on Λ inductively defined by:

- $e \to_\beta^* e$ for all e,
- if $e_1 \to_\beta e_1'$ and $e_1' \to_\beta^* e_2$, then $e_1 \to_\beta^* e_2$.

We say that M is in β-normal from if M cannot be β-reduced.

β-reduction satisfies the *confluence* property that we foreshadowed last time, from which we can deduce that every λ-term has a unique β-normal form. A relation $\to$ is confluent if whenever $e_0 \to^* e_1$ and $e_0 \to^* e_2$, there exists an e_3 such that $e_1 \to^* e_3$ and $e_2 \to^* e_3$. Pictorially,

$$\begin{tikzcd}
& e_0 \arrow[dl, "*"] \arrow[dr, "*"] & \\
e_1 \arrow[dr, "*"] & & e_2 \arrow[dl, "*"] \\
& e_3 &
\end{tikzcd}$$

Theorem 3 (Church-Rosser) *β-reduction is confluent.*

However, β-reduction is not what we want as a notion of computation. The reason is that β-reduction behaves a bit like equality: it can be applied anywhere in a term. As a result, operationally it is highly non-deterministic. Depending on how you apply β-reduction, you could either reach a β-normal form or fail to ever terminate. Consider for example the λ-term $(\lambda x.y)\Omega$. Applying β-reduction on the outermost β-redex gives $(\lambda x.y)\Omega \to_\beta y$. In contrast, if we repeatedly apply β-reduction to Ω, we never reach a β-normal form: $(\lambda x.y)\Omega \to_\beta (\lambda x.y)\Omega \to_\beta (\lambda x.y)\Omega \to_\beta \cdots$.

To make reduction deterministic, we use *reduction strategies*. The simplest of these is call-by-name (CBN) reduction, $\to_{CBN}$, defined to be the least relation on Λ satisfying for all $e_1, e_2 \in \Lambda$:

- $(\lambda x.e_1)e_2 \to_{CBN} [e_2/x]e_1$, and
- if $e_1 \to_{CBN} e_1'$, then $e_1 e_2 \to_{CBN} e_1' e_2$.

The intuition is that we eagerly reduce as far to the left as possible. Observe that this reduction strategy is deterministic: if $e_1 \to_{CBN} e_2$ and $e_1 \to_{CBN} e_2'$, then $e_2 \equiv e_2'$.

Theorem 4 *If $e_1 \to_{CBN} e_2$, then $e_1 \to_\beta e_2$. The converse is false.*

Proof: The first part is obvious. The term $(\lambda x.y)\Omega$ is a counter-example to the converse: $(\lambda x.y)\Omega \to_\beta (\lambda x.y)\Omega$, but $(\lambda x.y)\Omega \not\to_{CBN} (\lambda x.y)\Omega$. $\square$

References

[Bar12] Henk P. Barendregt. *The Lambda Calculus, its Syntax and Semantics*, volume 40 of *Studies in Logic*. College Publications, 2012.

Lecture Notes on
Simple Types

15-814: Types and Programming Languages
Frank Pfenning

Lecture 3
Tuesday, September 11, 2018

1 Introduction

We have experienced the expressive power of the λ-calculus in multiple ways. We followed the slogan of *data as functions* and represented types such as Booleans and natural numbers. On the natural numbers, we were able to express the same set of partial functions as with Turing machines, which gave rise to the Church-Turing thesis that these are all the effectively computable functions.

On the other hand, Church's original purpose of the pure calculus of functions was a new foundations of mathematics distinct from set theory [Chu32, Chu33]. Unfortunately, this foundation suffered from similar paradoxes as early attempts at set theory and was shown to be *inconsistent*, that is, every proposition has a proof. Church's reaction was to return to the ideas by Russell and Whitehead [WR13] and introduce *types*. The resulting calculus, called *Church's Simple Theory of Types* [Chu40] is much simpler that Russell and Whitehead's *Ramified Theory of Types* and, indeed, serves well as a foundation for (classical) mathematics.

We will follow Church and introduce *simple types* as a means to classify λ-expressions. An important consequence is that we can recognize the representation of Booleans, natural numbers, and other data types and distinguish them from other forms of λ-expressions. We also explore how typing interacts with computation.

2 Simple Types, Intuitively

Since our language of expression consists only of λ-abstraction to form functions, juxtaposition to apply functions, and variables, we would expect our language of types τ to just contain $\tau ::= \tau_1 \to \tau_2$. This type might be considered "empty" since there is no base case, so we add type variables α, β, γ, etc.

$$\begin{array}{lll} \text{Type variables} & \alpha & \\ \text{Types} & \tau & ::= \quad \tau_1 \to \tau_2 \mid \alpha \end{array}$$

We follow the convention that the function type constructor "$\to$" is *right-associative*, that is, $\tau_1 \to \tau_2 \to \tau_3 = \tau_1 \to (\tau_2 \to \tau_3)$.

We write $e : \tau$ if expression e has type τ. For example, the identity function takes an argument of arbitrary type α and returns a result of the same type α. But the type is not unique. For example, the following two hold:

$$\begin{array}{lcl} \lambda x.\, x & : & \alpha \to \alpha \\ \lambda x.\, x & : & (\alpha \to \beta) \to (\alpha \to \beta) \end{array}$$

What about the Booleans? *true* $= \lambda x.\, \lambda y.\, x$ is a function that takes an argument of some arbitrary type α, a second argument y of a potentially different type β and returns a result of type α. We can similarly analyze *false*:

$$\begin{array}{lclcl} \mathit{true} & = & \lambda x.\, \lambda y.\, x & : & \alpha \to (\beta \to \alpha) \\ \mathit{false} & = & \lambda x.\, \lambda y.\, y & : & \alpha \to (\beta \to \beta) \end{array}$$

This looks like bad news: how can we capture the Booleans by their type if *true* and *false* have a different type? We have to realize that types are not unique and we can indeed find a type that is shared by *true* and *false*:

$$\begin{array}{lclcl} \mathit{true} & = & \lambda x.\, \lambda y.\, x & : & \alpha \to (\alpha \to \alpha) \\ \mathit{false} & = & \lambda x.\, \lambda y.\, y & : & \alpha \to (\alpha \to \alpha) \end{array}$$

The type $\alpha \to (\alpha \to \alpha)$ then becomes our candidate as a type of Booleans in the λ-calculus. Before we get there, we formalize the type system so we can rigorously prove the right properties.

3 The Typing Judgment

We like to formalize various judgments about expressions and types in the form of inference rules. For example, we might say

$$\frac{e_1 : \tau_2 \to \tau_1 \quad e_2 : \tau_2}{e_1\ e_2 : \tau_1}$$

We usually read such rules from the conclusion to the premises, pronouncing the horizontal line as *"if"*:

> *The application $e_1\ e_2$ has type τ_1 if e_1 maps arguments of type τ_2 to results of type τ_1 and e_2 has type τ_2.*

When we arrive at functions, we might attempt

$$\frac{x_1 : \tau_1 \quad e_2 : \tau_2}{\lambda x_1.\, e_2 : \tau_1 \to \tau_2}\ ?$$

This is (more or less) Church's approach. It requires that each variable x intrinsically has a type that we can check, so probably we should write x^τ. In modern programming languages this can be bit awkward because we might substitute for type variables or apply other operations on types, so instead we record the types of variable in a *typing context*.

$$\text{Typing context} \quad \Gamma \quad ::= \quad x_1 : \tau_1, \ldots, x_n : \tau_n$$

Critically, we always assume:

> *All variables declared in a context are distinct.*

This avoids any ambiguity when we try to determine the type of a variable. The typing judgment now becomes

$$\Gamma \vdash e : \tau$$

where the context Γ contains declarations for the free variables in e. It is defined by the following three rules

$$\frac{\Gamma, x_1 : \tau_2 \vdash e_2 : \tau_2}{\Gamma \vdash \lambda x_1.\, e_2 : \tau_1 \to \tau_2}\ \mathsf{lam} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \mathsf{var}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_1}\ \mathsf{app}$$

As a simple example, let's type-check *true*. Note that we always construct such derivations bottom-up, starting with the final conclusion, deciding on rules, writing premises, and continuing.

$$\frac{\dfrac{\dfrac{}{x : \alpha, y : \alpha \vdash x : \alpha}\ \mathsf{var}}{x : \alpha \vdash \lambda y.\, x : \alpha \to \alpha}\ \mathsf{lam}}{\cdot \vdash \lambda x.\, \lambda y.\, x : \alpha \to (\alpha \to \alpha)}\ \mathsf{lam}$$

How about the expression $\lambda x.\, \lambda x.\, x$? This is α-equivalent to $\lambda x.\, \lambda y.\, y$ and therefore should check (among other types) as having type $\alpha \to (\beta \to \beta)$. It appears we get stuck:

$$\dfrac{\dfrac{??}{x : \alpha \vdash \lambda x.\, x : \beta \to \beta}\ \mathsf{lam??}}{\cdot \vdash \lambda x.\, \lambda x.\, x : \alpha \to (\beta \to \beta)}\ \mathsf{lam}$$

The worry is that applying the rule lam would violate our presupposition that no variable is declared more than once and $x : \alpha, x : \beta \vdash x : \beta$ would be ambiguous. But we said we can "*silently*" apply α-conversion, so we do it here, renaming x to x'. We can then apply the rule:

$$\dfrac{\dfrac{\dfrac{}{x : \alpha, x' : \beta \vdash x' : \beta}\ \mathsf{var}}{x : \alpha \vdash \lambda x.\, x : \beta \to \beta}\ \mathsf{lam}}{\cdot \vdash \lambda x.\, \lambda x.\, x : \alpha \to (\beta \to \beta)}\ \mathsf{lam}$$

4 Characterizing the Booleans

We would now like to show that the representation of the Booleans is in fact correct. We go through a sequence of conjectures to (hopefully) arrive at the correct conclusion.

Conjecture 1 (Representation of Booleans, v1)
If $\cdot \vdash e : \alpha \to (\alpha \to \alpha)$ *then* $e =$ *true or* $e =$ *false.*

If by "=" we mean mathematical equality that this is false. For example,

$$\cdot \vdash (\lambda z.\, z)\ (\lambda x.\, \lambda y.\, x) : \alpha \to (\alpha \to \alpha)$$

but the expression $(\lambda z.\, z)\ (\lambda x.\, \lambda y.\, x)$ represents neither true nor false. But it is in fact β-convertible to *true*, so we might loosen our conjecture:

Conjecture 2 (Representation of Booleans, v2)
If $\cdot \vdash e : \alpha \to (\alpha \to \alpha)$ *then* $e =_\beta$ *true or* $e =_\beta$ *false.*

This speaks to equality, but since we are interested in programming languages and computation, we may want to prove something ostensibly stronger. Recall that $e \longrightarrow_\beta^* e'$ means that we can β-reduce e to e' in an arbitrary number of steps (including zero). In other words, $\longrightarrow_\beta^*$ is the *reflexive* and *transitive* closure of $\longrightarrow_\beta$.

Conjecture 3 (Representation of Booleans, v3)
If $\cdot \vdash e : \alpha \rightarrow (\alpha \rightarrow \alpha)$ *then* $e \longrightarrow_{\beta}^{*}$ *true or* $e \longrightarrow_{\beta}^{*}$ *false.*

This is actually quite difficult to prove, so we break it down into several propositions, some of which we can actually prove. The first one concerns *normal forms*, that is, expressions that cannot be β-reduced. They play the role that *values* play in many programming language.

Conjecture 4 (Representation of Booleans as normal forms)
If $\cdot \vdash e : \alpha \rightarrow (\alpha \rightarrow \alpha)$ *and* e *is a normal form, then* $e =$ *true or* $e =$ *false.*

We will later combine this with the following theorems which yiels correctness of the representation of Booleans. These theorems are quite general (not just on Booleans), and we will see multiple versions of them in the remainder of the course.

Theorem 5 (Termination) *If* $\Gamma \vdash e : \tau$ *then* $e \longrightarrow_{\beta}^{*} e'$ *for a normal form* e'*.*

Theorem 6 (Subject reduction) *If* $\Gamma \vdash e : \tau$ *and* $e \longrightarrow_{\beta} e'$ *then* $\Gamma \vdash e' : \tau$*.*

We will prove subject reduction on Lecture 4, and we may or may not prove termination in a later lecture. Today, we will focus on the the correctness of the representation of normal forms.

5 Normal Forms

Recall the rules for reduction. We refer to the first three rules as *congruence rules* because they allow the reduction of a subterms.

$$\frac{e \longrightarrow e'}{\lambda x.\, e \longrightarrow \lambda x.\, e'}\ \mathsf{lm} \qquad \frac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2}\ \mathsf{ap}_1 \qquad \frac{e_1 \longrightarrow e_2'}{e_1\, e_2 \longrightarrow e_1\, e_2'}\ \mathsf{ap}_2$$

$$\frac{}{(\lambda x.\, e_1)\, e_2 \longrightarrow [e_2/x]e_1}\ \beta$$

A *normal form* is an expression e such that there does not exists an e' such that $e \longrightarrow e'$. Basically, we have to rule out β-redices $(\lambda x.\, e_1)\, e_2$, but we would like to describe normal forms via inference rules to we can easily prove inductive theorems. This definition should capture the form

$$\lambda x_1.\ \ldots\ \lambda x_n.\, ((x\, e_1) \ldots e_k)$$

where $e_1, \ldots e_k$ are again in normal form.

We can capture this intuition in two parts: the definition of normal forms allows us to descend through arbitrarily many λ-abstractions. We write e *nf* for the judgment that e is in normal form.

$$\frac{e\ \mathit{nf}}{\lambda x.\, e\ \mathit{nf}}$$

At some point we have to switch to an application. The key part of this is that if we keep looking at the function part it may again be an application, or it may be a variable, but it cannot be a λ-abstraction. We call such expressions *neutral* because when they are applied to an argument they do not reduce but remain a normal form.

$$\frac{e\ \mathit{neutral}}{e\ \mathit{nf}} \qquad \frac{e_1\ \mathit{neutral} \quad e_2\ \mathit{nf}}{e_1\ e_2\ \mathit{neutral}} \qquad \frac{}{x\ \mathit{neutral}}$$

6 Representation as Normal Forms

In the next lecture we will prove that every expression either reduces or is a normal form. In this lecture we will be concerned with the property that *closed normal forms* of type $\alpha \to (\alpha \to \alpha)$ are exactly *true* and *false*.

Many of our proofs will go by induction, either on the structure of expressions or the structure of deductions using inference rules. The latter is called *rule induction*. It states that if every rule preserves the property of a judgment we want to show, then the property must always be true since the rules are the *only* way to establish a judgment. We get to assume the property for the premise of the rule (the *induction hypothesis*) and have to show it for the conclusion.

A special case of rule induction is proof by cases on the rules. We try here a rule by cases.

Conjecture 7 (Representation of Booleans as normal forms, v1)
For any expression e, if $\cdot \vdash e : \alpha \to (\alpha \to \alpha)$ and e nf then e = true or e = false.

Proof attempt: By cases on the deduction of e *nf*. We analyze each rule in turn.

Case:

$$\frac{e_2\ \mathit{nf}}{\lambda x_1.\, e_2\ \mathit{nf}}$$

where $e = \lambda x_1 .\, e_2$. Let's gather our knowledge.

$\cdot \vdash \lambda x_1 .\, e_2 : \alpha \to (\alpha \to \alpha)$ Assumption

What can we do with this knowledge? Looking at the typing rules we see that there is only one possible rule that may have this conclusion. Since the judgment holds by assumption, it must have been inferred with this rule and the premise of that rule must also hold. We call this step *inversion*.

$x_1 : \alpha \vdash e_2 : \alpha \to \alpha$ By inversion

We also know that e_2 is a normal form (the premise of the rule in this case), so now we'd like to show that $e_2 = \lambda x_2 .\, x_1$ or $e_2 = \lambda x_2 .\, x_2$. We choose the generalize the theorem to make this property explicit as well (see version 2).

First, though, let's consider the second case.

Case:

$$\frac{e \; \textit{neutral}}{e \; \textit{nf}}$$

Again, restating our assumption, we have

$\cdot \vdash e : \alpha \to (\alpha \to \alpha)$ By assumption

But there is no closed neutral expression because at the head of the left-nested expressions would have to be a variable, of which we have none. This property will be an instance of a more general property we will need shortly.

□

Conjecture 8 (Representation of Booleans as normal forms, v2)

(i) *If* $\cdot \vdash e : \alpha \to (\alpha \to \alpha)$ *and* e *nf then* $e = \textit{true}$ *or* $e = \textit{false}$.

(ii) *If* $x : \alpha \vdash e : \alpha \to \alpha$ *and* e *nf then* $e = \lambda y .\, x$ *or* $e = \lambda y .\, y$.

(iii) *If* $x : \alpha, y : \alpha \vdash e : \alpha$ *and nf then* $e = x$ *or* $e = y$.

Proof attempt: By cases on the given deduction of e *nf*. Now parts (i) and (ii) proceed as in the previous attempt, (i) relying on (ii) and (ii) relying on (iii), analyzing the cases of normal forms. The last one is interesting: we know that e cannot be λ-abstraction because that would have function type, so it must be a neutral expression. But if it is a neutral expression it should have to be one of the variables x or y: it cannot be an application because we would have to find a variable of function type at the head of the left-nested applications. So we need a lemma before we can complete the proof. □

The insight in this proof attempt gives rise to the following lemma.

Lemma 9 (Neutral expressions)
If $x_1 : \alpha_1, \ldots, x_n : \alpha_n \vdash e : \tau$ and e neutral then $e = x_i$ and $\tau = \alpha_i$ for some i.

Proof: By rule induction over the definition of e *neutral*

Case:

$$\frac{}{x \textit{ neutral}}$$

where $e = x$.

$x_1 : \alpha_1, \ldots, x_n : \alpha_n \vdash x : \tau$ Assumption
$x : \tau \in (x_1 : \alpha_1, \ldots, x_n : \alpha_n)$ By inversion
$x = x_i$ and $\tau = \alpha_i$ for some i

Case:

$$\frac{e_1 \textit{ neutral} \quad e_2 \textit{ nf}}{e_1\, e_2 \textit{ neutral}}$$

and $e = e_1\, e_2$. This case is impossible:

$x_1 : \alpha_1, \ldots, x_n : \alpha_n \vdash e_1\, e_2 : \tau$ Assumption
$x_1 : \alpha_1, \ldots, x_n : \alpha_n \vdash e_1 : \tau_2 \to \tau$
and $x_1 : \alpha_1, \ldots, x_n : \alpha_n \vdash e_2 : \tau_2$ for some τ_2 By inversion
$e_1 = x_i$ and $\tau_2 \to \tau = \alpha_i$ By induction hypothesis
Contradiction, since $\tau_2 \to \tau \neq \alpha_i$

□

Now we are ready to prove the representation theorem.

Theorem 10 (Representation of Booleans as normal forms, v3)

(i) *If* $\cdot \vdash e : \alpha \to (\alpha \to \alpha)$ *and* e *nf then* $e = \mathit{true}$ *or* $e = \mathit{false}$.

(ii) *If* $x : \alpha \vdash e : \alpha \to \alpha$ *and* e *nf then* $e = \lambda y.\, x$ *or* $e = \lambda y.\, y$.

(iii) *If* $x : \alpha, y : \alpha \vdash e : \alpha$ *and nf then* $e = x$ *or* $e = y$.

Proof: By cases on the given deduction of e *nf*.

Case for (i):

$$\frac{e_2\ \mathit{nf}}{\lambda x.\, e_2\ \mathit{nf}}$$

where $e = \lambda x.\, e_2$.

$\cdot \vdash \lambda x.\, e_2 : \alpha \to (\alpha \to \alpha)$ Assumption
$x : \alpha \vdash e_2 : \alpha \to \alpha$ By inversion
$e_2 = \lambda y.\, x$ or $e_2 = \lambda y.\, y$ By part (ii)
$e = \lambda x.\, \lambda y.\, x$ or $e = \lambda x.\, \lambda y.\, y$ since $e = \lambda x.\, e_2$

Case for (i):

$$\frac{e\ \mathit{neutral}}{e\ \mathit{nf}}$$

$\cdot \vdash e : \alpha \to (\alpha \to \alpha)$ Assumption
Impossible, by the *neutral expression lemma* (9)

Cases for (ii): analogous to the cases for (i), appealing to part (iii).

Case for (iii):

$$\frac{e_2\ \mathit{nf}}{\lambda z.\, e_2\ \mathit{nf}}$$

$x : \alpha, y : \alpha \vdash \lambda z.\, e_2 : \alpha$ Assumption
Impossible by inversion (no typing rule matches this conclusion)

Case for (iii):

$$\frac{e\ \mathit{neutral}}{e\ \mathit{nf}}$$

$x : \alpha, y : \alpha \vdash e : \alpha$ Assumption
$e = x$ and or $e = y$ by the neutral expression lemma (9)

□

References

[Chu32] A. Church. A set of postulates for the foundation of logic I. *Annals of Mathematics*, 33:346–366, 1932.

[Chu33] A. Church. A set of postulates for the foundation of logic II. *Annals of Mathematics*, 34:839–864, 1933.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[WR13] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, 1910–13. 3 volumes.

Lecture Notes on
Subject Reduction and Normal Forms

15-814: Types and Programming Languages
Frank Pfenning

Lecture 4
September 13, 2018

1 Introduction

In the last lecture we proved some key aspect of a *representation theorem* for Booleans, namely that the closed normal forms type $\alpha \to (\alpha \to \alpha)$ are either *true* $= \lambda x.\, \lambda y.\, x$ or *false* $= \lambda x.\, \lambda y.\, y$. But is our characterization of normal forms correct? We would like to prove that any expression e is either a normal form (that is, satisfies e *nf* or can be reduced. We prove this theorem first. Then we show that typing is preserved under reduction, which means that if we start with an expression e of type τ and we reduce it all the way to a normal form e', the e' will still have type τ. For the special case where $\tau = \alpha \to (\alpha \to \alpha)$ which means that any expression e of type τ that has a normal form represents a Boolean.

2 Reduction and Normal Form

Recall our characterization of reduction from Lecture 2, written here as a collection of inference rules.

$$\frac{e \longrightarrow e'}{\lambda x.\, e \longrightarrow \lambda x.\, e'}\ \mathsf{lm} \qquad \frac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2}\ \mathsf{ap}_1 \qquad \frac{e_1 \longrightarrow e_2'}{e_1\, e_2 \longrightarrow e_1\, e_2'}\ \mathsf{ap}_2$$

$$\frac{}{(\lambda x.\, e_1)\, e_2 \longrightarrow [e_2/x]e_1}\ \beta$$

And our characterization of normal forms:

$$\frac{e\ \mathit{nf}}{\lambda x.\, e\ \mathit{nf}}\ \mathsf{nf/lam} \qquad \frac{e\ \mathit{neutral}}{e\ \mathit{nf}}\ \mathsf{nf/ne}$$

$$\frac{e_1\ \mathit{neutral} \quad e_2\ \mathit{nf}}{e_1\ e_2\ \mathit{neutral}}\ \mathsf{ne/app} \qquad \frac{}{x\ \mathit{neutral}}\ \mathsf{ne/var}$$

The correctness of this characterization consists of two parts, of which we will prove one: (i) every term either reduces or is a normal form, and (ii) normal forms don't reduce.

Theorem 1 (Reduction and normal forms)
For every expression e, either $e \longrightarrow e'$ for some e', or e nf.

Proof: We are only given an expression e, so the proof is likely by induction on the structure of e. Let's try! We write $e \longrightarrow$ if there is some e' such that $e \longrightarrow e'$.

Case: $e = x$. Then

$x\ \mathit{neutral}$	By rule ne/var
$x\ \mathit{nf}$	By rule nf/ne

Case: $e = \lambda x.\, e_1$. Then

Either $e_1 \longrightarrow$ or $e_1\ \mathit{nf}$	By ind.hyp. on e'
$e_1 \longrightarrow$	First subcase
$e = \lambda x.\, e_1 \longrightarrow$	By rule lm
$e_1\ \mathit{nf}$	Second subcase
$e = \lambda x.\, e_1\ \mathit{nf}$	By rule nf/lam

Case: $e = e_1\, e_2$. Then

Either $e_1 \longrightarrow$ or $e_1\ \mathit{nf}$	By ind.hyp. on e_1
$e_1 \longrightarrow$	First subcase
$e_1\ e_2 \longrightarrow$	By rule ap_1
$e_1\ \mathit{nf}$	Second subcase

Either $e_1 = \lambda x.\, e_1'$ or e_1 *neutral*	By inversion on e_1 *nf*
$e_1 = \lambda x.\, e_1'$	First sub^{2}case
$e = e_1\, e_2 = (\lambda x.\, e_1')\, e_2 \longrightarrow$	By rule β
e_1 *neutral*	Second sub^{2}case
Either $e_2 \longrightarrow$ or e_2 *nf*	By ind.hyp. on e_2
$e_2 \longrightarrow$	First sub^{3}case
$e = e_1\, e_2 \longrightarrow$	By rule ap_2
e_2 *nf*	Second sub^{3}case
$e = e_1\, e_2$ *neutral*	By rule $\mathsf{ne/app}$

□

The next in our quest for a representation theorem will be to show that types are preserved under reduction. We start with $e : \tau$ and want to know that if $e \longrightarrow^* e'$ where e' is a normal form, then $e' : \tau$. This is almost universally proven by showing that a single step of reduction preserves types, from which the above follows by a simple induction over the reduction sequence.

We begin by conjecturing a version of the theorem for closed expression of arbitrary type, because these are the expressions we are ultimately interested in.

Conjecture 2 (Subject reduction, v1)
If $\cdot \vdash e : \tau$ *and* $e \longrightarrow e'$ *then* $\cdot \vdash e' : \tau$.

Proof attempt: In this conjecture, we are given both an expression e and a reduction $e \longrightarrow e'$, so a priori there are three possible inductions: rule induction on $\cdot \vdash e : \tau$, rule induction on $e \longrightarrow e'$, and induction on the structure of e. Having done this kind of proof about a gazillion times, I know it should go by rule induction on $e \longrightarrow e'$.

Case:

$$\frac{e_1 \longrightarrow e_1'}{\lambda x.\, e_1 \longrightarrow \lambda x.\, e_1'}\ \mathsf{lm}$$

where $e = \lambda x.\, e_1'$. We gather knowledge, and then apply inversion because we cannot yet apply the induction hypothesis.

$\cdot \vdash \lambda x.\, e_1 : \tau$	Assumption
$x : \tau_2 \vdash e_1 : \tau_1$ and $\tau = \tau_2 \to \tau_1$ for some τ_1 and τ_2	By inversion

It looks like we are ready for an appeal to the induction hypothesis, but we are stuck because the context in the typing of e_1 the context is not empty! We realize have to generalize the theorem to allow arbitrary contexts Γ.

□

Theorem 3 (Subject reduction, v2)
If $\Gamma \vdash e : \tau$ *and* $e \longrightarrow e'$ *then* $\Gamma' \vdash e' : \tau$*.*

Proof: By rule induction on the deduction of $e \longrightarrow e'$.

Case:

$$\frac{e_1 \longrightarrow e_1'}{\lambda x.\, e_1 \longrightarrow \lambda x.\, e_1'}\ \mathsf{lm}$$

where $e = \lambda x.\, e_1'$.

$\Gamma \vdash \lambda x.\, e_1 : \tau$	Assumption
$\Gamma, x : \tau_2 \vdash e_1 : \tau_1$ and $\tau = \tau_2 \to \tau_1$ for some τ_1 and τ_2	By inversion
$\Gamma, x : \tau_2 \vdash e_1' : \tau_1$	By induction hypothesis
$\Gamma \vdash \lambda x.\, e_1' : \tau_2 \to \tau_1$	By rule lam

Case:

$$\frac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2}\ \mathsf{ap}_1$$

where $e = e_1\, e_2$. We start again by restating what we know in this case and then apply inversion.

$\Gamma \vdash e_1\, e_2 : \tau$	Assumption
$\Gamma \vdash e_1 : \tau_2 \to \tau$ and $\Gamma \vdash e_2 : \tau_2$ for some τ_2	By inversion

At this point we have a type for e_1 and a reduction for e_1, so we can apply the induction hypothesis.

$\Gamma \vdash e_1' : \tau_2 \to \tau$ By ind.hyp.

Now we can just apply the typing rule for application. Intuitively, in the typing for $e_1\, e_2$ we have replaced e_1 by e_1', which is okay since they e_1' has the type of e_1.

$\Gamma \vdash e_1'\, e_2 : \tau$ By rule lam

Case:

$$\frac{e_2 \longrightarrow e_2'}{e_1\; e_2 \longrightarrow e_1'\; e_2}\ \mathsf{ap}_2$$

where $e = e_1\, e_2$. This proceeds completely analogous to the previous case.

Case:

$$\frac{}{(\lambda x.\, e_1)\, e_2 \longrightarrow [e_2/x]e_1}\ \beta$$

where $e = (\lambda x.\, e_1)\, e_2$. In this case we apply inversion twice, since the structure of e is two levels deep.

$\Gamma \vdash (\lambda x.\, e_1)\, e_2 : \tau$ Assumption
$\Gamma \vdash \lambda x.\, e_1 : \tau_2 \to \tau$
and $\Gamma \vdash e_2 : \tau_2$ for some τ_2 By inversion
$\Gamma, x : \tau_2 \vdash e_1 : \tau$ By inversion

At this point we are truly stuck, because there is no obvious way to complete the proof.

To Show: $\Gamma \vdash [e_2/x]e_1 : \tau$

Fortunately, the gap that presents itself is exactly the content of the *substitution property*, stated below. The forward reference here is acceptable, since the proof of the substitution property does not depend on subject reduction.

$\Gamma \vdash [e_2/x]e_1 : \tau$ By the *substitution property* (4)

□

Theorem 4 (Substitution property)
If $\Gamma \vdash e' : \tau'$ *and* $\Gamma, x : \tau' \vdash e : \tau$ *then* $\Gamma \vdash [e'/x]e : \tau$

Proof sketch: By rule induction on the deduction of $\Gamma, x : \tau' \vdash e : \tau$. Intuitively, in this deduction we can use $x : \tau'$ only at the leaves, and there to conclude $x : \tau'$. Now we replace this leaf with the given derivation of $\Gamma \vdash e' : \tau'$ which concludes $e' : \tau'$. Luckily, $[e'/x]x = e'$, so this is the correct judgment.

There is only a small hiccup: when we introduce a different variable $y : \tau''$ into the context in the lam rule, the contexts of the two assumptions no longer match. But we can apply *weakening*, that is, adjoin the unused hypothesis $y : \tau''$ to every judgment in the deduction of $\Gamma \vdash e' : \tau'$. After that, we can apply the induction hypothesis. □

We recommend you write out the cases of the substitution property in the style of our other proofs, just to make sure you understand the details.

The substitution property is so critical that we may elevate it to an intrinsic property of the turnstile ($\vdash$). Whenever we write $\Gamma \vdash J$ for any judgment J we imply that a substitution property for the judgments in Γ must hold. This is an example of a *hypothetical* and *generic* judgment [ML83]. We may return to this point in a future lecture, especially if the property appears to be in jeopardy at some point. It is worth remembering that, while we may not want to prove an explicit substitution property, we still need to make sure that the judgments we define are hypothetical/generic judgments.

3 Taking Stock

Where do we stand at this point in our quest for a representation theorems for Booleans? We have the following:

Reduction and normal forms

For any e, either $e \longrightarrow$ or e *nf* (Theorem L4.1)

Representation of Booleans in normal form

For any e with $\cdot \vdash e : \alpha \to (\alpha \to \alpha)$ and e *nf*, either $e = \mathit{true} = \lambda x.\, \lambda y.\, x$ or $e = \mathit{false} = \lambda x.\, \lambda y.\, y$. (Theorem L3.10(i))

Subject reduction

For any e with $\Gamma \vdash e : \tau$ and $e \longrightarrow e'$ we have $\Gamma \vdash e' : \tau$. (Theorem L4.3)

Subject reduction to normal form

For any e with $\Gamma \vdash e : \tau$ and $e \longrightarrow^* e'$ with e' *nf* we have $\Gamma \vdash e' : \tau$. (Corollary of subject reduction)

Missing at this point are the following theorems

Normalization

If $\Gamma \vdash e : \tau$ then $e \longrightarrow^* e'$ for some e' with e' *nf*.

Confluence

If $e \longrightarrow^* e_1$ and $e \longrightarrow^* e_2$ then there exists an e' such that $e_1 \longrightarrow^* e'$ and $e_2 \longrightarrow e'$.

In this context, *normalization* (sometimes called *termination*) shows that any closed expression of type $\alpha \to (\alpha \to \alpha)$ denotes a Boolean. *Confluence* (also known as the Church-Rosser property) show that this Boolean is unique.

We could replay the whole development for the representation of natural numbers, with some additional complications, but we will forego this in favor of tackling more realistic programming languages.

References

[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 λ-Calculus to Programming Languages

15-814: Types and Programming Languages
Frank Pfenning

Lecture 5
September 18, 2018

1 Introduction

The λ-calculus is exceedingly elegant and minimal, but there are a number of problems if you want to think it of as the basis for an actual programming language. Here are some thoughts discussed in class.

Too abstract. Generally speaking, abstraction is good in the sense that it is an important role of functions (abstracting away from a particular special computation) or modules (abstracting away from a particular implementation). "Too abstract" would mean that we cannot express algorithms or ideas in code because the high level of abstraction prevents us from doing so. This is a legitimate concern for the λ-calculus. For example, what we observe as the result of a computation is only the normal form of an expression, but we might want to express some programs that interact with the world or modify a store. And, yes, the representation of data like natural numbers as functions has problems. While all recursive *functions* on natural numbers can be represented, not all *algorithms* can. For example, under some reasonable assumptions the minimum function on numbers n and k has complexity $O(\max(n, k))$ [CF98], which is surprisingly slow.

Observability of functions. Since reduction results in normal form, to interpret the result of a computation we need to be able to inspect the structure of functions. But generally we like to compile functions and think of them only as something opaque: we can probe it by applying it to arguments, but its structure should be hidden from us. This is a serious and major concern about the pure λ-calculus where all data are expressed as functions.

Generality of typing. The untyped λ-calculus can express fixed points (and therefore all partial recursive functions on its representation of natural numbers) but the same is not true for Church's simply-typed λ-calculus. In fact, the type system so far is very restrictive. Consider the conditional $\mathit{if} = \lambda b.\, b$, where we typed Booleans as $\alpha \to (\alpha \to \alpha)$. We would like to be able to type $\mathit{if}\ b\ e_1\ e_2$ for a Boolean e and expressions e_1 and e_2 of some type τ. Inspection of the typing rules will tell you that $e_1 : \alpha$ and $e_2 : \alpha$, but what if we want to type $\mathit{if}\ b\ \mathit{zero}\ (\mathit{succ}\ \mathit{zero})$ which returns $\overline{0}$ when b is true and $\overline{1}$ if b is false? Recall here that $\overline{n} : \beta \to (\beta \to \beta) \to \beta$ which is different from α. Can we then "instantiate" α with $\beta \to (\beta \to \beta) \to \beta$? It is possible to recover from this mess, but it is not easy.

In this lecture we focus on the first two points: rather than representing all data as functions, we add data to the language directly, with new types and new primitives. At the same time we make the structure of functions *unobservable* so that implementation can compile them to machine code, optimize them, and manipulate them in other ways. Functions become more *extensional* in nature, characterized via their input/output behavior rather than distinguishing functions that have different internal structure.

2 Revising the Dynamics of Functions

The *statics*, that is, the typing rules for functions, do not change, but the way we compute does. We have to change our notion of reduction as well as that of normal forms. Because the difference to the λ-calculus is significant, we call the result of computation *values* and define them with the judgment $e\ \mathit{val}$. Also, we write $e \mapsto e'$ for a single step of computation. For now, we want this step relation to be *deterministic*, that is, we want to arrange the rules so that every expression either steps in a unique way or is a value.

When we are done, we should then check the following properties.

Preservation. If $\cdot \vdash e : \tau$ and $e \mapsto e'$ then $\cdot \vdash e' : \tau$.

Progress. For every expression $\cdot \vdash e : \tau$ either $e \mapsto e'$ or $e\ \mathit{val}$.

Values. If $e\ \mathit{val}$ then there is no e' such that $e \mapsto e'$.

Determinacy. If $e \mapsto e_1$ and $e \mapsto e_2$ then $e_1 = e_2$.

Devising a set of rules is usually the key activity in programming language design. Proving the required theorems is just a way of checking one's work rather than a primary activity. First, one-step computation. We suggest you carefully compare these rules to those in Lecture 4 where reduction could take place in arbitrary position of an expression.

$$\frac{}{\lambda x.\, e\ \mathit{val}}\ \mathsf{val/lam}$$

Note that e here is unconstrained and need not be a value.

$$\frac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2}\ \mathsf{ap}_1 \qquad \frac{}{(\lambda x.\, e_1)\, e_2 \mapsto [e_2/x]e_1}\ \beta$$

These two rules together constitute a strategy called *call-by-name*. There are good practical as well as foundational reasons to use *call-by-value* instead, which we obtain with the following three alternative rules.

$$\frac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2}\ \mathsf{ap}_1 \qquad \frac{v_1\ \mathit{val} \quad e_2 \mapsto e_2'}{v_1\, e_2 \mapsto v_1\, e_2'}\ \mathsf{ap}_2$$

$$\frac{v_2\ \mathit{val}}{(\lambda x.\, e_1)\, v_2 \mapsto [v_2/x]e_1}\ \beta_{\mathit{val}}$$

We achieve determinacy by requiring certain subexpressions to be values. Consequently, computation first reduces the function part of an application, then the argument, and then performs a (restricted form) of β-reduction.

In lecture, we proceeded with the call-by-name rules because there are fewer of them. But there are good logical reasons why functions should be call-by-value, so in these notes we'll use the call-by-value rules instead.

We could now check our desired theorems, but we wait until we have introduced the Booleans as a new primitive type.

3 Booleans as a Primitive Type

Most, if not all, programming languages support Booleans. There are two values, true and false, and usually a conditional expression $\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$. From these we can define other operations such as conjunction or disjunction. Using, as before, α for type variables and x for expression variables, our language then becomes:

$$\begin{array}{llcl} \text{Types} & \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \mathsf{bool} \\ \text{Expressions} & e & ::= & x \mid \lambda x.\, e \mid e_1\, e_2 \\ & & \mid & \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e_1\ e_2\ e_3 \end{array}$$

The additional rules seem straightforward: true and false are values, and a conditional computes by first reducing the condition to true or false and then selecting the correct branch.

$$\frac{}{\mathsf{true}\ \mathit{val}} \qquad \frac{}{\mathsf{false}\ \mathit{val}}$$

$$\frac{e_1 \mapsto e_1'}{\mathsf{if}\ e_1\ e_2\ e_3 \mapsto \mathsf{if}\ e_1'\ e_2\ e_3}\ \mathsf{if}_1$$

$$\frac{}{\mathsf{if}\ \mathsf{true}\ e_2\ e_3 \mapsto e_2}\ \mathsf{if/true} \qquad \frac{}{\mathsf{if}\ \mathsf{false}\ e_2\ e_3 \mapsto e_3}\ \mathsf{if/false}$$

Note that we do not evaluate the branches of a conditional until we know whether the condition is true or false.

How do we type the new expressions? true and false are obvious.

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}$$

The conditional is more interesting. We know its subject e_1 should be of type bool, but what about the branches and the result? We want type preservation to hold and we cannot tell before the program is executed whether the subject of conditional will be true or false. Therefore we postulate that both branches have the same general type τ and that the conditional has the same type.

$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathsf{if}\ e_1\ e_2\ e_3 : \tau}$$

In lecture, a student made the excellent suggestion that we could instead type it as

$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma \vdash e_3 : \tau_3}{\Gamma \vdash \mathsf{if}\ e_1\ e_2\ e_3 : \tau_2 \vee \tau_3}$$

saying that the result must be either of type τ_2 or τ_3. Something like this is indeed possible using so-called *union types*, but it turns out they are quite complex. For example, what can we do safely with the result of the conditional if all we know is that the result is either bool or $\mathsf{bool} \to \mathsf{bool}$? We will make a few more remarks on this in the next lecture.

4 Type Preservation

Now we should revisit the most important theorems about the programming language we define, namely preservation and progress. These two together constitute what we call *type safety*. Since these theorems are of such pervasive importance, we will prove them in great detail. Generally speaking, the proof decomposes along the types present in the language because we carefully designed the rules so that this is the case. For example, we added *if* $e_1\ e_2\ e_3$ as a language primitive instead of as *if* a function of three arguments. Doing the latter would significantly complicate the reasoning.

We already know that the rules should satisfy the substitution property (Theorem L4.4). We can easily check the new cases in the proof because substitution remains compositional. For example, $[e'/x](\mathsf{if}\ \ e_1\ \ e_2\ \ e_3) = \mathsf{if}\ ([e'/x]e_1)\ ([e'/x]e_2)\ ([e'/x]e_3)$.

Property 1 (Substitution)
If $\Gamma \vdash e' : \tau'$ *and* $\Gamma, x : \tau' \vdash e : \tau$ *then* $\Gamma \vdash [e'/x]e : \tau$.

On to preservation.

Theorem 2 (Type Preservation)
If $\cdot \vdash e : \tau$ *and* $e \mapsto e'$ *then* $\cdot \vdash e' : \tau$.

Proof: By rule induction on the deduction of $e \mapsto e'$.

Case:

$$\frac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2}\ \mathsf{ap}_1$$

where $e = e_1\, e_2$ and $e' = e_1'\, e_2$.

$\cdot \vdash e_1\, e_2 : \tau$ Assumption
$\cdot \vdash e_1 : \tau_2 \to \tau$ and $\cdot \vdash e_2 : \tau_2$ for some τ_2 By inversion
$\cdot \vdash e_1' : \tau_2 \to \tau$ By ind.hyp.
$\cdot \vdash e_1'\, e_2 : \tau$ By rule app

Case:

$$\frac{v_1\ \mathit{val} \quad e_2 \mapsto e_2'}{v_1\, e_2 \mapsto v_1\, e_2'}\ \mathsf{ap}_2$$

where $e = v_1\, e_2$ and $e' = v_1\, e_2'$. As in the previous case, we proceed by inversion on typing.

$\cdot \vdash v_1\, e_2 : \tau$ Assumption
$\cdot \vdash v_1 : \tau_2 \to \tau$ and $\cdot \vdash e_2 : \tau_2$ for some τ_2 By inversion
$\cdot \vdash e_2' : \tau_2$ By ind.hyp.
$\cdot \vdash v_1\, e_2' : \tau$ By rule app

Case:

$$\frac{v_2\ \mathit{val}}{(\lambda x.\, e_1)\, v_2 \mapsto [v_2/x]e_1}\ \beta_{\mathit{val}}$$

where $e = (\lambda x.\, e_1)\, v_2$ and $e' = [v_2/x]e_1$. Again, we apply inversion on the typing of e, this time twice. Then we have enough pieces to apply the *substitution property* (Theorem 1).

$\cdot \vdash (\lambda x.\, e_1)\, v_2 : \tau$ Assumption
$\cdot \vdash \lambda x.\, e_1 : \tau_2 \to \tau$ and $\cdot \vdash v_2 : \tau_2$ for some τ_2 By inversion
$x : \tau_2 \vdash e_1 : \tau$ By inversion
$\cdot \vdash [v_2/x]e_1 : \tau$ By the *substitution property* (Theorem 1)

Case:

$$\frac{e_1 \mapsto e_1'}{\mathsf{if}\ e_1\ e_2\ e_3 \mapsto \mathsf{if}\ e_1'\ e_2\ e_3}\ \mathsf{if}_1$$

where $e = \mathsf{if}\ e_1\ e_2\ e_3$ and $e' = \mathsf{if}\ e_1'\ e_2\ e_3$. As might be expected by now, we apply inversion to the typing of e, followed by the induction hypothesis on the type of e_1, followed by re-application of the typing rule for if.

$\cdot \vdash \mathsf{if}\ e_1\ e_2\ e_3 : \tau$ Assumption
$\cdot \vdash e_1 : \mathsf{bool}$ and $\cdot \vdash e_2 : \tau$ and $\cdot \vdash e_3 : \tau$ By inversion
$\cdot \vdash e_1' : \mathsf{bool}$ By ind.hyp.
$\cdot \vdash \mathsf{if}\ e_1'\ e_2\ e_3 : \tau$ By rule

Case:

$$\frac{}{\mathsf{if}\ \mathsf{true}\ e_2\ e_3 \mapsto e_2}\ \mathsf{if/true}$$

where $e = \mathsf{if}\ \mathsf{true}\ e_2\ e_3$ and $e' = e_2$. This time, we don't have an induction hypothesis since this rule has no premise, but fortunately one step of inversion suffices.

$\cdot \vdash \mathsf{if}\ \mathsf{true}\ e_2\ e_3 : \tau$	Assumption
$\cdot \vdash \mathsf{true} : \mathsf{bool}$ and $\cdot \vdash e_2 : \tau$ and $\cdot \vdash e_3 : \tau$	By inversion
$\cdot \vdash e' : \tau$	Since $e' = e_2$.

Case: Rule if/false is analogous to the previous case.

□

5 Progress

To complete the lecture, we would like to prove progress: ever closed, well-typed expression is either already a value or can take a step. First, it is easy to see that the assumptions here are necessary. For example, the ill-typed expression if $(\lambda x.\, x)$ false true cannot take a step since the subject $(\lambda x.\, x)$ is a value but the whole expression is not and cannot take a step. Similarly, the expression if b false true is well-typed in the context with b : bool, but it cannot take a step nor is it a value.

Theorem 3 (Progress)
If $\cdot \vdash e : \tau$ then either $e \mapsto e'$ for some e' or e val.

Proof: There are not many candidates for this proof. We have e and we have a typing for e. From that scant information we need obtain evidence that e can step or is a value. So we try the rule induction on $\cdot \vdash e : \tau$.

Case:

$$\frac{x_1 : \tau_1 \vdash e_2 : \tau_2}{\cdot \vdash \lambda x_1.\, e_2 : \tau_1 \to \tau_2}$$

where $e = \lambda x_1.\, e_2$. Then we have

$\lambda x_1.\, e_2$ *val*	By rule val/lam

It is fortunate we don't need the induction hypothesis, because it cannot be applied! That's because the context of the premise is not empty, which is easy to miss. So be careful!

Case:

$$\frac{\cdot \vdash e_1 : \tau_2 \to \tau \quad \cdot \vdash e_2 : \tau_2}{\cdot \vdash e_1\, e_2 : \tau}$$

where $e = e_1\, e_2$. At this point we apply the induction hypothesis to e_1. If it reduces, so does $e = e_1\, e_2$. If it is a value, then we apply the induction hypothesis to e_2. If is reduces, so does $e_1\, e_2$. If not, we have a β_{val} redex. In more detail:

Either $e_1 \mapsto e_1'$ for some e_1' or e_1 *val* — By ind.hyp.

$e_1 \mapsto e_1'$ — Subcase
$e = e_1\, e_2 \mapsto e_1'\, e_2$ by rule ap_1

e_1 *val* — Subcase
Either $e_2 \mapsto e_2'$ for some e_2' or e_2 *val* — By ind.hyp.

$e_2 \mapsto e_2'$ — Sub^{2}case
$e_1\, e_2 \mapsto e_1\, e_2'$ — By rule ap_2 since e_1 *val*

e_2 *val* — Sub^{2}case
$e_1 = \lambda x.\, e_1'$ and $x : \tau_2 \vdash e_1' : \tau$ — By "inversion"

We pause here to consider this last step. We know that $\cdot \vdash e_1 : \tau_2 \to \tau$ and e_1 *val*. By considering all cases for how both of these judgments can be true at the same time, we see that e_1 must be a λ-abstraction. This is often summarized in a *canonical forms lemma* which we didn't discuss in lecture, but state after this proof. Finishing this sub^{2}case:

$e = (\lambda x\, e_1')\, e_2 \mapsto [e_2/x]e_1'$ — By rule β_{val} since e_2 *val*

Case:

$$\frac{}{\cdot \vdash \mathsf{true} : \mathsf{bool}}$$

where $e = \mathsf{true}$. Then $e = \mathsf{true}$ *val* by rule.

Case: Typing of false. As for true.

Case:

$$\frac{\cdot \vdash e_1 : \mathsf{bool} \quad \cdot \vdash e_2 : \tau \quad \cdot \vdash e_3 : \tau}{\cdot \vdash \mathsf{if}\ e_1\ e_2\ e_3 : \tau}$$

where $e = \mathit{if}\ e_1\ e_2\ e_3$.

Either $e_1 \mapsto e_1'$ for some e_1' or e_1 *val* — By ind.hyp.

$e_1 \mapsto e_1'$ — Subcase
$e = \mathsf{if}\ e_1\ e_2\ e_3 \mapsto \mathsf{if}\ e_1'\ e_2\ e_3$ — By rule if_1

e_1 *val* — Subcase
$e_1 = \mathsf{true}$ or $e_1 = \mathsf{false}$
By considering all cases for $\cdot \vdash e_1 : \mathsf{bool}$ and e_1 *val*

$e_1 = \mathsf{true}$ — Sub^{2}case
$e = \mathsf{if}\ \mathsf{true}\ e_2\ e_3 \mapsto e_2$ — By rule

$e_1 = \mathsf{false}$ — Sub^{2}case
$e = \mathsf{if}\ \mathsf{false}\ e_2\ e_3 \mapsto e_3$ — By rule

□

This completes the proof. The complex inversion steps can be summarized in the *canonical forms lemma* that analyzes the shape of well-typed values. It is a form of the representation theorem for Booleans we proved in an earlier lecture for the simply-typed λ-calculus.

Lemma 4 (Canonical Forms)

(i) If $\cdot \vdash v : \tau_1 \to \tau_2$ and v val then $v = \lambda x_1.\, e_2$ for some x_1 and e_2.

(ii) If $\cdot \vdash v : \mathsf{bool}$ and v val then $v = \mathsf{true}$ or $v = \mathsf{false}$.

Proof: For each part, analyzing all the possible cases for the value and typing judgments. □

References

[CF98] Loïc Colson and Daniel Fredholm. System T, call-by-value, and the minimum problem. *Theoretical Computer Science*, 206(1–2):301–315, 1998.

Lecture Notes on Sum Types

15-814: Types and Programming Languages
Frank Pfenning

Lecture 6
September 21, 2018

1 Introduction

So far in this course we have introduced only basic constructs that exist in pretty much any programming language: functions and Booleans. There may be details of syntax and maybe some small semantics differences such as call-by-value vs. call-by-name, but any such differences can be easily explained and debated within the framework set out so far.

At this point we have a choice between several different directions in which we can extend our inquiry into the nature of programming language.

Precision of Types. We can make types more or less precise in what they say about the program. For example, we might have type containing just true and another containing just false. At the end of this spectrum would be *dependent types* so precise that they can completely specify a function.

Expressiveness of Types. We can analyze which programs can not be typed and make the type system accept more programs, as long as it remains sound.

Computational Mechanisms. So far computation in our language is *value-oriented* in that evaluating an expression returns a value, but it cannot have any effect such as mutating a store, performing input or output, raising an exception, or execute concurrently.

Level of Dynamics. The rules for computation are at a very high level of abstraction and do not talk about, for example, where data might be allocated in memory, or how functions are compiled. A language admits a range of different operational specifications at different levels of abstraction.

Equality and Reasoning. We have introduced typing rules, but no informal or formal system for reasoning about programs. This might include various definitions when we might consider programs to be equal, and rules for establishing equality. Or it might include a language for specifying programs and rules for establishing that they satisfy their specifications. Under this general heading we might also consider translations between different languages and showing their correctness.

All of these are interesting and the subject of ongoing research in programming languages. At the moment, we do not yet have enough infrastructure to make most of these questions rich and interesting. So in the next few lectures we will introduce additional types and corresponding expressions to make the language expressive enough to recover partial recursive functions over interesting forms of data such as natural numbers, lists, trees, etc.

2 Disjoint Sums

Type theory is an open-ended enterprise: we are always looking to capture types of data, modes of computation, properties of programs, etc. One important building block are *type constructors* that build more complicated types out of simpler ones. The function type constructor $\tau_1 \to \tau_2$ is one example. Today we see another one: disjoint sums $\tau_1 + \tau_2$. A value of this type either a value of type τ_1 or a value of type τ_2 *tagged with the information about which side of the sum it is.* This last part is critical and distinguishes it from the *union type* which is not tagged and much more difficult to integrate soundly into a programming language. We use l and r as *tags* or *labels* and write $l \cdot e_1$ for the expression of type $\tau_1 + \tau_2$ if $e_1 : \tau_1$ and, analogously, $r \cdot e_2$ if $e_2 : \tau_2$.

$$\frac{\Gamma \vdash e_1 : \tau_1}{\Gamma \vdash l \cdot e_1 : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e_2 : \tau_2}{\Gamma \vdash r \cdot e_2 : \tau_1 + \tau_2}$$

These two forms of expressions allow us to form element of the disjoint sum. To destruct such a sum we need a case construct that discriminates based on whether element of the sum is injected on the left or on the right.

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathsf{case}\ e\ \{l \cdot x_1 \Rightarrow e_1 \mid r \cdot x_2 \Rightarrow e_2\} : \tau}$$

Let's talk through this rule. The subject of the case should have type $\tau_1 + \tau_2$ since this is what we are discriminating. If the value of this type is $l \cdot v_1$ then by the typing rule for the left injection, v_1 must have type τ_1. Since the variable x_1 stands for v_1 is should have type τ_1 in the first branch. Similarly, x_2 should have type τ_2 in the seond branch. Since we cannot tell until the program executes which branch will be taken, just like the conditional in the last lecture, we require that both branches have the same type τ, which is also the type of the whole case.

From this, we can also deduce the value and stepping judgments for the new constructs.

$$\frac{e\ \mathit{val}}{l \cdot e\ \mathit{val}}\ \mathsf{val}/l \qquad \frac{e\ \mathit{val}}{r \cdot e\ \mathit{val}}\ \mathsf{val}/r$$

$$\frac{e \mapsto e'}{l \cdot e \mapsto l \cdot e'}\ {\mapsto}/l \qquad \frac{e \mapsto e'}{r \cdot e \mapsto r \cdot e'}\ {\mapsto}/r$$

$$\frac{e \mapsto e'}{\mathsf{case}\ e\ \{\ldots\} \mapsto \mathsf{case}\ e'\ \{\ldots\}}\ {\mapsto}/\mathsf{case}_1$$

$$\frac{v_1\ \mathit{val}}{\mathsf{case}\ (l \cdot v_1)\ \{l \cdot x_1 \Rightarrow e_1 \mid \ldots\} \mapsto [v_1/x_1]e_1}\ {\mapsto}/\mathsf{case}/l$$

$$\frac{v_2\ \mathit{val}}{\mathsf{case}\ (r \cdot v_2)\ \{\ldots \mid r \cdot x_2 \Rightarrow e_2\} \mapsto [v_2/x_2]e_2}\ {\mapsto}/\mathsf{case}/r$$

We have carefully constructed our rules so that the new cases in the preservation and progress theorems should be straightforward.

Theorem 1 (Preservation)
If $\cdot \vdash e : \tau$ *and* $e \mapsto e'$ *then* $\cdot \vdash e' : \tau$

Proof: Before we dive into the new case, a remark on the rule. You can see that the type of an expression $l \cdot e_1$ is inherently ambiguous, even if we know that $e_1 : \tau_1$. In fact, it will have the type $\tau_1 + \tau_2$ for *every* type τ_2. This is acceptable because we either use bidirectional type checking, in which case both $\tau_1 + \tau_2$ and $l \cdot e_1$ are given to use, or we use some form of type inference that will determine the most general type for an expression.

In any case, these considerations do not affect type preservation. There, we just need to show that *any* type τ that e possesses will also be a type of e' if $e \mapsto e'$. Now, it is completely possible that e' will have *more* types than e, but that doesn't contradict the theorem.[1]

The proof of preservation proceeds as usual, by rule on induction on the step $e \mapsto e'$, applying inversion of the typing of e. We show only the new cases, because the cases for all other constructs remain exactly as before. We assume that the substitution property carries over.

Case:

$$\frac{e_1 \mapsto e_1'}{l \cdot e_1 \mapsto l \cdot e_1'} \mapsto/l$$

where $e = l \cdot e_1$ and $e' = l \cdot e_1'$

$\cdot \vdash l \cdot e_1 : \tau_1 + \tau_2$ Assumption
$\cdot \vdash e_1 : \tau_1$ By inversion
$\cdot \vdash e_1' : \tau_1$ By ind.hyp.
$\cdot \vdash l \cdot e_1' : \tau_1 + \tau_2$ By rule

Case: Rule $\mapsto/r$: analogous to $\mapsto/l$.

Case: Rule $\mapsto/\mathsf{case}_1$: similar to the previous two cases.

Case:

$$\frac{v_1\ \mathit{val}}{\mathsf{case}\ (l \cdot v_1)\ \{l \cdot x_1 \Rightarrow e_1 \mid \ldots\} \mapsto [v_1/x_1]e_1} \mapsto/\mathsf{case}/l$$

where $e = \mathsf{case}\ (l \cdot v_1)\ \{l \cdot x_1 \Rightarrow e_1 \mid \ldots\}$ and $e' = [v_1/x_1]e_1$.

$\cdot \vdash \mathsf{case}\ (l \cdot v_1)\ \{l \cdot x_1 \Rightarrow e_1 \mid r \cdot x_2 \Rightarrow e_2\} : \tau$ Assumption
$\cdot \vdash l \cdot v_1 : \tau_1 + \tau_2$
and $x_1 : \tau_1 \vdash e_1 : \tau$ and $x_2 : \tau_2 \vdash e_2 : \tau$ for some τ_1 and τ_2 By inversion
$\cdot \vdash v_1 : \tau_1$ By inversion
$[v_1/x_1]e_1 : \tau$ By the substitution property

[1] It is an instructive exercise to construct a well-typed closed term e with $e \mapsto e'$ such that e' has more types than e.

Case: Rule $\mapsto/\mathsf{case}/r$: analogous to $\mapsto/r$.

□

The progress theorem proceeds by induction on the typing derivation, as usual, analyzing the possible cases. Before we do that, it is always helpful to call out the canonical forms theorem that characterizew well-typed values. New here is part (iii).

Lemma 2 (Canonical Forms)

(i) If $\cdot \vdash v : \tau_1 \to \tau_2$ *and* v *val then* $v = \lambda x_1.\, e_2$ *for some* x_1 *and* e_2*.*

(ii) If $\cdot \vdash v : \mathsf{bool}$ *and* v *val then* $v = \mathsf{true}$ *or* $v = \mathsf{false}$*.*

(iii) If $\cdot \vdash v : \tau_1 + \tau_2$ *and* v *val then* $v = l \cdot v_1$ *for some* v_1 *val or* $v = r \cdot v_2$ *for some* v_2 *val.*

Proof sketch: For each part, analyzing all the possible cases for the value and typing judgments. □

Theorem 3 (Progress)
If $\cdot \vdash e : \tau$ *then either* $e \mapsto e'$ *for some* e' *or* e *val.*

Proof: By rule induction on the given typing derivation.

Cases: For constructs pertaining to types $\tau_1 \to \tau_2$ or bool, just as before since we did not change their rules.

Case:

$$\frac{\cdot \vdash e_1 : \tau_1}{\cdot \vdash l \cdot e_1 : \tau_1 + \tau_2}$$

where $e = l \cdot e_1$.

Either $e_1 \mapsto e_1'$ for some e_1' or e_1 *val* By ind.hyp.

$e_1 \mapsto e_1'$ Subcase
$l \cdot e_1 \mapsto l \cdot e_1'$ By rule $\mapsto/l$

e_1 *val* Subcase
$l \cdot e_1$ *val* By rule val/l

Case: Typing of $r \cdot e_2$: analogous to previous case.

Case:

$$\frac{\cdot \vdash e_0 : \tau_1 + \tau_2 \quad \cdot, x_1 : \tau_1 \vdash e_1 : \tau \quad \cdot, x_2 : \tau_2 \vdash e_2 : \tau}{\cdot \vdash \mathsf{case}\ e_0\ \{l \cdot x_1 \Rightarrow e_1 \mid r \cdot x_2 \Rightarrow e_2\} : \tau}$$

where $e = \mathsf{case}\ e_0\ \{l \cdot x_1 \Rightarrow e_1 \mid r \cdot x_2 \Rightarrow e_2\}$.

Either $e_0 \mapsto e_0'$ for some e_0' or e_0 *val* — By ind.hyp.

$e_0 \mapsto e_0'$ — Subcase
$e = \mathsf{case}\ e_0\ \{l \cdot x_1 \Rightarrow e_1 \mid r \cdot x_2 \Rightarrow e_2\}$
$\quad \mapsto \mathsf{case}\ e_0'\ \{l \cdot x_1 \Rightarrow e_1 \mid r \cdot x_2 \Rightarrow e_2\}$ — By rule $\mapsto/\mathsf{case}_1$

e_0 *val* — Subcase
$e_0 = l \cdot e_0'$ for some e_0' *val*
or $e_0 = r \cdot e_0'$ for some e_0' *val* — By the canonical forms property (4)

$e_0 = l \cdot e_0'$ and e_0' *val* — Sub^{2}case
$e = \mathsf{case}\ (l \cdot e_0')\ \{l \cdot x_1 \Rightarrow e_1 \mid \ldots\} \mapsto [e_0'/x_1]e_1$ — By rule $\mapsto/\mathsf{case}/l$

$e_0 = r \cdot e_0'$ and e_0' *val* — Sub^{2}case
$e = \mathsf{case}\ (r \cdot e_0')\ \{\ldots \mid r \cdot x_2 \Rightarrow e_2\} \mapsto [e_0'/x_2]e_2$ — By rule $\mapsto/\mathsf{case}/r$

□

3 The Unit Type 1

In order to use sums, it is helpful to have a unit type, written 1, that has exactly one element $\langle\,\rangle$. If we had such a type, we could define *bool* $= 1 + 1$ and bool would no longer be primitive. $1 + 1$ contains exactly two values, namely $l \cdot \langle\,\rangle$ and $r \cdot \langle\,\rangle$.

We have one form "constructing" the unit value and a corresponding case-like elimination, except that there is only on branch.

$$\frac{}{\Gamma \vdash \langle\,\rangle : 1} \qquad \frac{\Gamma \vdash e_0 : 1 \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash \mathsf{case}\ e_0\ \{\langle\,\rangle \Rightarrow e_1\} : \tau}$$

There is not much going on as far as the operational semantics is concerned.

$$\frac{}{\langle\,\rangle\ \mathsf{val}}$$

$$\frac{e_1 \mapsto e_1'}{\mathsf{case}\ e_1\ \{\langle\,\rangle \Rightarrow e_1\} \mapsto \mathsf{case}\ e_1'\ \{\langle\,\rangle \Rightarrow e_1\}} \qquad \frac{}{\mathsf{case}\ \langle\,\rangle\ \{\langle\,\rangle \Rightarrow e_1\} \mapsto e_1}$$

Preservation and progress continue to hold, and are proved following the pattern of the previous cases. We just restate the canonical forms lemma.

Lemma 4 (Canonical Forms)

(i) If $\cdot \vdash v : \tau_1 \to \tau_2$ and v val then $v = \lambda x_1.\, e_2$ for some x_1 and e_2.

(ii) If $\cdot \vdash v :$ bool and v val then $v =$ true or $v =$ false.

(iii) If $\cdot \vdash v : \tau_1 + \tau_2$ and v val then $v = l \cdot v_1$ for some v_1 val or $v = r \cdot v_2$ for some v_2 val.

(iv) If $\cdot \vdash v : 1$ and v val then $v = \langle\,\rangle$.

4 Using the Unit Type

As indicated in the previous section, we can now *define* the Boolean type using sums and unit. We have:

$$\begin{array}{lcl} \mathit{bool} & = & 1 + 1 \\ \mathit{true} & = & l \cdot \langle\,\rangle \\ \mathit{false} & = & r \cdot \langle\,\rangle \\ \mathit{if}\ e_0\ e_1\ e_2 & = & \mathsf{case}\ e_0\ (l \cdot x_1 \Rightarrow e_1 \mid r \cdot x_2 \Rightarrow e_2) \\ & & (\text{provided } x_1 \notin \mathsf{fv}(e_1) \text{ and } x_2 \notin \mathsf{fv}(e_2)) \end{array}$$

The provisos on the last definition are important because we don't want to accidentally capture a free variable in e_1 or e_2 during the translation. Recommended question to think about: could we define a function if_τ : $(1 + 1) \to \tau \to \tau \to \tau$ for arbitrary τ that implements the case construct?

Using 1 we can define other types. For example

$$\tau\ \mathit{option} \ = \ \tau + 1$$

represents an optional value of type τ. Its values are $l \cdot v$ for $v : \tau$ (we have a value) or $r \cdot \langle\,\rangle$, where $r \cdot \langle\,\rangle$ (we have no value).

A more interesting examples would be the natural numbers:

$$\begin{array}{lcl}
\mathit{nat} & = & 1 + (1 + (1 + \cdots)) \\
\overline{0} & = & l \cdot \langle\,\rangle \\
\overline{1} & = & r \cdot (l \cdot \langle\,\rangle) \\
\overline{2} & = & r \cdot (r \cdot (l \cdot \langle\,\rangle)) \\
\mathit{succ} & = & \lambda n.\, r \cdot n
\end{array}$$

Unfortunately, "$\cdots$" is not really permitted in the definition of types. We could define it *recursively* as

$$\mathit{nat} \quad = \quad 1 + \mathit{nat}$$

but supporting this style of recursive type definition is not straightforward. We will probably use explicit *recursive types* to define

$$\mathit{nat} \quad = \quad \rho\alpha.\, 1 + \alpha$$

So natural numbers, if we want to build them up from simpler components rather than as a primitive, require a unit type, sums, and recursive types.

5 The Empty Type 0

We have the singleton type 1, a type with two elements, $1 + 1$, so can we also have a type with no elements? Yes! We'll call it 0 because it will satisfy (in a way we do not make precise) that $0 + \tau \simeq \tau$. There are no constructors and no values of this type, so the e *val* judgment is not extended.

If we think ot 0 as a nullary sum, we expect there still to be a destructor. But instead of two branches it has zero branches!

$$\frac{\Gamma \vdash e_0 : 0}{\Gamma \vdash \mathsf{case}\ e_0\ \{\,\} : \tau}$$

Computation also makes some sense with a congruence rule reducing the subject, but the case can never be reduced.

$$\frac{e_0 \mapsto e_0'}{\mathsf{case}\ e_0\ \{\,\} \mapsto \mathsf{case}\ e_0'\ \{\,\}}$$

Progress and preservation extend somewhat easily, and the canonical forms property is extended with *(v) If $\cdot \vdash v : 0$ and v val then we have a contradiction.*

The empty type has somewhat limited uses precisely because there is no value of this type. However, there may still be expression e such that $\cdot \vdash e : 0$ if we have explicitly nonterminating expressions. Such terms can appear the subject of a case where they reduce forever by the only rule. We can also ask, for example, what would be functions from $0 \to 0$. We find:

$$\begin{array}{lcl} \lambda x.\, x & : & 0 \to 0 \\ \lambda x.\, \mathsf{case}\ x\ \{\,\} & : & 0 \to 0 \\ \lambda x.\, \mathit{loop} & : & 0 \to 0 \end{array}$$

assume we can define a looping term and give it type 0.

6 Summary

We present a brief summary of the language of types and expressions we have defined so far.

$$\begin{array}{llcl} \text{Types} & \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid 0 \mid 1 \\ \text{Expressions} & e & ::= & x \mid \lambda x.\, e \mid e_1\, e_2 \\ & & \mid & l \cdot e \mid r \cdot e \mid \mathsf{case}\ e_0\ \{l \cdot x_1 \Rightarrow e_1 \mid r \cdot x_2 \Rightarrow e_2\} \\ & & \mid & \mathsf{case}\ e_0\ \{\,\} \\ & & \mid & \langle\,\rangle \mid \mathsf{case}\ e_0\ \{\langle\,\rangle \Rightarrow e_1\} \end{array}$$

Functions.

$$\frac{\Gamma, x_1 : \tau_2 \vdash e_2 : \tau_2}{\Gamma \vdash \lambda x_1.\, e_2 : \tau_1 \to \tau_2} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_1}$$

$$\frac{}{\lambda x.\, e\ \mathit{val}}$$

$$\frac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2} \qquad \frac{v_1\ \mathit{val} \quad e_2 \mapsto e_2'}{v_1\, e_2 \mapsto v_1\, e_2'}$$

$$\frac{v_2\ \mathit{val}}{(\lambda x.\, e_1)\, v_2 \mapsto [v_2/x]e_1}$$

Disjoint Sums.

$$\frac{\Gamma \vdash e_1 : \tau_1}{\Gamma \vdash l \cdot e_1 : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e_2 : \tau_2}{\Gamma \vdash r \cdot e_2 : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathsf{case}\ e\ \{l \cdot x_1 \Rightarrow e_1 \mid r \cdot x_2 \Rightarrow e_2\} : \tau}$$

$$\frac{e\ \textit{val}}{l \cdot e\ \textit{val}} \qquad \frac{e\ \textit{val}}{r \cdot e\ \textit{val}}$$

$$\frac{e \mapsto e'}{l \cdot e \mapsto l \cdot e'} \qquad \frac{e \mapsto e'}{r \cdot e \mapsto r \cdot e'}$$

$$\frac{e \mapsto e'}{\mathsf{case}\ e\ \{\ldots\} \mapsto \mathsf{case}\ e'\ \{\ldots\}}$$

$$\frac{v_1\ \textit{val}}{\mathsf{case}\ (l \cdot v_1)\ \{l \cdot x_1 \Rightarrow e_1 \mid \ldots\} \mapsto [v_1/x_1]e_1}$$

$$\frac{v_2\ \textit{val}}{\mathsf{case}\ (r \cdot v_2)\ \{\ldots \mid r \cdot x_2 \Rightarrow e_2\} \mapsto [v_2/x_2]e_2}$$

Unit Type.

$$\frac{}{\Gamma \vdash \langle\,\rangle : 1} \qquad \frac{\Gamma \vdash e_0 : 1 \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash \mathsf{case}\ e_0\ \{\langle\,\rangle \Rightarrow e_1\} : \tau}$$

$$\frac{}{\langle\,\rangle\ \mathsf{val}}$$

$$\frac{e_1 \mapsto e_1'}{\mathsf{case}\ e_1\ \{\langle\,\rangle \Rightarrow e_1\} \mapsto \mathsf{case}\ e_1'\ \{\langle\,\rangle \Rightarrow e_1\}} \qquad \frac{}{\mathsf{case}\ \langle\,\rangle\ \{\langle\,\rangle \Rightarrow e_1\} \mapsto e_1}$$

Empty Type.

$$\frac{\Gamma \vdash e_0 : 0}{\Gamma \vdash \mathsf{case}\ e_0\ \{\,\} : \tau}$$

$$\frac{e_0 \mapsto e_0'}{\mathsf{case}\ e_0\ \{\,\} \mapsto \mathsf{case}\ e_0'\ \{\,\}}$$

Lecture Notes on
Eager Products

15-814: Types and Programming Languages
Ryan Kavanagh

Lecture 7
Tuesday, September 25, 2018

1 Introduction

Last time, we added sums to our language. This allowed us to deal with collections of individual "tagged" values. Sometimes we would like to simultaneously consider multiple values. To do this, we introduce *eager products*. These are akin to "pairs" or "tuples" of values.

2 Syntax

We need to extend the syntax for our types and our terms to handle the new constructs:

$$
\begin{aligned}
\tau &::= \cdots \\
&\quad \mid \tau_1 \otimes \tau_2 && \text{eager product of } \tau_1 \text{ and } \tau_2 \\
&\quad \mid \mathbf{1} && \text{nullary product} \\
e &::= \cdots \\
&\quad \mid \langle e_1, e_2 \rangle && \text{ordered pair of } e_1 \text{ and } e_2 \\
&\quad \mid \langle\,\rangle && \text{null tuple} \\
&\quad \mid \textbf{case } e\ \{\langle x_1, x_2\rangle \Rightarrow e'\} && \text{eager pair destructor} \\
&\quad \mid \textbf{case } e\ \{\langle\,\rangle \Rightarrow e'\} && \text{null tuple destructor}
\end{aligned}
$$

3 Statics

The product type has the following introduction rules:

$$\frac{}{\Gamma \vdash \langle\,\rangle : \mathbf{1}}\ (\text{I-}\mathbf{1}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 \otimes \tau_2}\ (\text{I-}\otimes)$$

Its elimination rules are:

$$\frac{\Gamma \vdash e_0 : \mathbf{1} \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash \textbf{case}\ e_0\ \{\langle\,\rangle \Rightarrow e_1\} : \tau}\ (\text{E-}\mathbf{1})$$

$$\frac{\Gamma \vdash e_0 : \tau_1 \otimes \tau_2 \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e_1 : \tau}{\Gamma \vdash \textbf{case}\ e_0\ \{\langle x_1, x_2\rangle \Rightarrow e_1\} : \tau}\ (\text{E-}\otimes)$$

4 Dynamics

The intended semantics is that we always eagerly evaluate pairs and only eliminate a **case** when both of the paired expressions are values. First, a tuple is a value only when all of its components are values:

$$\frac{}{\langle\,\rangle\ \mathit{val}}\ (\langle\,\rangle\text{-}\textsc{val}) \qquad \frac{v_1\ \mathit{val} \quad v_2\ \mathit{val}}{\langle v_1, v_2\rangle\ \mathit{val}}\ (\textsc{Pair-val})$$

Otherwise, we reduce the components to values from left to right:

$$\frac{e_1 \mapsto e_1'}{\langle e_1, e_2\rangle \mapsto \langle e_1', e_2\rangle}\ (\textsc{Step-L}) \qquad \frac{v_1\ \mathit{val} \quad e_2 \mapsto e_2'}{\langle v_1, e_2\rangle \mapsto \langle v_1, e_2'\rangle}\ (\textsc{Step-R})$$

In the elimination forms, we step the subjects until they become values:

$$\frac{e_0 \mapsto e_0'}{\textbf{case}\ e_0\ \{\langle\,\rangle \Rightarrow e_1\} \mapsto \textbf{case}\ e_0'\ \{\langle\,\rangle \Rightarrow e_1\}}\ (\textsc{Step-Subj-1})$$

$$\frac{e_0 \mapsto e_0'}{\textbf{case}\ e_0\ \{\langle x_1, x_2\rangle \Rightarrow e_1\} \mapsto \textbf{case}\ e_0'\ \{\langle x_1, x_2\rangle \Rightarrow e_1\}}\ (\textsc{Step-Subj-2})$$

Then we simultaneously substitute where applicable:

$$\frac{}{\textbf{case}\ \langle\,\rangle\ \{\langle\,\rangle \Rightarrow e_1\} \mapsto e_1}\ (\textsc{Step-Case-1})$$

$$\frac{\langle v_1, v_2\rangle\ \mathit{val}}{\textbf{case}\ \langle v_1, v_2\rangle\ \{\langle x_1, x_2\rangle \Rightarrow e_1\} \mapsto [v_1, v_2/x_1, x_2]e_1}\ (\textsc{Step-Case-2})$$

5 Desiderata

Our definition satisfies all of our desiderata. The proofs are left as exercises.

Theorem 1 (Type Safety) *Our rules satisfy the progress property, that is, for all e, τ_1, and τ_2,*

1. *if $\cdot \vdash e : \mathbf{1}$, then either e val or there exists an e' such that $e \mapsto e'$, and*
2. *if $\cdot \vdash e : \tau_1 \otimes \tau_2$, then either e val or there exists an e' such that $e \mapsto e'$.*

They also satisfy the preservation property, that is, for all e, e', τ_1, and τ_2,

1. *if $\cdot \vdash e : \mathbf{1}$ and $e \mapsto e'$, then $\cdot \vdash e' : \mathbf{1}$, and*
2. *if $\cdot \vdash e : \tau_1 \otimes \tau_2$ and $e \mapsto e'$, then $\cdot \vdash e' : \tau_1 \otimes \tau_2$.*

Proof: The proof of progress is by induction on the derivation of $\cdot \vdash e : \tau$. The proof of preservation is by induction on the derivation of $e \mapsto e'$. □

Theorem 2 (Canonical Forms for Eager Products) *Values have the following characterization:*

1. *If $\cdot \vdash e : \mathbf{1}$ and e val, then $e \equiv \langle\,\rangle$.*
2. *If $\cdot \vdash e : \tau_1 \otimes \tau_2$ and e val, then $e \equiv \langle v_1, v_2 \rangle$, where $\cdot \vdash v_i : \tau_i$ and v_i val.*

6 Programming with pairs

To better grasp how these pairs work, let us do a bit of programming. We say that types τ and τ' are **isomorphic**, $\tau \cong \tau'$, if there exist terms $f : \tau \to \tau'$ and $g : \tau' \to \tau$ such that are mutual inverses. The exact meaning of "mutual inverses" is subtle because it requires us to specify what we mean by equality when we say $f(g(x)) = x$ and $g(f(x)) = x$. For our call-by-value language, it will be sufficient to require for certain x that $f(g(x)) \mapsto^* x$ and $g(f(x)) \mapsto^* x$. Explicitly, types τ and τ' are isomorphic if there exist f and g satisfying:

- $\cdot \vdash f : \tau \to \tau'$,
- $\cdot \vdash g : \tau' \to \tau$,
- for all v such that $\cdot \vdash v : \tau$ and v *val*, $g(f(v)) \mapsto^* v$, and
- for all v such that $\cdot \vdash v : \tau'$ and v *val*, $f(g(v)) \mapsto^* v$.

In this case, we say that f and g are **witnesses** to the isomorphism.

6.1 Unit is a unit

Our first observation is that $\mathbf{1}$ is the unit for $\otimes$, i.e., that for all τ,

$$\tau \otimes \mathbf{1} \cong \tau.$$

This isomorphism is witnessed by the following pair of mutual inverses:

$$\begin{aligned} \rho &= \lambda x.\textbf{case}\ x\ \{\langle l, \_\rangle \Rightarrow l\}, \\ \rho^{-1} &= \lambda x.\langle x, \langle\,\rangle\rangle. \end{aligned}$$

We begin by showing that they have the right types. First, we show that $\cdot \vdash \rho : \tau \otimes \mathbf{1} \to \tau$:

$$\dfrac{\dfrac{\dfrac{}{x : \tau \otimes \mathbf{1} \vdash x : \tau \otimes \mathbf{1}}\ (\textsc{Var}) \qquad \dfrac{}{x : \tau \otimes \mathbf{1}, l : \tau, \_ : \mathbf{1} \vdash l : \tau}\ (\textsc{Var})}{x : \tau \otimes \mathbf{1} \vdash \textbf{case}\ x\ \{\langle l, \_\rangle \Rightarrow l\} : \tau}\ (\text{E-}\otimes)}{\cdot \vdash \lambda x.\textbf{case}\ x\ \{\langle l, \_\rangle \Rightarrow l\} : \tau \otimes \mathbf{1} \to \tau}\ (\textsc{Lam})$$

Next, we show $\cdot \vdash \rho^{-1} : \tau \to \tau \otimes \mathbf{1}$:

$$\dfrac{\dfrac{\dfrac{}{x : \tau \vdash x : \tau}\ (\textsc{Var}) \qquad \dfrac{}{x : \tau \vdash \langle\,\rangle : \mathbf{1}}\ (\text{I-}\mathbf{1})}{x : \tau \vdash \langle x, \langle\,\rangle\rangle \tau \otimes \mathbf{1}}\ (\text{I-}\otimes)}{\cdot \vdash \lambda x.\langle x, \langle\,\rangle\rangle : \tau \to \tau \otimes \mathbf{1}}\ (\textsc{Lam})$$

We must also show that these two functions are mutual inverses. This requires us to show that for all values v such that $\vdash v : \tau$, we have the following reduction, where we colour-code the redexes in red:

$$\begin{aligned} \rho(\rho^{-1}(v)) &\equiv (\lambda x.\textbf{case}\ x\ \{\langle l, \_\rangle \Rightarrow l\})((\lambda x.\langle x, \langle\,\rangle\rangle)v) \\ &\mapsto (\lambda x.\textbf{case}\ x\ \{\langle l, \_\rangle \Rightarrow l\})\langle v, \langle\,\rangle\rangle \\ &\mapsto \textbf{case}\ \langle v, \langle\,\rangle\rangle\ \{\langle l, \_\rangle \Rightarrow l\} \\ &\mapsto [v, \langle\,\rangle / l, \_]l \\ &\equiv v. \end{aligned}$$

We must also show that for all values v such that $\vdash v : \tau \otimes \mathbf{1}$, we have by the canonical forms theorem that $v \equiv \langle t, \langle\,\rangle\rangle$ for some value t satisfying $\vdash t : \tau$.

We then have the following reduction:

$$\begin{aligned}
\rho^{-1}(\rho(v)) &\equiv (\lambda x.\langle x, \langle\,\rangle\rangle)((\lambda x.\textbf{case}\ x\ \{\langle l, \_\rangle \Rightarrow l\})v) \\
&\mapsto (\lambda x.\langle x, \langle\,\rangle\rangle)(\textbf{case}\ v\ \{\langle l, \_\rangle \Rightarrow l\}) \\
&\equiv (\lambda x.\langle x, \langle\,\rangle\rangle)(\textbf{case}\ \langle t, \langle\,\rangle\rangle\ \{\langle l, \_\rangle \Rightarrow l\}) \\
&\mapsto (\lambda x.\langle x, \langle\,\rangle\rangle)([t, \langle\,\rangle / l, \_]l) \\
&\equiv (\lambda x.\langle x, \langle\,\rangle\rangle)t \\
&\mapsto \langle t, \langle\,\rangle\rangle \\
&\equiv v.
\end{aligned}$$

6.2 One is not two

In general, it is not the case for arbitrary τ that:

$$\tau \cong \tau + \tau.$$

To see why, it is sufficient to take $\tau = \mathbf{1}$ and observe that $\mathbf{1}$ has one value, $\langle\,\rangle$, while $\mathbf{1} + \mathbf{1}$ has two, $l \cdot \langle\,\rangle$ and $r \cdot \langle\,\rangle$. Consequently, no term can induce a surjection from the values of $\mathbf{1}$ to the values of $\mathbf{1} + \mathbf{1}$.

6.3 Distributivity

Products distribute over sums, i.e., for all τ, ρ, and σ:

$$\tau \otimes (\rho + \sigma) \cong \tau \otimes \rho + \tau \otimes \sigma.$$

This isomorphism is witnessed by the following pair of mutual inverses:

$$\begin{aligned}
\xi &= \lambda x.\textbf{case}\ x\ \{\langle t, s\rangle \Rightarrow \textbf{case}\ s\ \{l \cdot u \Rightarrow l \cdot \langle t, u\rangle \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mid r \cdot w \Rightarrow r \cdot \langle t, w\rangle\}\}, \\
\xi^{-1} &= \lambda x.\textbf{case}\ x\ \{l \cdot y \Rightarrow \textbf{case}\ y\ \{\langle t, r\rangle \Rightarrow \langle t, l \cdot r\rangle\} \\
&\qquad\qquad\qquad \mid r \cdot y \Rightarrow \textbf{case}\ y\ \{\langle t, s\rangle \Rightarrow \langle t, r \cdot s\rangle\}\}.
\end{aligned}$$

In the case of ξ, we take in a term x of type $\tau \otimes (\rho + \sigma)$ and decompose it into a $t : \tau$ and an $s : \rho + \sigma$. We do case analysis on s to determine if is a left injection or a right injection. If it is a left injection, then we get a term $u : \rho$ and inject the pair $\langle t, u\rangle$ into the left to get a term of type $\tau \otimes \rho + \tau \otimes \sigma$. We proceed symmetrically if s reduces to a left right injection. The definition of ξ^{-1} is similar. The details are left as an exercise.

6.4 Currying

We can *curry* functions, i.e., for all τ, ρ, and σ:

$$\tau \to (\rho \to \sigma) \cong (\tau \otimes \rho) \to \sigma.$$

This isomorphism is witnessed by the following pair of mutual inverses:

$$\begin{aligned} \zeta &= \lambda f.\lambda x.\textbf{case}\ x\ \{\langle t, r\rangle \Rightarrow ftr\}, \\ \zeta^{-1} &= \lambda f.\lambda t.\lambda r.f\langle t, r\rangle. \end{aligned}$$

In the ζ case, the intuition is that we must take in a function $f : \tau \to (\rho \to \sigma)$ and produce a function of type $(\tau \otimes \rho) \to \sigma$. We do so by taking in a term x of type $\tau \otimes \rho$, and eliminating it to get terms $t : \tau$ and $r : \rho$ to which we can apply f and ft, respectively.

In the ζ^{-1} case, the intuition is that we must take in a function $f : (\tau \otimes \rho) \to \sigma$ and produce a function of type $\tau \to (\rho \to \sigma)$. To do so, we need to take in a term $t : \tau$ and produce a term of type $\rho \to \sigma$. To produce such a term, we take in a $r : \rho$ and must produce a term of type σ. By pairing together t and r, we get a term $\langle t, r\rangle : \tau \otimes \rho$ to which we can apply f to get a term of type σ.

As we discovered in class, we could instead take the left and right projections out of x:

$$\zeta = \lambda f.\lambda x.f(\textbf{case}\ x\ \{\langle l, \rangle \Rightarrow l\})(\textbf{case}\ x\ \{\langle \_, r\rangle \Rightarrow r\}).$$

Lecture Notes on General Recursion & Recursive Types

15-814: Types and Programming Languages
Ryan Kavanagh

Lecture 8
Thursday, September 27, 2018

1 Introduction

To date, our programming examples have been limited to types with no self-referential structure: functions, sums, and products. Yet many useful types are self-referential, such as natural numbers, lists, streams, etc. Not only have our types not exhibited any form of self-reference, but neither have our programs. Today, we will see how to capture recursion in a typed setting, before then expanding our type system with recursive types. Before doing so, we make a brief digression to generalize binary sums (Lecture 6) to finite sums. Though we could encode finite sums as iterated or nested binary sums, the generalization is straightforward and it will allow us to use more descriptive labels for our injections than left "l" and right "r".

2 Finite sums

We generalize the definition of binary sums to finite sums indexed by some finite set I. We begin by extending our syntax as follows:

$$
\begin{array}{llr}
\tau ::= \cdots & & \\
\quad \mid \sum_{I}(i : \tau_i) & & \text{sum of types } \tau_i \text{ tagged with } i \text{ for } i \in I \\
e ::= \cdots & & \\
\quad \mid i \cdot e & & \text{inject } e \text{ with tag } i \\
\quad \mid \textbf{case}\ e\ \{i \cdot x_i \Rightarrow e_i\}_{i \in I} & & \text{elimination form for finite sums}
\end{array}
$$

We can use this syntax to give a more suggestive definition of the **bool** type:

$$\begin{aligned} \mathbf{bool} &= (t : \mathbf{1}) + (f : \mathbf{1}) \\ \mathbf{true} &= t \cdot \langle\,\rangle \\ \mathbf{false} &= f \cdot \langle\,\rangle \\ \mathbf{if}\ e\ \mathbf{then}\ e_t\ \mathbf{else}\ e_f &= \mathbf{case}\ e\ \{t \cdot \_ \Rightarrow e_t \mid f \cdot \_ \Rightarrow e_f\}. \end{aligned}$$

The statics and the dynamics generalize from the binary case in the obvious manner; the reader is referred to [Har16, § 11.2] for details.

3 General recursion

Let us think back to how we implemented recursion in the simply-typed λ-calculus. We wanted to define a recursive function

$$F = \cdots F \cdots ,$$

but found that we could not directly do so because of the circular or self-referential nature of the definition. To get around this, we abstracted out the F on the right hand side

$$F = (\lambda f. \cdots f \cdots)F$$

and observed that we could define F as the fixed point of $\zeta = \lambda f. \cdots f \cdots$. We constructed this fixed point using the fixed point combinator $\mathbf{Y}$, getting

$$F = \mathbf{Y}\zeta = \zeta(\mathbf{Y}\zeta) = \cdots \mathbf{Y}\zeta \cdots = \cdots F \cdots$$

as desired. Though we cannot encode the fixed point combinator in our typed setting, we can introduce a new term former and imbue it with the appropriate semantics. To this end, we introduce a new fixed point construct $\mathbf{fix}(x.e)$, with the intention that, as was the case with $\mathbf{Y}$, we get

$$\mathbf{fix}(f. \cdots f \cdots) = \cdots \mathbf{fix}(f. \cdots f \cdots) \cdots .$$

Its statics are captured by the rule

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mathbf{fix}(x.e) : \tau}\ (\textsc{Fix})$$

The intention is that x stand for any self-referential use of e in e. The operational behaviour is captured by the rule

$$\frac{}{\mathbf{fix}(x.e) \mapsto [\mathbf{fix}(x.e)/x]e}\;(\textsc{Fix-Step})$$

With this construction, we can easily implement a divergent term:

$$\mathit{loop} = \mathbf{fix}(x.x).$$

Then $\mathit{loop} \mapsto [\mathbf{fix}(x.x)/x]x = \mathit{loop}$.

This isn't a very interesting example, so let us consider recursive functions on the natural numbers. Define the type of natural numbers to be

$$\mathbf{nat} \text{ ``$=$'' } (\mathsf{z} : \mathbf{1}) + (\mathsf{s} : \mathbf{nat}).$$

This definition is dubious because we are defining **nat** in terms of itself: the type **nat** appears on both sides of the equation and it is unclear that a unique solution exists. We will give a correct definition in section 4, but let us assume the above definition for the sake of illustrating $\mathbf{fix}(x.e)$. We define numerals as follows:

$$\begin{aligned} \overline{0} &= \mathsf{z} \cdot \langle\,\rangle, \\ \overline{n+1} &= \mathsf{s} \cdot \overline{n}. \end{aligned}$$

We can now implement various functions on natural numbers:

$$\begin{aligned} \mathsf{pred} &= \lambda n.\mathbf{case}\; n\; \{\mathsf{z} \cdot \_ \Rightarrow \overline{0} \mid \mathsf{s} \cdot n' \Rightarrow n'\} \\ \mathsf{add} &= \mathbf{fix}(f.\lambda n.\lambda m.\mathbf{case}\; n\; \{\mathsf{z} \cdot \_ \Rightarrow m \mid \mathsf{s} \cdot n' \Rightarrow \mathsf{s} \cdot (fn'm)\}) \\ \mathsf{mult} &= \mathbf{fix}(f.\lambda n.\lambda m.\mathbf{case}\; n\; \{\mathsf{z} \cdot \_ \Rightarrow m \mid \mathsf{s} \cdot n' \Rightarrow \mathsf{add}(m)(fn'm)\}) \\ \mathsf{fact} &= \mathbf{fix}(f.\lambda n.\mathbf{case}\; n\; \{\mathsf{z} \cdot \_ \Rightarrow \overline{1} \mid \mathsf{s} \cdot n' \Rightarrow \mathsf{mult}(n)(fn')\}) \end{aligned}$$

To illustrate the typing rule for the $\mathbf{fix}(x.e)$ construct, we show that

$$\cdot \vdash \mathsf{add} : \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat}.$$

Let $\Gamma = f : \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat}, n : \mathbf{nat}, m : \mathbf{nat}$. Then:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\Gamma \vdash n : (\mathsf{z} : \mathbf{1}) + (\mathsf{s} : \mathbf{nat})}\;(\textsc{Var}) \qquad \cfrac{}{\Gamma, \_ : \mathbf{1} \vdash m : \mathbf{nat}}\;(\textsc{Var}) \qquad \mathcal{D}}{\Gamma \vdash \mathbf{case}\; n\; \{\mathsf{z} \cdot \_ \Rightarrow m \mid \mathsf{s} \cdot n' \Rightarrow \mathsf{s} \cdot (fn'm)\}}\;(\text{E-+})}{f : \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat}, n : \mathbf{nat} \vdash \lambda m.\mathbf{case}\; n\; \{\cdots\} : \mathbf{nat} \to \mathbf{nat}}\;(\textsc{Lam})}{f : \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat} \vdash \lambda n.\lambda m.\mathbf{case}\; n\; \{\cdots\} : \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat}}\;(\textsc{Lam})}{\cdot \vdash \mathbf{fix}(f.\lambda n.\lambda m.\mathbf{case}\; n\; \{\cdots\}) : \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat}}\;(\textsc{Fix})$$

where $T_f = \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat}$ and $\mathcal{D}$ is the derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, n' : \mathbf{nat} \vdash f : T_f}\;(\textsc{Var}) \quad \dfrac{}{\Gamma, n' : \mathbf{nat} \vdash m : \mathbf{nat}}\;(\textsc{Var})}{\Gamma, n' : \mathbf{nat} \vdash f n' : \mathbf{nat} \to \mathbf{nat}}\;(\textsc{App}) \quad \dfrac{}{\Gamma, n' : \mathbf{nat} \vdash n' : \mathbf{nat}}\;(\textsc{Var})}{\Gamma, n' : \mathbf{nat} \vdash f n' m : \mathbf{nat}}\;(\textsc{App})}{\Gamma, n' : \mathbf{nat} \vdash \mathsf{s} \cdot (f n' m) : \mathbf{nat}}\;(\text{I-+})$$

We can also define the type of lists of elements of type τ:

$$\tau\ \mathbf{list}\ \text{“=”}\ (\mathsf{nil} : \mathbf{1}) + (\mathsf{cons} : \tau \otimes (\tau\ \mathbf{list})).$$

We can then write an append function, that concatenates two lists:

$$\begin{aligned} \mathsf{append} = \mathbf{fix}(a.\lambda l.\lambda r.\mathbf{case}\ l\ \{&\mathsf{nil} \cdot \_ \Rightarrow r \\ &\mid \mathsf{cons} \cdot p \Rightarrow \mathbf{case}\ p\ \{\langle h, t \rangle \Rightarrow \mathsf{cons} \cdot \langle h, atr \rangle\}\}) \end{aligned}$$

In assignment 2, you are asked to explore *lazy products* $\tau \mathbin{\&} \sigma$. It is interesting to reflect on what would have happened had we used lazy products instead of eager products in the definition of τ **list**. That is, what values inhabit the following type:

$$\tau\ \mathbf{mystery}\ \text{“=”}\ (\mathsf{nil} : \mathbf{1}) + (\mathsf{cons} : \tau \mathbin{\&} (\tau\ \mathbf{mystery}))?$$

4 Recursive types

We have so far played fast and loose with our definitions of recursive types. We defined recursive types as solutions to type equations, where the type we were defining appeared on both sides of the equation. It is unclear whether a solution to such an equation exists, let alone if it is unique.

To put recursive types on surer footing, we begin by extending our syntax of types and terms:

$$\begin{array}{ll} \tau ::= \cdots & \\ \quad \mid \rho(\alpha.\tau) & \text{recursive type} \\ e ::= \cdots & \\ \quad \mid \mathbf{fold}(e) & \text{fold } e \text{ into a recursive type} \\ \quad \mid \mathbf{unfold}(e) & \text{unfold } e \text{ out of a recursive type} \end{array}$$

We remark that α may appear bound in τ, i.e., that τ may depend on α. Indeed, the intention is that the bound occurrences of α in τ stand in for any self-reference in τ.

The intention is that we "fold" an expression e of type $[\rho(\alpha.\tau)/\alpha]\tau$ into the recursive type $\rho(\alpha.\tau)$:

$$\frac{\Gamma \vdash e : [\rho(\alpha.\tau)/\alpha]\tau}{\Gamma \vdash \mathbf{fold}(e) : \rho(\alpha.\tau)}\ (\textsc{Fold})$$

Symmetrically, given an expression e of type $\rho(\alpha.\tau)$, we can "unfold" its type to get an expression of type $[\rho(\alpha.\tau)/\alpha]\tau$:

$$\frac{\Gamma \vdash e : \rho(\alpha.\tau)}{\Gamma \vdash \mathbf{unfold}(e) : [\rho(\alpha.\tau)/\alpha]\tau}\ (\textsc{Fold})$$

To illustrate these concepts, we revisit the type **nat**. We define

$$\mathbf{nat} = \rho(\alpha.(\mathsf{z} : \mathbf{1}) + (\mathsf{s} : \alpha)).$$

We then define

$$\begin{aligned} \overline{0} &= \mathbf{fold}(\mathsf{z} \cdot \langle\,\rangle), \\ \overline{n+1} &= \mathbf{fold}(\mathsf{s} \cdot \overline{n}). \end{aligned}$$

These definitions type-check:

$$\frac{\dfrac{\dfrac{}{\cdot \vdash \langle\,\rangle : \mathbf{1}}\ (\text{I-}\mathbf{1})}{\cdot \vdash \mathsf{z} \cdot \langle\,\rangle : (\mathsf{z} : \mathbf{1}) + (\mathsf{s} : \rho(\alpha.(\mathsf{z} : \mathbf{1}) + (\mathsf{s} : \alpha)))}\ (\text{I-}+)}{\cdot \vdash \mathbf{fold}(\mathsf{z} \cdot \langle\,\rangle) : \rho(\alpha.(\mathsf{z} : \mathbf{1}) + (\mathsf{s} : \alpha))}\ (\textsc{Fold})$$

and

$$\frac{\dfrac{\cdot \vdash \overline{n} : \rho(\alpha.(\mathsf{z} : \mathbf{1}) + (\mathsf{s} : \alpha))}{\cdot \vdash \mathsf{s} \cdot \overline{n} : (\mathsf{z} : \mathbf{1}) + (\mathsf{s} : \rho(\alpha.(\mathsf{z} : \mathbf{1}) + (\mathsf{s} : \alpha)))}\ (\text{I-}+)}{\cdot \vdash \mathbf{fold}(\mathsf{s} \cdot \overline{n}) : \rho(\alpha.(\mathsf{z} : \mathbf{1}) + (\mathsf{s} : \alpha))}\ (\textsc{Fold})$$

We can recover our examples from Section 3 by introducing $\mathbf{fold}(\cdot)$ and $\mathbf{unfold}(\cdot)$ in the appropriate places:

$$\begin{aligned} \mathsf{pred} &= \lambda n.\mathbf{case}\ (\mathbf{unfold}(n))\ \{\mathsf{z} \cdot \_ \Rightarrow \overline{0} \mid \mathsf{s} \cdot n' \Rightarrow n'\} \\ \mathsf{add} &= \mathbf{fix}(f.\lambda n.\lambda m.\mathbf{case}\ (\mathbf{unfold}(n))\ \{\mathsf{z} \cdot \_ \Rightarrow m \mid \mathsf{s} \cdot p \Rightarrow \mathbf{fold}(\mathsf{s} \cdot (fpm))\}) \\ \mathsf{mult} &= \mathbf{fix}(f.\lambda n.\lambda m.\mathbf{case}\ (\mathbf{unfold}(n))\ \{\mathsf{z} \cdot \_ \Rightarrow m \mid \mathsf{s} \cdot n' \Rightarrow \mathsf{add}(m)(fn'm)\}) \\ \mathsf{fact} &= \mathbf{fix}(f.\lambda n.\mathbf{case}\ (\mathbf{unfold}(n))\ \{\mathsf{z} \cdot \_ \Rightarrow \overline{1} \mid \mathsf{s} \cdot n' \Rightarrow \mathsf{mult}(n)(fn')\}) \end{aligned}$$

We give terms inhabiting recursive types an eager dynamics:

$$\frac{e\ \mathit{val}}{\mathbf{fold}(e)\ \mathit{val}} \qquad \frac{e \mapsto e'}{\mathbf{fold}(e) \mapsto \mathbf{fold}(e')}$$

$$\frac{e \mapsto e'}{\mathbf{unfold}(e) \mapsto \mathbf{unfold}(e')} \qquad \frac{\mathbf{fold}(e)\ \mathit{val}}{\mathbf{unfold}(\mathbf{fold}(e)) \mapsto e}$$

These definitions satisfy the usual progress and preservation properties.

We illustrate our dynamics by considering the following example over the natural numbers, recalling that $\overline{1} \equiv \mathbf{fold}(\mathsf{s} \cdot \overline{0})$:

$$\begin{aligned}
&\mathsf{add}\,\overline{1}\,\overline{2}\\
&\equiv \mathbf{fix}(f.\lambda n.\lambda m.\mathbf{case}\ (\mathbf{unfold}(n))\ \{\mathsf{z} \cdot \_ \Rightarrow m \mid \mathsf{s} \cdot p \Rightarrow \mathbf{fold}(\mathsf{s} \cdot (f p m))\})\overline{1}\,\overline{2}\\
&\mapsto (\lambda n.\lambda m.\mathbf{case}\ (\mathbf{unfold}(n))\ \{\mathsf{z} \cdot \_ \Rightarrow m \mid \mathsf{s} \cdot p \Rightarrow \mathbf{fold}(\mathsf{s} \cdot (\mathsf{add}\,p\,m))\})\overline{1}\,\overline{2}\\
&\mapsto (\lambda m.\mathbf{case}\ (\mathbf{unfold}(\overline{1}))\ \{\mathsf{z} \cdot \_ \Rightarrow m \mid \mathsf{s} \cdot p \Rightarrow \mathbf{fold}(\mathsf{s} \cdot (\mathsf{add}\,p\,m))\})\overline{2}\\
&\mapsto \mathbf{case}\ \mathbf{unfold}(\mathbf{fold}(\mathsf{s} \cdot \overline{0}))\ \{\mathsf{z} \cdot \_ \Rightarrow \overline{2} \mid \mathsf{s} \cdot p \Rightarrow \mathbf{fold}(\mathsf{s} \cdot (\mathsf{add}\,p\,\overline{2}))\}\\
&\mapsto \mathbf{fold}(\mathsf{s} \cdot \mathsf{add}\,\overline{0}\,\overline{2})\\
&\mapsto^{*} \mathbf{fold}(\mathsf{s} \cdot \overline{2})\\
&\equiv \overline{3}
\end{aligned}$$

References

[Har16] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2nd edition, 2016.

Lecture Notes on
Data Representation

15-814: Types and Programming Languages
Frank Pfenning

Lecture 9
Tuesday, October 2, 2018

1 Introduction

In this lecture we'll see our type system in action. In particular we will see how types enable and guide data representation. We first look at a traditional problem (representing numbers in binary form) then at a less traditional one (representing the untyped λ-calculus). Before that, we'll review recursive types and their properties, since they play a central role in what follows.

2 Natural Numbers, Revisited

Recall that we were thinking of natural numbers as the type

$$\mathit{nat} = 1 + (1 + (1 + \ldots))$$

which doesn't seem directly implementable. Instead, we noticed that under the approach we have

$$\mathit{nat} \text{ ``='' } 1 + \mathit{nat}$$

where the notion of equality between these two types was a bit murky. So we devised an explicit construction $\rho\,\alpha.\,\tau$ to form a recursive type of this nature.

$$\mathit{nat} = \rho\,\alpha.\,1 + \alpha$$

The *constructor* for elements of recursive types is fold, while unfold *destructs* elements.

$$\frac{\Gamma \vdash e : [\rho\,\alpha.\,\tau/\alpha]\tau}{\Gamma \vdash \mathsf{fold}\, e : \rho\,\alpha.\,\tau} \qquad \frac{\Gamma \vdash e : \rho\,\alpha.\,\tau}{\Gamma \vdash \mathsf{unfold}\, e : [\rho\,\alpha.\,\tau/\alpha]\tau}$$

This "unfolding" of the recursion seems like a strange operation, and it is. For example, for all other data constructors the components have a smaller type than the constructed expression, but that's not the case here because $[\rho\,\alpha.\,\tau/\alpha]\tau$ is in general a larger type than $\rho\,\alpha.\,\tau$. To get more intuition, let's look at the special case of these rules for natural numbers. We exploit the definition of *nat* in order to avoid explicitly use of the ρ binder and substitution.

$$[\rho\,\alpha.\,1 + \alpha/\alpha](1 + \alpha) = 1 + \mathit{nat}$$

With this shortcut, the specialized rules are

$$\frac{\Gamma \vdash e : 1 + \mathit{nat}}{\Gamma \vdash \mathsf{fold}\, e : \mathit{nat}} \qquad \frac{\Gamma \vdash e : \mathit{nat}}{\Gamma \vdash \mathsf{unfold}\, e : 1 + \mathit{nat}}$$

When recursive types are given names (which is usually the case), this technique makes it much easier to see how the fold and unfold operations actually work.

The funky equality from the beginning of the lecture is actually an *isomorphism*, that is,

$$\mathit{nat} \cong 1 + \mathit{nat}$$

In fact, the functions going back and forth are exactly fold and unfold.

$$\mathit{nat} \quad \underset{\underset{\mathsf{unfold}}{\longrightarrow}}{\overset{\overset{\mathsf{fold}}{\longleftarrow}}{\cong}} \quad 1 + \mathit{nat}$$

We can (re)write simple programs. As we did in lecture, you should write these programs following the structure of the type; here we just show the final code.

$$\begin{array}{lcl} \mathit{zero} : \mathit{nat} & = & \mathsf{fold}\,(l \cdot \langle\,\rangle) \\ \mathit{succ} : \mathit{nat} \to \mathit{nat} & = & \lambda n.\,\mathsf{fold}\,(r \cdot n) \end{array}$$

In order to check the isomorphism, we need to show that the functions compose to the identity in both directions. That is:

(i) For every value $v : 1 + \mathit{nat}$, $\mathsf{unfold}\,(\mathsf{fold}\, v) = v$, and

(ii) for every value v : *nat*, $\mathsf{fold}\,(\mathsf{unfold}\,v) = v$

Before we can prove this, we should write down the definition of values and the operational semantics. The constructor is fold, and we had decided to make it eager, that is

$$\frac{e\ \mathsf{val}}{\mathsf{fold}\,e\ \mathsf{val}}$$

The destructor is unfold, so it acts on a value of the expect form, namely a fold.

$$\frac{v\ \mathsf{val}}{\mathsf{unfold}\,(\mathsf{fold}\,v) \mapsto v}$$

Finally, we have congruence rules: for the constructor because it is eager, and for the destructor because we need to reduce the argument until it exposes the constructor.

$$\frac{e \mapsto e'}{\mathsf{fold}\,e \mapsto \mathsf{fold}\,e'} \qquad \frac{e \mapsto e'}{\mathsf{unfold}\,e \mapsto \mathsf{unfold}\,e'}$$

Back to our putative isomorphism. The first direction is almost trivial, since we can directly step.

(i) $\mathsf{unfold}\,(\mathsf{fold}\,v) \mapsto v$ since v val.

The second part is slightly more complex

(ii) We want to show that $\mathsf{fold}\,(\mathsf{unfold}\,v) = v$ for any value v : *nat*. The left-hand side does not appear to reduce, because fold is the constructor. However, because v : *nat* is a value we know it must have the form $\mathsf{fold}\ v'$ for a value v' (by the canonical forms theorem, see below) and then we reason:

$$\begin{array}{lll} & \mathsf{fold}\,(\mathsf{unfold}\,v) & \\ = & \mathsf{fold}\,(\mathsf{unfold}\,(\mathsf{fold}\ v')) & \text{since } v = \mathsf{fold}\,v' \\ \mapsto & \mathsf{fold}\,v' & \text{by computation rule} \\ = & v & \text{since } v = \mathsf{fold}\,v' \end{array}$$

Before stating the canonical form theorem it is worth realizing that properties (i) and (ii) actually do not depend on the particular recursive type *nat* but hold for any recursive type $\rho\,\alpha.\,\tau$. This means that we have in general

$$\rho\,\alpha.\,\tau \quad \underset{\xrightarrow[\mathsf{unfold}]{}}{\overset{\xleftarrow{\mathsf{fold}}}{\cong}} \quad [\rho\,\alpha.\,\tau/\alpha]\rho$$

This is why we call types in this form *isorecursive*. There is a different form called *equirecursive* which attempts to get by without explicit fold and unfold constructs. Programs become more succinct, but type-checking easily becomes undecidable or impractical, depending on the details of the language. We therefore take the more explicit isorecursive approach here.

Theorem 1 (Canonical forms for recursive types)
If $\cdot \vdash v : \rho\,\alpha.\,\tau$ *and* v *val then* $v = \mathsf{fold}\ v'$ *for some* v' *val.*

Proof: By case analysis of values and typing rules. □

3 Representing Binary Numbers

Natural numbers in unary form are an elegant foundational representation, but the size of the representation of n is linear in n. We can do much better if we have a *binary* representation with two bits. A binary number then is a finite string of bits, satisfying something like

$$\mathit{bin} \cong \mathit{bin} + \mathit{bin} + 1$$

where the first summand represents a bit 0, the second a bit 1, and the last the empty string of bits. Code is easier to write if we use the n-ary form of the sum where each alternative is explicitly labeled.

$$\mathit{bin} \cong (\mathsf{b0} : \mathit{bin}) + (\mathsf{b1} : \mathit{bin}) + (\epsilon : 1)$$

Here we have used the labels b0 (for a 0 bit), b1 (for a 1 bit), and ϵ (for the empty bit string).

Now it is convenient (but not necessary) to represent 0 by the empty bit string.

$$\mathit{bzero} : \mathit{bin} \quad = \quad \mathsf{fold}\,(\epsilon \cdot \langle\,\rangle)$$

We can also construct larger numbers from smaller ones by adding a bit at the end. For the purposes of writing programs, it is most convenient to represent numbers in "little endian" form, that is, the least significant bit comes first. The two constructors then either double the number n to $2n$ (if we add bit 0) or $2n + 1$ if we add bit 1.

$$\begin{array}{rcl} \mathit{dbl0} : \mathit{bin} \to \mathit{bin} & = & \lambda x.\,\mathsf{fold}\,(\mathsf{b0} \cdot x) \\ \mathit{dbl1} : \mathit{bin} \to \mathit{bin} & = & \lambda x.\,\mathsf{fold}\,(\mathsf{b1} \cdot x) \end{array}$$

As a sample program that must analyze the structure of numbers in binary form, consider a function to increment a number. In order to analyze the argument of type *bin* we must first unfold its represenation to a sum and then case over the possible summands. There are three possibilities, so our code so far has the form

$$
\begin{array}{l}
\mathit{inc} : \mathit{bin} \to \mathit{bin} = \\
\quad \lambda x.\, \mathsf{case}\ (\mathsf{unfold}\, x) \\
\qquad\qquad \{\, \mathsf{b0} \cdot y \Rightarrow \ldots \\
\qquad\qquad \mid \mathsf{b1} \cdot y \Rightarrow \ldots \\
\qquad\qquad \mid \epsilon \cdot y \Rightarrow \ldots \,\}
\end{array}
$$

In each branch, the missing code should have type *bin*. In the case of $\mathsf{b0} \cdot y$ we just need to flip the lowest bit from b0 to b1 and keep the rest of the bit string the same.

$$
\begin{array}{l}
\mathit{inc} : \mathit{bin} \to \mathit{bin} = \\
\quad \lambda x.\, \mathsf{case}\ (\mathsf{unfold}\, x) \\
\qquad\qquad \{\, \mathsf{b0} \cdot y \Rightarrow \mathsf{fold}\ (\mathsf{b1} \cdot y) \\
\qquad\qquad \mid \mathsf{b1} \cdot y \Rightarrow \ldots \\
\qquad\qquad \mid \epsilon \cdot y \Rightarrow \ldots \,\}
\end{array}
$$

In the second branch, we need to flip b1 to b0, and we also need to implement the "carry", which means that we have to *increment* the remaining higher-order bits.

$$
\begin{array}{l}
\mathit{inc} : \mathit{bin} \to \mathit{bin} = \\
\quad \lambda x.\, \mathsf{case}\ (\mathsf{unfold}\, x) \\
\qquad\qquad \{\, \mathsf{b0} \cdot y \Rightarrow \mathsf{fold}\ (\mathsf{b1} \cdot y) \\
\qquad\qquad \mid \mathsf{b1} \cdot y \Rightarrow \mathsf{fold}\ (\mathsf{b0} \cdot (\mathit{inc}\ y)) \\
\qquad\qquad \mid \epsilon \cdot y \Rightarrow \ldots \,\}
\end{array}
$$

Finally, in the last case we need to return the representation of the number 1, because $\mathsf{fold}\ (\epsilon \cdot \langle\,\rangle)$ represents 0. We obtain it from the the representation of 0 (which we called *bzero*) by adding a bit 1.

$$
\begin{array}{l}
\mathit{inc} : \mathit{bin} \to \mathit{bin} = \\
\quad \lambda x.\, \mathsf{case}\ (\mathsf{unfold}\, x) \\
\qquad\qquad \{\, \mathsf{b0} \cdot y \Rightarrow \mathsf{fold}\ (\mathsf{b1} \cdot y) \\
\qquad\qquad \mid \mathsf{b1} \cdot y \Rightarrow \mathsf{fold}\ (\mathsf{b0} \cdot (\mathit{inc}\ y)) \\
\qquad\qquad \mid \epsilon \cdot y \Rightarrow \mathsf{fold}\ (\mathsf{b1} \cdot \mathit{bzero}) \,\}
\end{array}
$$

In the last branch, $y : 1$ and it is unused. As suggest in lecture, we could have written instead

$$
\mid \epsilon \cdot y \Rightarrow \mathsf{fold}\ (\mathsf{b1} \cdot \mathsf{fold}\ (\epsilon \cdot y))
$$

In this program we largely reduced the operations back to fold and explicitly labeled sums, but we could have also used the *dbl0* and *dbl1* functions.

At this point we have seen all the pieces we need to implement addition, multiplication, subtraction, etc. on the numbers in binary form.

4 Representing the Untyped λ-Calculus

Recall that in the pure, untyped lambda calculus we only have three forms of expression: λ-abstraction $\lambda x.\, e$, application $e_1\, e_2$ and variables x. A completely straightforward representation would be given by the following recursive type:

$$\begin{array}{lcl} \mathit{var} & \cong & \mathit{nat} \\ \mathit{exp} & \cong & (\mathsf{lam} : \mathit{var} \otimes \mathit{exp}) + (\mathsf{app} : \mathit{exp} \otimes \mathit{exp}) + (\mathsf{v} : \mathit{var}) \end{array}$$

Here we have chosen variables to be represented by natural numbers because we need unboundedly many different ones.

This representation is fine, but it turns out to be somewhat awkward to work with. One issue is that we have already said that $\lambda x.\, x$ and $\lambda y.\, y$ should be indistinguishable, but in the representation above they are (for example, x might be the number 35 and y the number 36.

In order to solve this problem, de Bruijn [dB72] developed a representation where we cannot distinguish these two terms. It is based on the idea that a variable occurrence should be a *pointer* back to the place where it is bound. A convenient representation for such a pointer is a natural number that indicates how many binders we have to traverse upwards to reach the appropriate λ-abstraction. For example:

$$\begin{array}{lcl} \lambda x.\, x & \sim & \lambda.0 \\ \lambda y.\, y & \sim & \lambda.0 \\ \lambda x.\, \lambda y.\, x & \sim & \lambda.\lambda.1 \\ \lambda x.\, \lambda y.\, y & \sim & \lambda.\lambda.0 \end{array}$$

For free variables, we have to assume they are ordered in some context and the variables refers to them, counting from right to left. For example:

$$\begin{array}{lcl} y, z \vdash \lambda x.\, x & \sim & \lambda.0 \\ y, z \vdash \lambda x.\, y & \sim & \lambda.2 \\ y, z \vdash \lambda x.\, z & \sim & \lambda.1 \end{array}$$

One strange effect of this representation (which we did not mention in lecture) is that in de Bruijn notation, the same variable may occur with different numbers in an expression. For example

$$\lambda x.\,(\lambda y.\,x\,y)\,x \quad\sim\quad \lambda.(\lambda.1\,0)\,0$$

The first occurrence of x becomes 1 because it is located under another binder (that for y), while the second occurrence of x becomes 0 because it is not in the scope of the binder on y.

There are some clever algorithms for implementing operations such as substitution on this representation. However, we will move on to an even cooler representation.

5 A Shallow Embedding of the Untyped λ-Calculus

The standard representations we have seen so far are sometimes called *deep embeddings*: objects we are trying to represent simply become "lifeless" data. Any operation on them (as would usually be expected) has to be implemented explicitly and separately.

A *shallow embedding* tries to exploit the features present in the host language (here: our statically typed functional language) as directly as possible. In shallow embeddings mostly we represent only the constructors (or values) and try to implement the destructors. In the case of the untyped λ-calculus, the only constructor is a λ-abstraction so a shallow embedding would postulate

$$\mathsf{E} \;\underset{\xrightarrow[\mathsf{unfold}]{}}{\overset{\xleftarrow{\mathsf{fold}}}{\cong}}\; \mathsf{E} \to \mathsf{E}$$

At first it seems implausible that a type E would be isomorphic to its own function space, but surprisingly we can make it work! In the different context of *denotational semantics* this isomorphism was first solved by Dana Scott [Sco70]. Let's work out the representation function $\ulcorner e \urcorner$ where e is an expression in the untyped λ-calculus. We start with some examples.

$$\ulcorner \lambda x.\,x \urcorner = \underbrace{\ldots}_{:\,\mathsf{E}}$$

We want the representation to be of type E. Since the left-hand side represents a λ-expression, it should be the result of a fold. A fold requires an argument of type $\mathsf{E} \to \mathsf{E}$

$$\ulcorner \lambda x.\,x \urcorner = \mathsf{fold}\;\;\underbrace{\ldots}_{:\,\mathsf{E}\to\mathsf{E}}$$

That should be a λ-expression in the host language, which binds a variable x of type E. The body of the expression is again of E.

$$\ulcorner \lambda x.\, x \urcorner = \mathsf{fold}\ (\lambda x.\ \underbrace{\ldots}_{:\,\mathsf{E}})$$

Because we want to represent the identity function, we finish with

$$\ulcorner \lambda x.\, x \urcorner = \mathsf{fold}\ (\lambda x.\, x)$$

The following two examples work similarly:

$$\begin{array}{l} \ulcorner \lambda x.\, \lambda y.\, x \urcorner = \mathsf{fold}\ (\lambda x.\, \mathsf{fold}\ (\lambda y.\, x)) \\ \ulcorner \lambda x.\, \lambda y.\, y \urcorner = \mathsf{fold}\ (\lambda x.\, \mathsf{fold}\ (\lambda y.\, y)) \end{array}$$

The first hurdle arises when we try to represent application. Let's consider something that might be difficult, namely self-application.

$$\omega = \lambda x.\, x\, x$$

Note that this expression itself cannot in the host language. If there were a typing derivation, it would have to look as follows for some τ, σ, and τ':

$$\dfrac{\dfrac{x : \tau \vdash x : \tau' \to \sigma \quad x : \tau \vdash x : \tau'}{x : \tau \vdash x\, x : \sigma}}{\cdot \vdash \lambda x.\, x\, x : \tau \to \sigma}$$

To complete the derivations, we would have to have simultaneously

$$\tau = \tau' \to \sigma \quad \text{and} \quad \tau = \tau'$$

and there is no solution, because

$$\tau' = \tau' \to \sigma$$

has no solution. Therefore, ω cannot be typed in the simply-typed λ-calculus, even though it is a perfectly honorable untyped term. The key now is the following general table of representations

$$\begin{array}{lcl} \ulcorner \lambda x.\, e \urcorner & = & \mathsf{fold}\, (\lambda x.\, \ulcorner e \urcorner) \\ \ulcorner x \urcorner & = & x \\ \ulcorner e_1\, e_2 \urcorner & = & \underbrace{(\mathsf{unfold}\, \ulcorner e_1 \urcorner)}_{:\,\mathsf{E} \to \mathsf{E}}\ \underbrace{\ulcorner e_2 \urcorner}_{:\,\mathsf{E}} \end{array}$$

To summarize, λ-abstraction becomes a fold, application becomes an unfold, and a variable is represented by a corresponding variable with (for convenience) the same name.

To get back to self-application, we obtain

$$\ulcorner\omega\urcorner = \ulcorner\lambda x.\, x\, x\urcorner = \mathsf{fold}\,(\lambda x.\,(\mathsf{unfold}\, x)\, x) : \mathsf{E}$$

Recall that $\Omega = \omega\,\omega = (\lambda x.\, x\, x)\,(\lambda x.\, x\, x)$ has no normal form in the untyped λ-calculus in the sense that it only reduces to itself. We would expect the representation to diverge as well. Let's check:

$$\begin{array}{lll}
 & \ulcorner\omega\,\omega\urcorner & \\
= & (\mathsf{unfold}\,\ulcorner\omega\urcorner)\,\ulcorner\omega\urcorner & \\
= & (\mathsf{unfold}\,(\mathsf{fold}\,(\lambda x.\,(\mathsf{unfold}\, x)\, x)))\,\ulcorner\omega\urcorner & \\
\mapsto & (\lambda x.\,(\mathsf{unfold}\, x)\, x)\,\ulcorner\omega\urcorner & \text{since } (\lambda x.\,(\mathsf{unfold}\, x)\, x)\ \mathit{val} \\
\mapsto & (\mathsf{unfold}\,\ulcorner\omega\urcorner)\,\ulcorner\omega\urcorner & \text{since } \ulcorner\omega\urcorner\ \mathit{val} \\
= & \ulcorner\omega\,\omega\urcorner &
\end{array}$$

We can see that the representation of Ω also steps to itself, but now in two steps instead of one. That's because the fold/unfold reduction requires one additional step.

We haven't proved this, but without a fixed point constructor for programs ($\mathsf{fix}\, x.\, e$) and without recursive types, every expression in our language reduces to a value. This example demonstrates that this is no longer true in the presence of recursive types. Note that we did not need the fixed point constructor—just the single recursive type $\mathsf{E} = \rho\,\alpha.\, \alpha \to \alpha$ was sufficient.

6 Untyped is Unityped

In the previous section we have seen that there is a compositional embedding of the untyped λ-calculus in our simply-typed language with recursive types. This demonstrates that we don't lose any expressive power by moving to a typed language, as long as we are prepared to accept recursive types. In fact, the whole untyped language is mapped to a *single type* in our host language, so we summarize this by saying that

The untyped λ-calculus is unityped.

It is important to see that the typed language is in fact a *generalization of the untyped language* rather than the other way around. By using fold and unfold we can still express all untyped programs. In the next lecture we will explore this a little bit further to talk about *dynamic typing* and that the observation made in this lecture generalizes to richer settings.

Beyond typing there is one more difference between the untyped λ-calculus and our typed representation that we should not lose sight of. The meaning of an untyped λ-expression is given by its *normal form*, which means we can reduce any subexpression including under λ-abstractions. On the other hand, in the functional host language we do not evaluate under λ-abstractions or lazy pairs. For example, $\lambda z.\, \Omega$ has no normal form, but its representation $\ulcorner \lambda x.\, \Omega \urcorner = \mathsf{fold}\, (\lambda x.\, \ulcorner \Omega \urcorner)$ is a value. So we have to be careful when reasoning about the *operational behavior* of the embedding, which is true for all shallow embeddings.

References

[dB72] N. G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[Sco70] Dana S. Scott. Outline of a mathematical theory of computation. Technical Monograph PRG-2, Oxford University Computing Laboratory, Oxford, England, November 1970.

Lecture Notes on
Parametric Polymorphism

15-814: Types and Programming Languages
Frank Pfenning

Lecture 11
October 9, 2018

1 Introduction

Polymorphism refers to the possibility of an expression to have multiple types. In that sense, all the languages we have discussed so far are polymorphic. For example, we have

$$\lambda x.\, x : \tau \to \tau$$

for any type τ. More specifically, then, we are interested in reflecting this property in a type itself. For example, the judgment

$$\lambda x.\, x : \forall \alpha.\, \alpha \to \alpha$$

expresses all the types above, but now in a single form. This means we can now reason within the type system about polymorphic functions rather than having to reason only at the metalevel with statements such as "*for all types* $\tau, \ldots$".

Christopher Strachey [**?**] distinguished two forms of polymorphism: *ad hoc polymorphism* and *parametric polymorphism*. Ad hoc polymorphism refers to multiple types possessed by a given expression or function which has different implementations for different types. For example, *plus* might have type $\mathit{int} \to \mathit{int} \to \mathit{int}$ but als $\mathit{float} \to \mathit{float} \to \mathit{float}$ with different implementations at these two types. Similarly, a function $\mathit{show} : \forall \alpha.\, \alpha \to \mathit{string}$ might convert an argument of any type into a string, but the conversion function itself will of course have to depend on the type of the argument: printing Booleans, integers, floating point numbers, pairs, etc. are all very different operations.

Even though it is an important concept in programming languages, in this lecture we will not be concerned with ad hoc polymorphism.

In contrast, *parametric polymorphism* refers to a function that behaves the same at all possible types. The identity function, for example, is parametrically polymorphic because it just returns its argument, regardless of its type. The essence of "parametricity" wasn't rigorously captured the beautiful analysis by John Reynolds [**?**], which we will sketch in Lecture 12 on *Parametricity*. In this lecture we will present typing rules and some examples.

2 Extrinsic Polymorphic Typing

We now return to the pure simply-typed λ-calculus.

$$\begin{array}{lcl} \tau & ::= & \alpha \mid \tau_1 \rightarrow \tau_2 \\ e & ::= & x \mid \lambda x.\, e \mid e_1\, e_2 \end{array}$$

We would like the judgment $e : \forall \alpha.\, \tau$ to express that e has *all* types $[\sigma/\alpha]\tau$ for arbitrary σ. This will close an important gap in our earlier development, where the fixed type variables seemed to be inflexible. The construct $\forall \alpha.\, \tau$ binds the type variable α with scope τ. As usual, we identify types that differ only in the names of their bound type variables.

Now we would like to allow the following:

$$\begin{array}{lcl} \mathit{bool} & = & \forall \alpha.\, \alpha \rightarrow \alpha \\[1ex] \mathit{true} & : & \mathit{bool} \\ \mathit{true} & = & \lambda x.\, \lambda y.\, x \\[1ex] \mathit{false} & : & \mathit{bool} \\ \mathit{false} & = & \lambda x.\, \lambda y.\, y \\[2ex] \mathit{nat} & = & \forall \alpha.\, \alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha \\[1ex] \mathit{zero} & : & \mathit{nat} \\ \mathit{zero} & = & \lambda z.\, \lambda s.\, z \\[1ex] \mathit{succ} & : & \mathit{nat} \rightarrow \mathit{nat} \\ \mathit{succ} & : & \lambda n.\, \lambda z.\, \lambda s.\, s\, (n\, z\, s) \end{array}$$

This form of typing is called *extrinsic* because polymorphic types describe a properties of expression, but the expressions themselves remain unchanged.

In an *intrinsic* formulation the expressions themselves carry types and express polymorphism. There are good arguments for both forms of presentation. For the sake of simplicity we use the extrinsic form. This means we depart from our approach so far where each new type constructor was accompanied by corresponding expression constructors and destructors for the new type.

In slightly different forms these calculi were designed independently by Jean-Yves Girard [**?**] and John Reynolds [**?**]. Girard started from higher-order logic while Reynolds from a programming where types could be passed as arguments to functions.

Given that $\lambda x.\, x : \alpha \to \alpha$ we might propose the following simple rule:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \forall \alpha.\, \tau}\ \forall I\ ?$$

We can then derive, for example,

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : \alpha, y : \beta \vdash x : \alpha}\ \textsf{var}}{x : \alpha \vdash \lambda y.\, x : \beta \to \alpha}\ {\to}I}{x : \alpha \vdash \lambda y.\, x : \forall \beta.\, \beta \to \alpha}\ \forall I\ ?}{\cdot \vdash \lambda x.\, \lambda y.\, x : \alpha \to \forall \beta.\, \beta \to \alpha}\ {\to}I}{\cdot \vdash \lambda x.\, \lambda y.\, x : \forall \alpha.\, \alpha \to \forall \beta.\, \beta \to \alpha}\ \forall I\ ?$$

This seems certainly correct. $\lambda x.\, \lambda y.\, x$ should **not** have type $\forall \alpha.\, \alpha {\to} \forall \beta.\, \beta {\to} \beta$. But:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : \alpha, y : \alpha \vdash x : \alpha}\ \textsf{var}}{x : \alpha \vdash \lambda y.\, x : \alpha \to \alpha}\ {\to}I}{x : \alpha \vdash \lambda y.\, x : \forall \alpha.\, \alpha \to \alpha}\ \forall I\ ?}{\cdot \vdash \lambda x.\, \lambda y.\, x : \alpha \to \forall \alpha.\, \alpha \to \alpha}\ {\to}I}{\cdot \vdash \lambda x.\, \lambda y.\, x : \forall \alpha.\, \alpha \to \forall \alpha.\, \alpha \to \alpha}\ \forall I\ ?$$

is clearly incorrect, because by variable renaming we would obtain

$$\lambda x.\, \lambda y.\, x : \forall \alpha.\, \alpha \to \forall \beta.\, \beta \to \beta$$

and the function does not have this type. For example, instantiating α with *bool* and β with *nat* we would conclude the result is of type *nat* when it actually returns a Boolean.

The problem here lies in the instance of $\forall I$ in the third line. We say that $\lambda y.\, x$ has type $\forall \alpha.\, \alpha \rightarrow \alpha$ when it manifestly does not have this type. The problem is that α appears as the type of $x : \alpha$ in the context, so we should be not allowed to quantify over α at this point in the deduction. One way to prohibit this is the have a side condition on the rule

$$\frac{\Gamma \vdash e : \tau \quad \alpha \text{ not free in } \Gamma}{\Gamma \vdash e : \forall \alpha.\, \tau}\ \forall I\ ?$$

This would work, but in the similar situation when we wanted to avoid confusion between expression variables, we postulated that the variable was not already declared. We adopt a similar restriction here by adding a new form of context declaring type variables.

$$\Delta ::= \alpha_1\ \mathit{type}, \ldots, \alpha_n\ \mathit{type}$$

Here, all the α_i must be distinct. The typing judgment is then generalized to

$$\Delta\ ; \Gamma \vdash e : \tau$$

where all the free type variables in Γ and τ are declared in Δ, and (as before) all free expression variables in e are declared in Γ. We express that a type is well-formed in the judgment

$$\Delta \vdash \tau\ \mathit{type}$$

For now, this is just defined compositionally—we show only two rules by way of example. We refer to these as *type formation rules*.

$$\frac{\Delta \vdash \tau_1\ \mathit{type} \quad \Delta \vdash \tau_2\ \mathit{type}}{\Delta \vdash \tau_1 \otimes \tau_2\ \mathit{type}}\ {\otimes}F \qquad \frac{\Delta, \alpha\ \mathit{type} \vdash \tau\ \mathit{type}}{\Delta \vdash \forall \alpha.\, \tau\ \mathit{type}}\ \forall F$$

Now we can formulate the correct rule for introducing the universal quantifier in the type.

$$\frac{\Delta, \alpha\ \mathit{type}\ ; \Gamma \vdash e : \tau}{\Delta\ ; \Gamma \vdash e : \forall \alpha.\, \tau}\ \forall I^{\alpha}$$

In order to keep the context Δ, α *type* well-formed, we imply that α is not already declared in Δ and therefore does not occur in Γ. In future, when we might allow types in expressions, α would not be allowed to occur there as well: it must be globally fresh. Sometimes we add the superscript on the rule to remind ourselves of the freshness condition.

When we instantiate the quantifer to get a more specific types we need to make sure the type we substitute is well-formed.

$$\frac{\Delta\;;\Gamma \vdash e : \forall\alpha.\,\tau \quad \Delta \vdash \sigma\ \mathit{type}}{\Delta\;;\Gamma \vdash e : [\sigma/\alpha]\tau}\ \forall E$$

Now we can easily derive that the Booleans *true* and *false* have the expected type $\forall\alpha.\,\alpha \to \alpha \to \alpha$. How about the conditional? Based on the usual conditional, we might expect

$$\mathit{if} : \mathit{bool} \to \tau \to \tau \to \tau$$

for any type τ, where the first occurrence is the 'then' branch, the second the 'else' branch and the final one the result of the conditional. But we can capture this without having to resort to metalevel quantification:

$$\mathit{if} : \mathit{bool} \to \forall\beta.\,\beta \to \beta \to \beta$$

But this is exactly the same as

$$\mathit{if} : \mathit{bool} \to \mathit{bool}$$

which makes sense since we saw in the lecture on the untyped λ-calculus that

$$\mathit{if} = \lambda b.\,b$$

3 Encoding Pairs

Now that we have the rules in place, we can consider if we can type some of the other constructions of generic data types in the pure λ-calculus. Recall:

$$\begin{array}{lcl} \mathit{pair} & = & \lambda x.\,\lambda y.\,\lambda f.\,f\,x\,y \\ \mathit{fst} & = & \lambda p.\,p\,(\lambda x.\,\lambda y.\,x) \\ \mathit{snd} & = & \lambda p.\,p\,(\lambda x.\,\lambda y.\,y) \end{array}$$

With these definitions we can easily verify

$$\begin{array}{lcl} \mathit{fst}\ (\mathit{pair}\,x\,y) & = & \mathit{fst}\ (\lambda f.\,f\,x\,y) \\ & \mapsto & (\lambda f.\,f\,x\,y)\,(\lambda x.\,\lambda y.\,x) \\ & \mapsto & x \\ \mathit{snd}\ (\mathit{pair}\,x\,y) & \mapsto^* & y \end{array}$$

Can we type these constructors and destructors in the polymorphic λ-calculus? Let's consider defining a type $\mathit{prod}\,\tau\,\sigma$ to form the product of τ and σ.

$$\begin{array}{lcl}
\mathit{prod}\,\tau\,\sigma & = & ??\\
\mathit{pair} & : & \forall\alpha.\,\forall\beta.\,\alpha\rightarrow\beta\rightarrow\mathit{prod}\,\alpha\,\beta\\
\mathit{pair} & = & \lambda x.\,\lambda y.\ \underbrace{\lambda f.\,f\,x\,y}_{:\,\mathit{prod}\,\alpha\,\beta}
\end{array}$$

Since $x:\alpha$ and $y:\beta$ we see that $f:\alpha\rightarrow\beta\rightarrow ?$. But what should be the type $??$ When we apply this function to the first projection (in the function *fst*), then it should be α, when we apply it to the second projection it should be β. Therefore we conjecture it should be an arbitrary type γ. So $f:\alpha\rightarrow\beta\rightarrow\gamma$ and $\mathit{prod}\,\alpha\,\beta=\forall\gamma.\,(\alpha\rightarrow\beta\rightarrow\gamma)\rightarrow\gamma$.

$$\begin{array}{lcl}
\mathit{prod}\,\tau\,\sigma & = & \forall\gamma.\,(\tau\rightarrow\sigma\rightarrow\gamma)\rightarrow\gamma\\
\\
\mathit{pair} & : & \forall\alpha.\,\forall\beta.\,\alpha\rightarrow\beta\rightarrow\mathit{prod}\,\alpha\,\beta\\
\mathit{pair} & = & \lambda x.\,\lambda y.\,\lambda f.\,f\,x\,y\\
\\
\mathit{fst} & : & \forall\alpha.\,\forall\beta.\,\mathit{prod}\,\alpha\,\beta\rightarrow\alpha\\
\mathit{fst} & = & \lambda p.\,p\,(\lambda x.\,\lambda y.\,x)\\
\\
\mathit{snd} & : & \forall\alpha.\,\forall\beta.\,\mathit{prod}\,\alpha\,\beta\rightarrow\beta\\
\mathit{snd} & = & \lambda p.\,p\,(\lambda x.\,\lambda y.\,x)
\end{array}$$

As an example, in the definition of *fst*, the argument p will be of type $\forall\gamma.\,(\alpha\rightarrow\beta\rightarrow\gamma)\rightarrow\gamma$. We instantiate this quantifier with α to get $p:(\alpha\rightarrow\beta\rightarrow\alpha)\rightarrow\alpha$. Now we apply p to the first projection function to obtain the result α.

The observation that it may be difficult to see whether a given expression has a given type is not accidental. In fact, the question whether an expression is typable is undecidable [?], even if significant information is added to the expressions [?, ?].

4 Encoding Sums

Now that we have represented products in the polymorphic λ-calculus, let's try sums. But it is useful to analyze a bit more how we ended up encoding products. The destructor for eager products is

$$\frac{\Gamma\vdash e:\tau\otimes\sigma\quad\Gamma,x:\tau,y:\sigma\vdash e':\tau'}{\Gamma\vdash\textbf{case}\ e\ \{\langle x,y\rangle\Rightarrow e'\}:\tau'}\ \otimes E$$

If we try to reformulate the second premise as a function, it would be $(\lambda x.\, \lambda y.\, e') : \tau \to \sigma \to \tau'$. If we think of this version of **case** as a function, it would have type $\tau \otimes \sigma \to (\tau \to \sigma \to \tau') \to \tau'$. We can now abstract over τ' to obtain $\tau \otimes \sigma \to \forall\gamma.\, (\tau \to \sigma \to \gamma) \to \gamma$. The conjecture about the representation of pairs then arises from replacing the function type an isomorphism

$$\tau \otimes \sigma \cong \forall\gamma.\, (\tau \to \sigma \to \gamma) \to \gamma$$

Our calculations in the previous section lend support to this, although we didn't actually prove such an isomorphism, just that the functions *pair*, *fst*, and *snd* satisfy the given typing and also compute correctly.

Perhaps the elimination rule for sums is subject to a similar interpretation?

$$\frac{\Gamma \vdash e : \tau + \sigma \quad \Gamma, x : \tau \vdash e_1 : \tau' \quad \Gamma, y : \sigma \vdash e_2 : \tau'}{\Gamma \vdash \textbf{case}\ e\ \{l \cdot x \Rightarrow e_1 \mid r \cdot y \Rightarrow e_2\} : \tau'}\ {+E}$$

The second premise would have type $\tau \to \tau'$, the third $\sigma \to \tau'$ and the conclusion has type τ'. Therefore we conjecture

$$\tau + \sigma \cong \forall\gamma.\, (\tau \to \gamma) \to (\sigma \to \gamma) \to \gamma$$

As a preliminary study, we can define

$$\begin{array}{lcl}
\mathit{sum}\,\tau\,\sigma & = & \forall\gamma.\, (\tau \to \gamma) \to (\sigma \to \gamma) \to \gamma \\[1ex]
\mathit{inl} & : & \tau \to \mathit{sum}\,\tau\,\sigma \\
\mathit{inl} & = & \lambda x.\, \lambda l.\, \lambda r.\, l\, x \\[1ex]
\mathit{inr} & : & \sigma \to \mathit{sum}\,\tau\,\sigma \\
\mathit{inr} & = & \lambda y.\, \lambda l.\, \lambda r.\, r\, y \\[1ex]
\mathit{case\_sum} & : & \mathit{sum}\,\tau\,\sigma \to \forall\gamma.\, (\tau \to \gamma) \to (\sigma \to \gamma) \to \gamma \\
\mathit{case\_sum} & = & \lambda s.\, s
\end{array}$$

Then we verify the expected reductions

$$\begin{array}{rcl}
\mathit{case\_sum}\ (\mathit{inl}\ x)\ z_1\ z_2 & \mapsto & (\mathit{inl}\ x)\ z_1\ z_2 \\
 & \mapsto & (\lambda l.\, \lambda r.\, l\, x)\ z_1\ z_2 \\
 & \mapsto^2 & z_1\ x \\[1ex]
\mathit{case\_sum}\ (\mathit{inr}\ y)\ z_1\ z_2 & \mapsto^* & z_2\ y
\end{array}$$

5 Predicativity

First, we summarize the language and rules of the polymorphic λ-calculus, sometimes referred to as System F, in its extrinsic formulation.

$$\begin{array}{lcl} \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \forall \alpha.\, \tau \\ e & ::= & x \mid \lambda x.\, e \mid e_1\, e_2 \end{array}$$

$$\frac{\Delta, \alpha\ \mathit{type}\ ; \Gamma \vdash e : \tau}{\Delta\ ; \Gamma \vdash e : \forall \alpha.\, \tau}\ \forall I^{\alpha} \qquad \frac{\Delta\ ; \Gamma \vdash e : \forall \alpha.\, \tau \quad \Delta \vdash \sigma\ \mathit{type}}{\Delta\ ; \Gamma \vdash e : [\sigma/\alpha]\tau}\ \forall E$$

Several objections may be made to this system. A practical objection is the aforementioned undecidablity of the typing judgment. A philosophical objection is that the system is *impredicative*, that is, the domain of quantification includes the quantifier itself. The latter can be addressed by stratifying the language of types into simple types and type schemas.

$$\begin{array}{llcl} \text{Simple types} & \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \ldots \\ \text{Type schemas} & \sigma & ::= & \forall \alpha.\, \sigma \mid \tau \end{array}$$

This simple stratification allows type inference using an algorithm due to Robin Milner [?], which adopts a previous algorithm by Roger Hindley for combinatory logic [?].

The decomposition into simple types and type schemas is the core of the solution adopted in functional languages such as OCaml, Standard ML, Haskell and even object-oriented languages such as Java where polymorphic functions are implemented in so-called *generic methods* and classes.

The system of type schemes can be further extended (while remaining *predicative*) by considering a hierarchy of *universes* where the quantifier ranges over types at a lower universe. Systems with dependent types such as NuPrl or Agda employ universes for the added generality and sound type-theoretic foundation.

References

[Boe85] Hans Boehm. Partial polymorphic type inference is undecidable. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science (FOCS'85)*, pages 339–345. IEEE, October 1985.

[Gir71] Jean-Yves Girard. Une extension de l'interpretation de gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92, Amsterdam, 1971.

[Hin69] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal Of Computer And System Sciences*, 17:348–375, August 1978.

[Pfe93] Frank Pfenning. On the undecidability of partial polymorphic type reconstruction. *Fundamenta Informaticae*, 19(1,2):185–199, 1993. Preliminary version available as Technical Report CMU-CS-92-105, School of Computer Science, Carnegie Mellon University, January 1992.

[Rey74] John C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, Berlin, 1974. Springer-Verlag.

[Rey83] John C. Reynolds. Types, abstraction, and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing 83*, pages 513–523. Elsevier, September 1983.

[Str00] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13:11–49, 2000. Notes for lecture course given at the International Summer School in Computer Programming at Copenhagen, Denmark, August 1967.

[Wel94] J. B. Wells. Typability and type checking in the second-order lambda-calculus are equivalent and undecidable. In *Proceedings of the 9th Symposium on Logic in Computer Science (LICS'94)*, pages 176–185, 1994.

Lecture Notes on Parametricity

15-814: Types and Programming Languages
Frank Pfenning

Lecture 12
October 11, 2018

1 Introduction

> **Disclaimer:** The material in this lecture is a redux of presentations by Reynolds [Rey83], Wadler [Wad89], and Harper [Har16, Chapter 48]. The quoted "theorems" have not been checked against the details of our presentation of the inference rules and operational semantics.

As discussed in the previous lecture, parametric polymorphism is the idea that a function of type $\forall\alpha.\,\tau$ will "behave the same" on all types σ that might be used for α. This has far-reaching consequences, in particular for modularity and data abstraction. As we will see in a future lecture, if a client to a library that hides an implementation type is *parametric* in this type, then the library implementer or maintainer has the opportunity the replace the implementation with a different one without risk of breaking the client code.

The informal idea that a function behaves parametrically in a type variable α is surprisingly difficult to capture technically. Reynolds [Rey83] realized that is must be done *relationally*. For example, a function $f : \forall\alpha.\,\alpha \to \alpha$ is parametric if for any two types τ and σ, and any relation between values of type τ and σ, if we pass f related arguments it will return related results. This oversimplifies the situation somewhat, but it may provide the right intuition. What Reynolds showed is that in a polymorphic λ-calculus with products and Booleans, all expressions are parametric.

We begin by considering how to define different practically useful notions of equality since, ultimately, parametricity will allow us to prove program equalities.

2 Kleene Equality

The most elementary nontrivial notion of equality just requires that expressions are equal if they evaluate to the same value. We write $e \simeq e'$ (e is *Kleene-equal* to e') if either $e \mapsto^* v$ and $e' \mapsto^* v$ for some value v, or e and e' both diverge.

For the remainder of this lecture we assume that all expressions terminate, that is, evaluate to a value. This means we cannot permit arbitrary recursive types (due to the shallow embedding of the untyped λ-calculus) or arbitrary recursive expressions. We will not be precise about possible syntactic restrictions or extensions in the study of parametricity, but you may consult the given sources for details.

How far does Kleene equality go? For Booleans, for example, it works very well because $e \simeq e' : \mathit{bool}$ is quite sensible: two Boolean expressions are equal if they both evaluate to *true* or they both evaluate to *false*. Similarly, $e \simeq e' : \mathit{nat}$ is the appropriate notion of equality: two expressions of type *nat* are equal if they evaluate to the same natural number.

We can construct bigger types for which Kleene equality still has the right meaning. For example, expressions of type $\mathit{bool} \otimes \mathit{nat}$ should be equal if they evaluate to the same value, which will be in fact a pair of two values whose equality we already understand.

The following so-called *purely positive types* all have *fully observable values*, so Kleene equality equates *exactly* those expressions we would like to be equal.

$$\text{Purely positive types} \quad \tau^+ \quad ::= \quad \tau_1^+ \otimes \tau_2^+ \mid 1 \mid \tau_1^+ + \tau_2^+ \mid 0 \mid \rho\alpha^+.\, \tau^+ \mid \alpha^+$$

With *negative types*, namely $\tau \to \sigma$ or $\tau \mathbin{\&} \sigma$ this is no longer the case. The problem is that we assumed we cannot directly observe the body of a function (which is an arbitrary expression). So, even though intuitively the function on Booleans that doubly-negates its argument and the identity function should be equal. We write $\cong$ for this stronger notion of equality.

$$\lambda x.\, \mathit{not}\, (\mathit{not}\, x) \cong \lambda x.\, x : \mathit{bool} \to \mathit{bool}$$

Another way to express this situation is that we would like to consider functions *extensionally*, via their input/output relationship, but not their definition. There are other aspects of these two functions that are not equal. For example, the identity function has many other types, while the double-negation does not. The identity function is likely to be more efficient. And the former may lose some points in a homework assignment on functional programming because it is less elegant than the latter. Similarly, a function performing bubble sort is extensionally equivalent to one performing quicksort, while otherwise they have many different characteristics.

We ignore intensional aspects of functions in our extensional notions of equality in this lecture. Keeping this in mind, a reasonable approach would be to define

($\rightarrow$) $e \cong e' : \tau_1 \rightarrow \tau_2$ iff for all $v_1 : \tau_1$ we have $e\, v_1 \cong e'\, v_1 : \tau_2$

(&) $e \cong e' : \tau_1 \mathbin{\&} \tau_2$ iff $e \cdot l \cong e' \cdot l : \tau_1$ and $e \cdot r \cong e' \cdot r : \tau_2$

With this definition we can now easily prove that the two Boolean functions above are extensionally equal. The key is to distinguish the cases of $v_1 = \mathit{true}$ and $v_1 = \mathit{false}$ for $v_1 : \mathit{bool}$, which follows from the canonical form theorem.

3 Logical Equality

The notions of Kleene equality and extensional equality are almost sufficient, but when we come to parametricity the extensional equality as sketched so does not function correctly any more. The problem is that we want to compare expressions not at the same, but at related types. This means, for example, that in comparing e and e' and type $\tau_1 \rightarrow \tau_2$ we cannot apply e and e' to the exact *same* value. Instead, we must apply it to *related* values. The second clause for lazy pairs can remain the same. We write $e \sim e' : \tau$ for this refined notion. It is called *logical equality* because it is based on *logical relations*, one of the many connections between logic and computation.

($\rightarrow$) $e \sim e' : \tau_1 \rightarrow \tau_2$ iff for all $v_1 \sim v_1' : \tau_1$ we have $e\, v_1 \sim e'\, v_1' : \tau_2$

(&) $e \sim e' : \tau_1 \mathbin{\&} \tau_2$ iff $e \cdot l \sim e' \cdot l : \tau_1$ and $e \cdot r \sim e' \cdot r : \tau_2$

We can also fill in the definitions for positive type constructors. Because their values are directly observable, we just inspect their form and compare the component values.

(+) $e \sim e' : \tau_1 + \tau_2$ iff either $e \mapsto^* l \cdot v_1$, $e \mapsto^* l \cdot v_1'$ and $v_1 \sim v_1' : \tau_2$ or $e \mapsto^* r \cdot v_2$, $e \mapsto^* r \cdot v_2'$ and $v_2 \sim v_2' : \tau_2$.

(0) $e \sim e' : 0$ never.

($\otimes$) $e \sim e' : \tau_1 \otimes \tau_2$ iff $e \mapsto^* \langle v_1, v_2 \rangle$ and $e' \mapsto^* \langle v_1', v_2' \rangle$ and $v_1 \sim v_1' : \tau_1$ and $v_2 \sim v_2' : \tau_2$

(1) $e \sim e' : 1$ iff $e \mapsto^* \langle\,\rangle$ and $e' \mapsto^* \langle\,\rangle$.

A key aspect of this notion of equality is that it is defined by induction over the structure of the type, which can easily be seen by examining the definitions. We always reduce the question of equality at a type to its components (assuming there are any). This is also the reason why recursive types are excluded, even though large classes of recursive types (in particular, *inductive* and *coinductive* types) can be included systematically.

The question for this lecture is how to extend it to include parametric polymorphism. The straightforward approach

$$e \sim e' : \forall\alpha.\,\tau \text{ iff for all closed } \sigma,\ e \sim e' : [\sigma/\alpha]\tau\ ?$$

fails because the type $[\sigma/\alpha]\tau$ may contain $\forall\alpha.\,\tau$. Moreover, parametric functions are supposed to map related values at related types to related results, and this definition does not express this. Instead, we write $R : \sigma \leftrightarrow \sigma'$ for a relation between expressions $e : \sigma$ and $e' : \sigma'$, and $e\ R\ e'$ if R relates e and e'. Furthermore we require R to be *admissible*, which means it is closed under Kleene equality.[1] That is, if $f \simeq e$, $e \sim e'$, and $e' \simeq f'$ then also $f \sim f'$. Now we define

($\forall$) $e \sim e' : \forall\alpha.\,\tau$ iff for all closed types σ and σ' and admissible relations $R : \sigma \leftrightarrow \sigma'$ we have $e \sim e' : [R/\alpha]\tau$

(R) $e \sim e' : R$ with $e : \tau$, $e' : \tau'$ and $R : \tau \leftrightarrow \tau'$ iff $e\ R\ e'$.

This is a big conceptual step, because what we write as type τ actually now contains admissible relations instead of type variables, as well as ordinary types constructors. Because Kleene equality itself is admissible (it's trivially closed under Kleene equality) we can instantiate α with Kleene equality on the same type σ. A base case of the inductive definitions is then ordinary Kleene equality.

The quantification structure should make it clear that logical equality in general is difficult to establish. It requires a lot: for two arbitrary types and an arbitrary admissible relation, we have to establish properties of e and e'. It is an instructive exercise to check that

$$\lambda x.\,x \sim \lambda x.\,x : \forall\alpha.\,\alpha \to \alpha$$

Conversely, we can imagine that *knowing* that two expressions are parametrically equal is very powerful, because we can instantiate this with arbitrary types σ and σ' and relations between them. The *parametricity theorem* now states that all well-typed expressions are related to themselves.

[1] Other admissibility conditions are possible, depending on the application.

Theorem 1 (Parametricity [Rey83]) *If* $\cdot\,;\cdot \vdash e : \tau$ *then* $e \sim e : \tau$

What we suggested you tediously prove by hand above is an immediate consequence of this theorem.

4 Exploiting Parametricity

Parametricity allows us to deduce information about functions knowing only their (polymorphic) types. For example, with only terminating functions, the type

$$f : \forall\alpha.\, \alpha \to \alpha$$

implies that f is (logically) equivalent to the identity function

$$f \sim \lambda x.\, x : \forall\alpha.\, \alpha \to \alpha$$

Let's prove this. Unfortunately, the first few steps are the "difficult" direction of the parametricity.

By definition, this means to show that

> *For every pair of types* τ *and* τ' *and admissible relation* $R : \tau \leftrightarrow \tau'$, *we have* $f \sim \lambda x.\, x : R \to R$

Now fix arbitrary τ, τ' and R. Next, we use the definition of logical equivalence at type $\tau \to \tau'$ to see that this is equivalent to

> *For every pair of values* $v_0 \sim v_0' : R$ *we have* $f\, v_0 \sim (\lambda x.\, x)\, v_0' : R$

By definition of logical equality at R, this is equivalent to showing that

> $v_0\ R\ v_0'$ *implies* $f\, v_0\ R\ (\lambda x.\, x)\, v_0'$

Since R is closed under Kleene equality this is the case if and only if

> $f\, v_0\ R\ v_0'$ assuming $v_0\ R\ v_0'$

This is true if $f\, v_0 \mapsto^* v_0$ since R is closed under Kleene equality.

So our theorem is complete if we can show that $f\, v_0 \mapsto^* v_0$. To prove this, we use the parametricity theorem with a well-chose relation. We start with

> $f \sim f : \forall\alpha.\, \alpha \to \alpha$ *by parametricity.*

Now define the new relation $S : \tau \leftrightarrow \tau$ such that $v_0\ S\ v_0$ for the specific v_0 from the first half of the argument and close it under Kleene equality. Then

$f \sim f : S \to S$ by definition of $\sim$ at polymorphic type.

Applying the definition of logical equality at function type and the assumption that $v_0\ S\ v_0$ we conclude

$$f\ v_0 \sim f\ v_0 : S$$

which is the same as saying

$$f\ v_0\ S\ f\ v_0$$

By definition, S only relates expressions that are Kleene-equal to v_0, so

$$f\ v_0 \mapsto^* v_0$$

This completes the proof.

Similar proofs show, for example, that $f : \forall\alpha.\, \alpha \to \alpha \to \alpha$ must be equal to the first or second projection function. It is instructive to reason through the details of such arguments, but we move on to a different style of example.

5 Theorems for Free!

A slightly different style of application of parametricity is laid out in Philip Wadler's *Theorems for Free!* [Wad89]. Let's see what we can derive from

$$f : \forall\alpha.\, \alpha \to \alpha$$

First, parametricity tells us

$$f \sim f : \forall\alpha.\, \alpha \to \alpha$$

This time, we pick types τ and τ' and a relation R which is in fact a function $R : \tau \to \tau'$. Evaluation of R has the effect of closing the corresponding relation under Kleene equality. Then

$$f \sim f : R \to R$$

Now, for arbitrary values $x : \tau$ and $x' : \tau'$, $x\ R\ x'$ actually means $R\, x \mapsto^* x'$. Using the definition of $\sim$ at function type we get

$$f\, x \sim f\, (R\, x) : R$$

but this in turn means

$$R\, (f\, x) \simeq f\, (R\, x)$$

This means, for any function $R : \tau \rightarrow \tau'$,

$$R \circ f \simeq f \circ R$$

that is, f commutes with any function R. If τ is non-empty and we have $v_0 : \tau$ and choose $\tau' = \tau$ and $R = \lambda x.\, v_0$ we obtain

$$\begin{array}{rcl} R\,(f\,v_0) & \simeq & v_0 \\ f\,(R\,v_0) & \simeq & f\,v_0 \end{array}$$

so we find $f\,v_0 \simeq v_0$ which, since v_0 was arbitrary, is another way of saying that f is equivalent to the identity function.

For more interesting examples, we extend the notion of logical equivalence to lists. Since lists are inductively defined, we can call upon a general theory to handle them, but since we haven't discussed this theory we give the specific definition.

(τ list) $e \sim e' : \tau$ **list** iff $e \mapsto^* [v_1, \ldots, v_n]$, $e' \mapsto^* [v'_1, \ldots, v'_n]$ and $v_i \sim v'_i : \tau$ for all $1 \leq i \leq n$.

The example(s) are easier to understand if we isolate the special case R **list** for an admissible relation $R : \tau \rightarrow \tau'$ which is actually a function. In this case we obtain

$e \sim e' : R$ **list** for an admissible $R : \tau \rightarrow \tau'$ iff $(\mathit{map}\ R)\,e \simeq e'$.

Here, *map* : $(\tau \rightarrow \tau') \rightarrow (\tau$ **list** $\rightarrow \tau'$ **list**$)$ is the usual mapping function with

$$(\mathit{map}\ R)\,[v_1, \ldots, v_n] \mapsto^* [R\,v_1, \ldots, R\,v_n]$$

Returning to examples, what can the type tell us about a function

$$f : \forall \alpha.\, \alpha\ \textbf{list} \rightarrow \alpha\ \textbf{list}?$$

If the function is parametric, it should not be able to examine the list elements, or create new ones. However, it should be able to drop elements, duplicate elements, or rearrange them. We will try to capture this equationally, just following our nose in using parametricity to see what we end up at.

We start with

$f \sim f : \forall \alpha.\, \alpha$ **list** $\rightarrow \alpha$ **list** *by parametricity.*

Now let $R : \tau \rightarrow \tau'$ be an admissible relation that's actually a function. Then

> $f \sim f : R\,\mathbf{list} \to R\,\mathbf{list}$ *by definition of* $\sim$.

Using the definition of $\sim$ on function types, we obtain

> *For any* $l : \tau\,\mathbf{list}$ *and* $l' : \tau\,\mathbf{list}$ *with* $l\;(R\,\mathbf{list})\;l'$ *we have* $f\,l\;(R\,\mathbf{list})\;f\,l'$

By the remark on the interpretation of R **list** when R is a function, this becomes

> *If* $(\mathit{map}\ R)\,l \simeq l'$ *then* $(\mathit{map}\ R)\,(f\,l) \simeq f\,l'$

or, equivalently,

> $(\mathit{map}\ R)\,(f\,l) \simeq f\,((\mathit{map}\ R)\,l)$.

In short, f commutes with *map* R. This means we can either map R over the list and then apply f to the result, or we can apply f first and then map R over the result. This implies that f could not, say, make up a new element v_0 not in l. Such an element would occur in the list returned by the right-hand side, but would occur as $R\,v_0$ on the left-hand side. So if we have a type with more than one element we can choose R so that $R\,v_0 \neq v_0$ (like a constant function) and the two sides would be different, contradicting the equality we derived.

We can use this equation of improve efficiency of code. For example, if we know that f might reduce the number of elements in the list (for example, skipping every other element), then mapping R over the list after the elements have been eliminated is more efficient than the other way around. Conversely, if f may duplicate some elements then it would be more efficient to map R over the list first and then apply f. The equality we derived from parametricity allows this kind of optimization.

We have, however, to be careful when nonterminating functions may be involved. For example, if R diverges on an element v_0 then the two sides may not be equal. For example, f might drop v_0 from the list l so the right-hand side would diverge while the left-hand side would have a value.

Here are two other similar results provided by Wadler [Wad89].

$$\begin{array}{l} f : \forall\alpha.\,(\alpha\,\mathbf{list})\,\mathbf{list} \to \alpha\,\mathbf{list} \\ (\mathit{map}\ R)\,(f\,l) \simeq f\ ((\mathit{map}\ (\mathit{map}\ R))\ l) \end{array}$$

$$\begin{array}{l} f : \forall\alpha.\,(\alpha \to \mathit{bool}) \to \alpha\,\mathbf{list} \to \alpha\,\mathbf{list} \\ (\mathit{map}\ R)\ (f\,(\lambda x.\,p\,(R\,x))\,l) \simeq f\,p\,((\mathit{map}\,R)\,l) \end{array}$$

These theorems do not quite come "for free", but they are fairly straightforward consequences of parametricity, keeping in mind the requirement of termination.

References

[Har16] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, second edition, April 2016.

[Rey83] John C. Reynolds. Types, abstraction, and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing 83*, pages 513–523. Elsevier, September 1983.

[Wad89] Philip Wadler. Theorem for free! In J. Stoy, editor, *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*, pages 347–359, London, UK, September 1989. ACM.

Lecture Notes on Data Abstraction

15-814: Types and Programming Languages
Frank Pfenning

Lecture 14
October 23, 2018

1 Introduction

Since we have moved from the pure λ-calculus to functional programming languages we have added rich type constructs starting from functions, disjoint sums, eager and lazy pairs, recursive types, and parametric polymorphism. The primary reasons often quoted for such a rich static type system are discovery of errors before the program is ever executed and the efficiency of avoiding tagging of runtime values. There is also the value of the types as documentation and the programming discipline that follows the prescription of types. Perhaps more important than all of these is the strong guarantees of data abstraction that the type system affords that are sadly missing from many other languages. Indeed, this was one of the original motivation in the development of ML (which stands for MetaLanguage) by Milner and his collaborators [GMM^{+}78]. They were interested in developing a theorem prover and wanted to reduce its overall correctness to the correctness of a trusted core. To this end they specified an *abstract type* of *theorem* on which the only allowed operations are inference rules of the underlying logic. The connection between abstract types and existential types was made made Mitchell and Plotkin [MP88].

2 Signatures and Structures

Data abstraction in today's programming languages is usually enforced at the level of modules (if it is enforced at all). As a running example we consider a simple module providing and implementation of numbers with constant *zero* and functions *succ* and *pred*. We will consider two implementations and their relationship. One is using numbers in unary form (type *nat*) and numbers in binary form (type *bin*), and we will eventually prove that they are logically equivalent. We are making up some syntax (loosely based on ML), specify interfaces between a library and its client.

Below we name `NUM` as the *signature* that describes the interface of a module.

```
NUM = {
  type Num
  zero : Num
  succ : Num -> Num
  pred : Num -> Option Num
}
```

The function `pred` returns a `Option Num` since we consider the predecessor of `zero` to be undefined. Recall the option type

```
  data Option a = Null | Just a
```

For the implementations, we use the following types for numbers in unary and binary representation.

```
data Nat = Z | S Nat
data Bin = E | B0 Bin | B1 Bin
```

Then we define the first implementation

```
NAT : NUM = {
  type Num = Nat

  zero = Z

  succ n = S n

  pred Z = Null
  pred (S n) = Just n
}
```

An interesting aspect of this definition is that, for example, `zero : Nat` while the interface specifies `zero : Num`. But this is okay because the type `Num` is in fact implemented by `Nat` in this version. Next, we show the implementation using numbers in binary representation. It is helpful to have a function `map` operating on optional values.

```
map : (a -> b) -> Option a -> Option b
map f Null = Null
map f (Just x) = Just (f x)

BIN : NUM = {
  type Num = Bin

  zero = E

  succ E = B1 E
  succ (B0 x) = B1 x
  succ (B1 x) = B0 (succ x)

  pred E = Null
  pred (B1 x) = Just (B0 x)
  pred (B0 x) = map B1 (pred x)
}
```

Now what does a client look like? Assume it has an implemention `N : NUM`. It can then "open" or "import" this implementation to use its components, but it will not have any knowledge about the type of the implementation. For example, we can write

```
open N : NUM

isZero : Num -> Bool
isZero x = case pred x
              Null => True
              Just y => False
```

but **not**

```
open N : NUM

isZero : Num -> Bool
isZero Z = true          % type error here: Nat not equal
isZero (S n) = false     % and here
```

because the latter supposes that the library `N : NUM` implements the type `Num` by `Nat`, which it may not.

3 Formalizing Abstract Types

We will write a signature such as

```
NUM = {
  type Num
  zero : Num
  succ : Num -> Num
  pred : Num -> Option Num
}
```

in abstract form as

$$\exists\alpha.\ \underbrace{\alpha}_{\textit{zero}} \otimes \underbrace{(\alpha \to \alpha)}_{\textit{succ}} \otimes \underbrace{(\alpha \to \alpha\ \mathbf{option})}_{\textit{pred}}$$

where the name annotations are just explanatory and not part of the syntax. Note that α stands for *Num* which is bound here by the existential quantifier, just as we would expect the scope of `Num` in the signature to only include the three specified components.

Now what should an expression

$$e : \exists\alpha.\, \alpha \otimes (\alpha \to \alpha) \otimes (\alpha \to \alpha\ \mathbf{option})$$

look like? It should provide a concrete type (such as *nat* or *bin*) for α, as well as an implementation of the three functions. We obtain this with the following rule

$$\frac{\Delta \vdash \sigma\ \textit{type} \quad \Delta\ ; \Gamma \vdash e : [\sigma/\alpha]\tau}{\Delta\ ; \Gamma \vdash \langle \sigma, e\rangle : \exists\alpha.\, \tau}\ (\mathbf{I}\text{-}\exists)$$

Besides checking that σ is indeed a type with respect to all the type variables declared in Δ, the crucial aspect of this rule is that the implementation e is at type $[\sigma/\alpha]\tau$.

For example, to check that *zero*, *succ*, and *pred* are well-typed we substitute the implementation type for `Num` (namely `Nat` in one case and `Bin` in the other case) before proceeding with checking the definitions.

The pair $\langle \sigma, e\rangle$ is sometimes referred to as a *package*, which is opened up by the destructor. This destructor is often called **open**, but for uniformity with all analogous cases we'll write is as a **case**.

$$\begin{array}{lrcl} \text{Types} & \tau & ::= & \ldots \mid \exists\alpha.\, \tau \\ \text{Expressions} & e & ::= & \ldots \mid \langle \sigma, e\rangle \mid \mathbf{case}\ e\ \{\langle \alpha, x\rangle \Rightarrow e'\} \end{array}$$

The elimination form provides a new name α for the implementation types and a new variable x for the (eager) pair making up the implementations.

$$\frac{\Delta \,;\, \Gamma \vdash e : \exists \alpha.\, \tau \quad \Delta, \alpha \; \mathit{type} \,;\, \Gamma, x : \tau \vdash e' : \tau'}{\Delta \,;\, \Gamma \vdash \mathbf{case}\; e\; \{\langle \alpha, x\rangle \Rightarrow e'\} : \tau'} \;(\text{E-}\exists)$$

The fact that the type α must be *new* is implicit in the rule in the convention that Δ may not contain an repeated variables. If we happened to have used the name α before then we can just rename it and then apply the rule. It is crucial for data abstraction that this variable α is new because we cannot and should not be able to assume anything about what α might stand for, except the operations that might be exposed in τ and are accessible via the name x. Among other things, α may not appear in τ'.

To be a little more explicit about this (because it is critical here), whenever we write $\Delta \,;\, \Gamma \vdash e : \tau$ we make the following *presuppositions*:

1. All the type variables in Δ are distinct.

2. All the variables in Γ are distinct.

3. $\Delta \vdash \tau_i$ *type* for all $x_i : \tau_i \in \Gamma$.

4. $\Delta \vdash \tau$ *type*.

Whenever we write a rule we assume this presuppositions holds for the conclusion and we have to make sure they hold for all the premises. Let's look at (E-$\exists$) again in this light.

1. We assume all variables in Δ are distinct, which also means they are distinct in the first premise. In the second premise they are distinct because that's how we interpret Δ, α *type*, which may include an implicit renaming of the type variable α bound in the the expression $\langle \alpha, x\rangle \Rightarrow e'$.

2. Similarly for the context Γ, where the freshness of x might be achieved by renaming it before applying the rule.

3. By assumption (from the conclusion), every free type variable in Γ appears in Δ. But what about τ? Strictly speaking, perhaps we should have a premise that Δ, α *type* $\vdash \tau$ *type* but that's usually elided, implied by adding $x : \tau$ to the context Γ.

4. By assumption (from the conclusion), $\Delta \vdash \tau'$ *type*. This covers the second premise. Often, this rule is given with an explicit premise $\Delta \vdash \tau'$ *type* to emphasize τ' must be independent of α. Indeed, the scope of α is the type of x and e'.

We also see that the client e' is *parametric* in α, which means that it cannot depend on what α might actually be at runtime. It is this parametricity that will allow us to swap one implementation out for another without affecting the client as long as the two implementations are equivalent in an appropriate sense.

The operational rules are straightforward and not very interesting.

$$\frac{v\ \mathit{val}}{\langle \sigma, v\rangle\ \mathit{val}}\ (\text{V-}\exists)$$

$$\frac{e \mapsto e'}{\langle \sigma, e\rangle \mapsto \langle \sigma, e'\rangle}\ (\text{CI-}\exists) \qquad \frac{e_0 \mapsto e_0'}{\textbf{case}\ e_0\ \{\langle \alpha, x\rangle \Rightarrow e_1\} \mapsto \textbf{case}\ e_0'\ \{\langle \alpha, x\rangle \Rightarrow e_1\}}\ (\text{C}$$

$$\frac{}{\textbf{case}\ \langle \sigma, v\rangle\ \{\langle \alpha, x\rangle \Rightarrow e\} \mapsto [\sigma/\alpha, v/x]e}\ (\text{R-}\exists)$$

4 Logical Equality for Existential Types

We extend our definition of logical equivalence to handle the case of existential types. Following the previous pattern for parametric polymorphism, we cannot talk about arbitrary instances of the existential type, but we must instantiate it with a relation that is closed under Kleene equality.

Recall from Lecture 12:

($\forall$) $e \sim e' : \forall \alpha.\, \tau$ iff for all closed types σ and σ' and admissible relations $R : \sigma \leftrightarrow \sigma'$ we have $e \sim e' : [R/\alpha]\tau$

(R) $e \sim e' : R$ with $e : \tau$, $e' : \tau'$ and $R : \tau \leftrightarrow \tau'$ iff $e\ R\ e'$.

We add

($\exists$) $e \sim e' : \exists \alpha.\, \tau$ iff $e \simeq \langle \sigma, e_0\rangle$ and $e' \simeq \langle \sigma', e_0'\rangle$ for some closed types σ, σ' and expressions e_0, e_0', and there is an admissible relation $R : \sigma \leftrightarrow \sigma'$ such that $e_0 \sim e_0' : [R/\alpha]\tau$.

In our example, we ask if

$$\texttt{NAT} \sim \texttt{BIN} : \texttt{NUM}$$

which unfolds into demonstrating that there is a relation $R : \mathit{nat} \leftrightarrow \mathit{bin}$ such that

$$\langle Z, \langle S, \mathit{pred}_n \rangle\rangle \sim \langle E, \langle \mathit{succ}_b, \mathit{pred}_b \rangle\rangle : R \otimes (R \to R) \otimes (R \to R\ \mathbf{option})$$

Here we have disambiguated the occurrences of the successor and predecessor function as operating on type *nat* or *bin*.

Since logical equality at type $\tau_1 \otimes \tau_2$ just decomposes into logical equality at the component types, this just decomposes into three properties we need to check. The key step is to define the correct relation R.

5 Defining a Relation Between Implementations

$R : \mathit{nat} \leftrightarrow \mathit{bin}$ needs to relate natural numbers in two different representations. It is convenient and general to define such relations by using inference rules.

Once we have made this decision, the relation could be based on the structure of $n : \mathit{nat}$ or on the structure of $x : \mathit{bin}$. The former may run into difficulties because each number actually corresponds to infinitely many numbers in binary form: just add leading zeros that do not contribute to its value. Therefore, we define it based on the binary representation. In order to define it, we use a function *dbl* on unary numbers.

```
dbl : Nat -> Nat
dbl Z = Z
dbl (S n) = S (S (dbl n))
```

$$\frac{}{Z\ R\ E}\ R_e \qquad \frac{n\ R\ x}{(\mathit{dbl}\ n)\ R\ (B_0\ x)}\ R_0 \qquad \frac{n\ R\ x}{S\ (\mathit{dbl}\ n)\ R\ (B_1\ x)}\ R_1$$

6 Verifying the Relation

Because our signature exposes three constants, we now have to check three properties.

Lemma 1 $Z \sim E : R$

Proof: By definition $Z \sim E : R$ is equivalent to $Z\ R\ E$, which follows immediately from rule R_e. □

Lemma 2 *$S \sim succ_b : R \to R$.*

Proof: By definition of logical equality, this is equivalent to showing

> *For all $n : nat$, $x : bin$ with $n\ R\ x$ we have $(S\ n)\ R\ (succ_b\ x) : R$.*

Since R is defined inductively by a collection of inference rules, the natural attempt is to prove this by rule induction on the given relation, namely $n\ R\ x$.

Case: Rule

$$\frac{}{Z\ R\ E}\ R_e$$

with $n = Z$ and $x = E$. We have to show that $(S\,n)\ R\ (succ\ \ x)$ (abbreviating now $succ_b$ as $succ$).

$$\begin{array}{lr}
Z\ R\ E & \text{By rule } R_e \\
(S\,(dbl\,Z))\ R\ (B_1\ E) & \text{By rule } R_1 \\
(S\,Z)\ R\ (B_1 E) & \text{Since } dbl\,Z \simeq Z \\
(S\,Z)\ R\ (succ\,E) & \text{Since } succ\,E \simeq B_1\,Z \\
(S\,n)\ R\ (succ\,x) & \text{Since } n = Z \text{ and } x = E
\end{array}$$

This proof is most likely discovered and should perhaps be read starting with the last line and going upwards.

Case: Rule

$$\frac{n'\ R\ x'}{(dbl\ n')\ R\ (B_0\ x')}\ R_0$$

where $n = dbl\ n'$ and $x = B_0\ x'$. We have to show that $(S\,n)\ R\ (succ\ x)$. Again, you may want to read the proof below starting at the bottom.

$$\begin{array}{lr}
n'\ R\ x' & \text{Premise in this case} \\
(S\,(dbl\ n'))\ R\ (B_1\ x') & \text{By rule } R_1 \\
(S\,(dbl\ n'))\ R\ (succ\ (B_0\ x')) & \text{Since } succ\ (B_0\ x') \simeq B_1\ x' \\
(S\,n)\ R\ (succ\ x) & \text{Since } n = dbl\ n' \text{ and } x = B_0\ x'
\end{array}$$

Case: Rule

$$\frac{n'\ R\ x'}{S\ (dbl\ n')\ R\ (B_1\ x')}\ R_1$$

where $n = S\ (dbl\ n')$ and $x = B_1\ x'$. We have to show that $(S\,n)\ R\ (succ\ x)$. Again, you may want to read the proof below starting at the bottom.

$n' \; R \; x'$	Premise in this case
$(S \; n') \; R \; (\mathit{succ} \; x')$	By induction hypothesis
$(\mathit{dbl} \; (S \; n')) \; R \; (B_0 \; (\mathit{succ} \; x'))$	By rule R_0
$(S \, (S \, (\mathit{dbl} \; n'))) \; R \; (B_0 \; (\mathit{succ} \; x'))$	Since $\mathit{dbl} \; (S \; n') \simeq S \; (S \; (\mathit{dbl} \; n'))$
$(S \, (S \, (\mathit{dbl} \; n'))) \; R \; (\mathit{succ} \; (B_1 \; x'))$	Since $\mathit{succ} \; (B_1 \; x') \simeq B_0 \; (\mathit{succ} \; x')$
$(S \, n) \; R \; (\mathit{succ} \; x)$	Since $n = S \, (\mathit{dbl} \; n')$ and $x = B_1 \; x'$

□

In order to prove the relation between the implementation of the predecessor function we should explicitly write out the interpretation of τ **option**.

(τ option) $e \sim e' : \tau$ **option** iff either $e \simeq$ **null** and $e' \simeq$ **null** or $e \simeq$ **just** e_1 and $e' \simeq$ **just** e'_1 and $e_1 \sim e'_1 : \tau$.

Lemma 3 $\mathit{pred}_n \simeq \mathit{pred}_b : R \to R$ **option**

Proof: By definition of logical equality, this is equivalent to show

> *For all $n : \mathit{nat}$, $x : \mathit{bin}$ with $n \; R \; x$ we have either (i) $\mathit{pred}_n \; n \simeq$* **null** *and $\mathit{pred}_b \; x \simeq$* **null** *or (ii) $\mathit{pred}_n \; n \simeq$* **just** *$n'$ and $\mathit{pred}_b \; x \simeq$* **just** *$x'$ and $n' \; R \; x'$.*

This can now be proven by rule induction on the given relation, with a slightly more complicated argument.

Case: Rule

$$\frac{}{Z \; R \; E} \; R_e$$

with $n = Z$ and $x = E$. Then $\mathit{pred}_n \; Z =$ **null** $= \mathit{pred}_b \; E$.

Case: Rule

$$\frac{n' \; R \; x'}{(\mathit{dbl} \; n') \; R \; (B_0 \; x')} \; R_0$$

where $n = \mathit{dbl} \; n'$ and $x = B_0 \; x'$.

$n' \; R \; x'$ — Premise in this case

Either $\mathit{pred}_n \; n' =$ **null** $= \mathit{pred}_b \; x'$
or $\mathit{pred}_n \; n' =$ **just** n'' and $\mathit{pred}_b \; x' =$ **just** x'' with $n'' \; R \; x''$
By induction hypothesis

$\mathit{pred}_n \; n' =$ **null** $= \mathit{pred}_b \; x'$ — First subcase

$n' = Z$ By inversion on the defn. of pred_n

$\mathit{pred}_n\ (\mathit{dbl}\ n') = \mathit{pred}_n\ Z = \mathbf{null}$ By definition of pred_n

$\mathit{pred}_b\ x = \mathit{pred}_b\ (B_0\ x') = \mathit{map}\ B_1\ (\mathit{pred}_b\ x')$

$\quad = \mathit{map}\ B_1\ \mathbf{null} = \mathbf{null}$ By definition of pred_b

$\mathit{pred}_n\ n' = \mathbf{just}\ n''$ and $\mathit{pred}_b\ x' = \mathbf{just}\ x''$ and $n''\ R\ x''$

Second subcase

$n' = S\ n''$ By inversion on the definition of pred_n

$\mathit{pred}_n\ (\mathit{dbl}\ n') = \mathit{pred}_n\ (S\ (S\ (\mathit{dbl}\ n'')))$

$\quad = \mathbf{just}\ (S\ (\mathit{dbl}\ n''))$ By definition of pred_n

$\mathit{pred}_b\ (B_0\ x') = \mathit{map}\ B_1\ (\mathit{pred}_b\ x')$

$\quad = \mathit{map}\ B_1\ (\mathbf{just}\ x'') = \mathbf{just}\ (B_1\ x'')$ By definition of pred_b

$(S\ (\mathit{dbl}\ n''))\ R\ (B_1\ x'')$ By rule R_1

Case: Rule

$$\frac{n'\ R\ x'}{S\ (\mathit{dbl}\ n')\ R\ (B_1\ x')}\ R_1$$

where $n = S\ (\mathit{dbl}\ n')$ and $x = B_1\ x'$.

$\mathit{pred}_n\ n = \mathit{pred}_n\ (S\ (\mathit{dbl}\ n')) = \mathbf{just}\ (\mathit{dbl}\ n')$ By defn. of pred_n

$\mathit{pred}_b\ x = \mathit{pred}_b\ (B_1\ x') = \mathbf{just}\ (B_0\ x')$ By defn. of pred_b

$(\mathit{dbl}\ n')\ R\ (B_0\ x')$ By rule R_0

□

7 The Upshot

Because the two implementations are logically equal we can replace one implementation by the other without changing any client's behavior. This is because all clients are parametric, so their behavior does not depend on the library's implementation.

It may seem strange that this is possible because we have picked a particular relation to make this proof work. Let us reexamine the (E-∃) rule:

$$\frac{\Delta\ ;\ \Gamma \vdash e : \exists\alpha.\,\tau \quad \Delta, \alpha\ \mathit{type}\ ;\ \Gamma, x : \tau \vdash e' : \tau'}{\Delta\ ;\ \Gamma \vdash \mathbf{case}\ e\ \{\langle\alpha, x\rangle \Rightarrow e'\} : \tau'}\ (\text{E-}\exists)$$

In the second premise we see that the client e' is checked with a fresh type α and $x : \tau$ which may mention α. if we reify this into a function, we find

$$\lambda x.\, e' : \forall\alpha.\, \tau \to \tau'$$

where τ' does not depend on α.

By Reynolds's parametricity theorem we know that this function is parametric. This can now be applied for any σ and σ' and relation $R : \sigma \leftrightarrow \sigma'$ to conclude that if $v_0 \sim v_0' : [R/\alpha]\tau$ then $[v_0/x]e' \sim [v_0'/x]e' : [R/\alpha]\tau'$. But α does not occur in τ', so this is just saying that $[v_0/x]e' \sim [v_0'/x]e' : \tau'$. So the result of substituting the two different implementations is equivalent.

References

[GMM+78] Michael J.C. Gordon, Robin Milner, L. Morris, Malcolm C. Newey, and Christopher P. Wadsworth. A metalanguage for interactive proof in LCF. In A. Aho, S. Zillen, and T. Szymanski, editors, *Conference Record of the 5th Annual Symposium on Principles of Programming Languages (POPL'78)*, pages 119–130, Tucson, Arizona, January 1978. ACM Press.

[MP88] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

Lecture Notes on
The K Machine

15-814: Types and Programming Languages
Frank Pfenning

Lecture 15
October 25, 2018

1 Introduction

After examining an exceedingly pure, but universal notion of computation in the λ-calculus, we have been building up an increasingly expressive language including recursive types, universal types (parametric polymorphism), and existential types (abstract types). The standard theorems to validate the statics and dynamics are progress and preservation, relying also on canonical forms. The pinnacle of this development is Reynolds's *parametricity theorem* that ensures data abstraction for implementations of libraries using existential types. We have also seen that the supposed opposition of dynamic and static typing is instead just a reflection of breadth of properties we would like to enforce statically, and the supposed opposition of eager (strict) and lazy constructors is just a question of which types we choose to include in our language.

At this point we turn our attention to defining the dynamics of the constructs at a lower level of abstraction that we have done so far. This introduces some complexity in what we call "dynamic artifacts", that is, objects beyond the source expressions that help us describe how programs execute. In this lecture, we show the K machine in which a *stack* is made explicit. This stack can also be seen as a *continuation*, capturing everything that remains to be done after the current expression has been evaluated. At the end of the lecture we show an elegant high-level implementation of the K machine in Haskell.

2 Introducing the K Machine

Let's review the dynamics of functions.

$$\frac{}{\lambda x.\, e\ \mathit{val}}\ (\text{V-}{\to})$$

$$\frac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2}\ (\text{CE-}{\to}_1) \qquad \frac{v_1\ \mathit{val} \quad e_2 \mapsto e_2'}{v_1\, e_2 \mapsto v_1\, e_2'}\ (\text{CE-}{\to}_2)$$

$$\frac{}{(\lambda x.\, e_1')\, v_2 \mapsto [v_2/x]e_1'}\ (\text{R-}{\to})$$

The rule (CE-$\to_1$) and (CE-$\to_2$) are *congruence rules*: they descend into an expression e in order to find a *redex*, $(\lambda x.\, e_1')\, v_2$ in this case. The reduction rule (R-$\to$) is the "actual" computation, which takes place when a *constructor* (here: λ-abstraction) is met by a *destructor* (here: application).

The rules for all other forms of expression follow the same pattern. The definition of a value of the given type guides which congruence rules are required. Overall, the preservation and progress theorems verify that a particular set of rules for a type constructor was defined coherently.

In a multistep computation

$$e_0 \mapsto e_1 \mapsto e_2 \mapsto \cdots \mapsto e_n = v$$

each expression e_i represents *the whole program* and v its final value. This makes the dynamics economical: only expressions are required when defining it. But a straightforward implementation would have to test whether expressions are values, and also *find* the place where the next reduction should take place by traversing the expression using congruence rules.

It would be a little bit closer to an implementation if we could keep track where in a large program we currently compute. The key idea needed to make this work is to also remember *what we still have to do after we are done evaluating the current expression*. This is the role of a *continuation* (read: *"how we continue after this"*). In the particular abstract machine we present, the continuation is organized as a stack, which appears to be a natural data structure to represent the continuation.

The machine has two different forms of states

$k \rhd e$ evaluate e with continuation k
$k \lhd v$ return value v to continuation k

In the second form, we will always have v *val*. We call this an *invariant* or *presupposition* and we have to verify that all transition rules of the abstract machine preserve this invariant.

As for continuations, we'll have to see what we need as we develop the dynamics of the machine. For now, we only know that we will need an *initial continuation* or *empty stack*, written as ϵ.

$$\text{Continuations} \quad k \quad ::= \quad \epsilon \mid \ldots$$

In order to evaluate an expression, we start the machine with

$$\epsilon \rhd e$$

and we expect that it transitions to a final state

$$\epsilon \lhd v$$

if and only if $e \mapsto^* v$. Actually, we can immediately generalize this: no matter what the continuation k, we want evaluation of e return the value of e to k:

> *For any continuation k, expression e and value v,*
> $k \triangleright e \mapsto^* k \triangleleft v$ *iff* $e \mapsto^* v$

We should keep this in mind as we are developing the rules for the K machine.

3 Evaluating Functions

Just as for the usual dynamics, the transitions of the machine are organized by type. We begin with functions. An expression $\lambda x.\, e$ is a value. Therefore, it is immediately returned to the continuation.

$$k \triangleright \lambda x.\, e \quad \mapsto \quad k \triangleleft \lambda x.\, e$$

It is immediate that the theorem we have in mind about the machine is satisfied by this transition.

How do we evaluate an application $e_1\, e_2$? We start by evaluating e_1 until it is a value, then we evaluate e_2, and then we perform a β-reduction. When we evaluate e_1 we have to remember what remains to be done. We do this with the continuation

$$(\_\; e_2)$$

which has a blank in place of the expression that is currently being evaluated. We push this onto the stack, because once this continuation has done its work, we still need to do whatever remains after that.

$$k \triangleright e_1\, e_2 \quad \mapsto \quad k \circ (\_\, e_2) \triangleright e_1$$

When the evaluation of e_1 returns a value v_1 to the continuation $k \circ (\_\, e_2)$ we evaluate e_2 next, remembering we have to pass the result to v_1.

$$k \circ (\_\, e_2) \triangleleft v_1 \quad \mapsto \quad k \circ (v_1\, \_) \triangleright e_2$$

Finally, when the value v_2 of e_2 is returned to this continuation we can carry out the β-reduction, substituting v_2 for the formal parameter x in the body e_1' of the function. The result is an expression that we then proceed to evaluate.

$$k \circ ((\lambda x.\, e_1')\, \_) \triangleleft v_2 \quad \mapsto \quad k \triangleright [v_2/x]e_1'$$

The continuation for $[v_2/x]e_1'$ is the original continuation of the application, because the ultimate value of the application is the ultimate value of $[v_2/x]e_1'$.

Summarizing the rules pertaining to functions:

$$\begin{array}{rclcrcl}
k & \triangleright & \lambda x.\, e & \mapsto & k & \triangleleft & \lambda x.\, e \\
k & \triangleright & e_1\, e_2 & \mapsto & k \circ (\_\, e_2) & \triangleright & e_1 \\
k \circ (\_\, e_2) & \triangleleft & v_1 & \mapsto & k \circ (v_1\, \_) & \triangleright & e_2 \\
k \circ ((\lambda x.\, e_1')\, \_) & \triangleleft & v_2 & \mapsto & k & \triangleright & [v_2/x]e_1'
\end{array}$$

And the continuations required:

$$\begin{array}{lrcl}
\text{Continuations} & k & ::= & \epsilon \\
 & & \mid & k \circ (\_\, e_2) \mid k \circ (v_1\, \_)
\end{array}$$

4 A Small Example

Let's run the machine through a small example,

$$((\lambda x.\, \lambda y.\, x)\, v_1)\, v_2$$

for some arbitrary values v_1 and v_2.

$$\begin{array}{lrcl}
 & \epsilon & \rhd & ((\lambda x.\, \lambda y.\, x)\, v_1)\, v_2 \\
\mapsto & \epsilon \circ (\_\; v_2) & \rhd & (\lambda x.\, \lambda y.\, x)\, v_1 \\
\mapsto & \epsilon \circ (\_\; v_2) \circ (\_\; v_1) & \rhd & \lambda x.\, \lambda y.\, x \\
\mapsto & \epsilon \circ (\_\; v_2) \circ (\_\; v_1) & \lhd & \lambda x.\, \lambda y.\, x \\
\mapsto & \epsilon \circ (\_\; v_2) \circ ((\lambda x.\, \lambda y.\, x)\; \_) & \rhd & v_1 \\
\mapsto^* & \epsilon \circ (\_\; v_2) \circ ((\lambda x.\, \lambda y.\, x)\; \_) & \lhd & v_1 \\
\mapsto & \epsilon \circ (\_\; v_2) & \rhd & \lambda y.\, v_1 \\
\mapsto & \epsilon \circ (\_\; v_2) & \lhd & \lambda y.\, v_1 \\
\mapsto & \epsilon \circ ((\lambda y.\, v_1)\; \_) & \rhd & v_2 \\
\mapsto^* & \epsilon \circ ((\lambda y.\, v_1)\; \_) & \lhd & v_2 \\
\mapsto & \epsilon & \rhd & v_1 \\
\mapsto^* & \epsilon & \lhd & v_1
\end{array}$$

If v_1 and v_2 are functions, then the multistep transitions based on our desired correctness theorem are just a single step each.

We can see that the steps are quite small, but that the machine works as expected. We also see that some *values* (such as v_1) appear to be evaluated more than once. A further improvement of the machine would be to mark values so that they are not evaluated again.

5 Eager Pairs

Functions are lazy in the sense that the body of a λ-abstraction is not evaluated, even in a call-by-value language. As another example we consider eager pairs $\tau_1 \otimes \tau_2$. Recall the rules:

$$\frac{v_1\ \mathit{val} \quad v_2\ \mathit{val}}{\langle v_1, v_2 \rangle\ \mathit{val}}\ (\text{V-}\otimes)$$

$$\frac{e_1 \mapsto e_1'}{\langle e_1, e_2 \rangle \mapsto \langle e_1', e_2 \rangle}\ (\text{CI-}\otimes_1) \qquad \frac{v_1\ \mathit{val} \quad e_2 \mapsto e_2'}{\langle v_1, e_2 \rangle \mapsto \langle v_1, e_2' \rangle}\ (\text{CI-}\otimes_2)$$

$$\frac{e_0 \mapsto e_0'}{\textbf{case}\ e_0\ \{\langle x_1, x_2 \rangle \Rightarrow e\} \mapsto \textbf{case}\ e_0'\ \{\langle x_1, x_2 \rangle \Rightarrow e\}}\ (\text{CE-}\otimes)$$

$$\frac{v_1\ \mathit{val} \quad v_2\ \mathit{val}}{\textbf{case}\ \langle v_1, v_2 \rangle\ \{\langle x_1, x_2 \rangle \Rightarrow e\} \mapsto [v_1/x_1, v_2/x_2]e}\ (\text{R-}\otimes)$$

We develop the rules in a similar way. Evaluation of a pair begins by evaluating the first component.

$$k \triangleright \langle e_1, e_2 \rangle \quad \mapsto \quad k \circ \langle \_, e_2 \rangle \triangleright e_1$$

When the value is returned, we start with the second component.

$$k \circ \langle \_, e_2 \rangle \triangleleft v_1 \quad \mapsto \quad k \circ \langle v_1, \_ \rangle \triangleright e_2$$

When the second value is returned, we can immediately form the pair (a new value) and return it to the continuation further up the stack.

$$k \circ \langle v_1, \_ \rangle \triangleleft v_2 \quad \mapsto \quad k \triangleleft \langle v_1, v_2 \rangle$$

For a **case** expression, we need to evaluate the subject of the case.

$$k \triangleright \textbf{case}\ e_0\ \{\langle x_1, x_2 \rangle \Rightarrow e\} \quad \mapsto \quad k \circ \textbf{case}\ \_\ \{\langle x_1, x_2 \rangle \Rightarrow e\} \triangleright e_0$$

When e_0 has been evaluated, a pair should be returned to this continuation, and we can carry out the reduction and continue with evaluating e after substitution.

$$k \circ \textbf{case}\ \_\ \{\langle x_1, x_2 \rangle \Rightarrow e\} \triangleleft \langle v_1, v_2 \rangle \quad \mapsto \quad k \triangleright [v_1/x_1, v_2/x_2]e$$

To summarize:

$$\begin{array}{rcl}
k \triangleright \langle e_1, e_2 \rangle & \mapsto & k \circ \langle \_, e_2 \rangle \triangleright e_1 \\
k \circ \langle \_, e_2 \rangle \triangleleft v_1 & \mapsto & k \circ \langle v_1, \_ \rangle \triangleright e_2 \\
k \circ \langle v_1, \_ \rangle \triangleleft v_2 & \mapsto & k \triangleleft \langle v_1, v_2 \rangle \\
k \triangleright \textbf{case}\ e_0\ \{\langle x_1, x_2 \rangle \Rightarrow e\} & \mapsto & k \circ \textbf{case}\ \_\ \{\langle x_1, x_2 \rangle \Rightarrow e\} \triangleright e_0 \\
k \circ \textbf{case}\ \_\ \{\langle x_1, x_2 \rangle \Rightarrow e\} \triangleleft \langle v_1, v_2 \rangle & \mapsto & k \triangleright [v_1/x_1, v_2/x_2]e
\end{array}$$

$$\begin{array}{lcll}
\text{Continuations} \quad k & ::= & \epsilon & \\
& \mid & k \circ (\_\ e_2) \mid k \circ (v_1\ \_) & (- \\
& \mid & k \circ \langle \_, e_2 \rangle \mid k \circ \langle v_1, \_ \rangle \mid k \circ \textbf{case}\ \_\ \{\langle x_1, x_2 \rangle \Rightarrow e\} & (\otimes
\end{array}$$

6 Correctness of the K Machine

Given the relatively simple construction of the machine it is surprisingly tricky to prove its correctness. We refer to the textbook [Har16, Chapter 28] for a complete formal development. We already cited the key property

> *For any continuation k, expression e and value v,*
> $k \triangleright e \mapsto^* k \triangleleft v$ *iff* $e \mapsto^* v$

This implies that $k \triangleright v \mapsto^* k \triangleleft v$ because $v \mapsto^* v$.

A key step in the proof is to find a relation between expressions and machine states $k \triangleright e$ and $k \triangleleft v$. In this case we actually define this relation as a function that *unravels* the state back into an expression. As stated in the property above, the state $k \triangleright e$ expects the value of e being passed to k. When we unravel the state we don't wait for evaluation finish, but we just substitute expression e back into k. Consider, for example,

$$k \triangleright e_1\, e_2 \quad \mapsto \quad k \circ (\_\ e_2) \triangleright e_1$$

If we plug e_1 into the hole of the continuation $(\_\ e_2)$ we recover $e_1\, e_2$, which we can then pass to k.

We write $k \bowtie e = e'$ for the operation of reconstituting an expression from the state $k \triangleright e$ or $k \triangleleft e$ (ignoring the additional information that e is a value in the second case). We define this inductively over the structure of k. First, when the stack is empty we just take the expression.

$$\epsilon \bowtie e = e$$

Otherwise, we plug the expression into the frame on top of the stack (which is the rightmost part of the continuation), and then recursively plug the result into the remaining contintuation.

$$\begin{array}{lcl}
\epsilon \bowtie e & = & e \\
k \circ (\_\ e_2) \bowtie e_1 & = & k \bowtie e_1\, e_2 \\
k \circ (v_1\ \_) \bowtie e_2 & = & k \bowtie v_1\, e_2 \\
k \circ \langle \_, e_2 \rangle \bowtie e_1 & = & k \bowtie \langle e_1, e_2 \rangle \\
k \circ \langle v_1, \_ \rangle \bowtie e_2 & = & k \bowtie \langle v_1, e_2 \rangle \\
k \circ \textbf{case}\ \_\ \{\langle x_1, x_2 \rangle \Rightarrow e\} \bowtie e_0 & = & k \circ \textbf{case}\ e_0\ \{\langle x_1, x_2 \rangle \Rightarrow e\}
\end{array}$$

We now observe that the rules of the K machine that decompose an expression leave the unravelling of a state unchanged.

We write $e\ R\ s$ if $e = k \bowtie f$ when $s = k \triangleright f$ or $e = k \bowtie v$ when $s = k \triangleleft v$. This relation R between is[1] a *bisimulation* in the sense that

(i) If $e \mapsto e'$ and $e\ R\ s$ then there exists an s' with $e'\ R\ s'$ and $s \mapsto^* s'$.

(ii) If $s \mapsto s'$ and $s\ R\ e$ then there exists an e' with $s'\ R\ e'$ and $e \mapsto^* e'$.

This form of relationship is often displayed in pictorial form, where solid lines denote given relationship and dashed lines denote relationship whose

[1] we conjecture, but have not proved

existence is to be proved. In this case we might display the two properties as

$$\begin{tikzcd} e \arrow[r, dash, "R"] \arrow[d, mapsto] & s \arrow[d, dashed, mapsto, "*"] \\ e' \arrow[r, dash, dashed, "R"'] & s' \end{tikzcd} \quad \text{and} \quad \begin{tikzcd} e \arrow[r, dash, "R"] \arrow[d, dashed, mapsto, "*"'] & s \arrow[d, mapsto] \\ e' \arrow[r, dash, dashed, "R"'] & s' \end{tikzcd}$$

These are generic pictures for relation R to be a *weak bisimulation*, where "weak" indicates that the side simulating a one-step transition may take many steps (including none at all).

7 Typing the K Machine

In general, it is informative to maintain static typing to the extent possible when we transform the dynamics. If there is a new language involved we might say we have a *typed intermediate language*, but even if in the case of the K machine where we still evaluate expressions and just add continuations, we still want to maintain typing.

We type a continuation as *receiving* a value of type τ and eventually producing the final answer for the whole program of type σ. That is, $k \div \tau \Rightarrow \sigma$. Continuations are always closed, so there is no context Γ of free variables. We use a different symbol $\div$ for typing and $\Rightarrow$ for the functional interpretation of the continuation so there is no confusion with the usual notation.

The easiest case is

$$\frac{}{\epsilon \div \tau \Rightarrow \tau}$$

since the empty continuation ϵ immediately produces the value that it is passed as the final value of the computation.

We consider $k \circ (\_\ e_2)$ in some detail. This is a continuation that takes a value of type $\tau_2 \to \tau_1$ and applies it to an expression $e_2 : \tau_2$. The resulting value is passed to the remaining continuation k. The final answer type of $k \circ (\_\ e_2)$ and k are the same σ. Writing this out in the form of an inference rule:

$$\frac{k \div \tau_1 \Rightarrow \sigma \qquad \cdot \vdash e_2 : \tau_2}{k \circ (\_\ e_2) \div (\tau_2 \to \tau_1) \Rightarrow \sigma}$$

The order in which we develop this rule is important: when designing or recalling such rules yourself we strongly recommend you fill in the various judgments and types incrementally, as we did in lecture.

The other function-related continuations follows a similar pattern. We arrive at

$$\frac{k \div \tau_1 \Rightarrow \sigma \quad \cdot \vdash v_1 : \tau_2 \to \tau_1 \quad v_1 \ \mathit{val}}{k \circ (v_1 \ \_) \div \tau_2 \Rightarrow \sigma}$$

Pairs follow a similar pattern and we just show the rules.

$$\frac{k \div (\tau_1 \otimes \tau_2) \Rightarrow \sigma \quad \cdot \vdash e_2 : \tau_2}{k \circ \langle \_, e_2 \rangle \div \tau_1 \Rightarrow \sigma} \qquad \frac{k \div (\tau_1 \otimes \tau_2) \Rightarrow \sigma \quad \cdot \vdash v_1 : \tau_1 \quad v_1 \ \mathit{val}}{k \circ \langle v_1, \_ \rangle \div \tau_2 \Rightarrow \sigma}$$

$$\frac{k \div \tau' \Rightarrow \sigma \quad x_1 : \tau_1, x_2 : \tau_2 \vdash e' : \tau'}{k \circ \mathsf{case} \ \_ \ \{\langle x_1, x_2 \rangle \Rightarrow e'\} \div (\tau_1 \otimes \tau_2) \Rightarrow \sigma}$$

With these rules, we can state preservation and progress theorems for the K machine, but their formulation and proof entirely follow previous developments so we elide them here.

8 Implementing the K Machine

The K machine can be extended to encompass all the type constructors we have introduced so far. Both statics and dynamics (almost) write themselves, following the same ideas we have presented in this lecture. During lecture, we also live-coded an elegant implementation of the K-machine, adding the unit type 1 for good measure.

The first question is how to implement the source expressions. We use a deep embedding in the sense that both constructors and destructors of each type have an explicit representation. But we nevertheless use functions in the metalanguage to represent bound variables together with their scope in the object language, a technique called *higher-order abstract syntax*. In the textbook, at the level of mathematical discourse, expressions with bindings are represented as *abstract binding trees*.

In Haskell, we write

```
data E = Lam (E -> E)
       | App E E
       | Pair E E
       | CasePair E (E -> E -> E)
       | Unit
       | CaseUnit E E
```

Note that λ-abstraction binds one variable and the case construct over pairs binds two.

The second question is how we represent the continuation stack. The idea suggested by the analysis in the previous section is that the continuation stack itself might be represented as a function. We represent $k \triangleright e$ by *eval* e k and $k \triangleleft v$ by *retn* v k. Writing the continuation as a second argument aids in the readability of the code.

```
eval :: E -> (E -> E) -> E
retn :: E -> (E -> E) -> E
```

Now we transcribe the rules. For example,

$$k \triangleright \lambda x.\, e \quad \mapsto \quad k \triangleleft \lambda x.\, e$$

Since a λ-expression is a value, evaluating it immediately returns it to the continuation. This becomes

```
eval (Lam f) k = retn (Lam f) k
```

Also, returning a value to a continuation simply applies the continuation (which is a function) to the value.

```
retn v k = k v
```

Application $e_1\, e_2$ is a bit more complicated. First, we evaluate e_1, returning its value to the continuation.

```
eval (App e1 e2) k = eval e1 (\v1 -> ...)
```

The continuation (here `...`) that expects v_1 has to evaluate e_2 next and pass *its* value to a further continuation.

```
eval (App e1 e2) k = eval e1 (\v1 -> eval e2 (\v2 -> ...))
```

Now we have to perform the actual reduction, substituting v_2 in the body of the λ-expression that is v_1. In order to be able to write that, we pattern-match against a λ-value when we receive v_1.

```
eval (App e1 e2) k = eval e1 (\(Lam f) -> eval e2 (\v2 ->
```

Since the constructor `Lam :: (E -> E) -> E`, we see that `f :: E -> E`. Applying `f` to `e2` will effectively substitute `e2` into the body of `f`.

```
eval (App e1 e2) k =
    eval e1 (\(Lam f) -> eval e2 (\v2 -> ... (f v2) ...))
```

That will result in an expression representing $[v_2/x]e_1'$, which we need to evaluate further.

```
eval (App e1 e2) k =
    eval e1 (\(Lam f) -> eval e2 (\v2 -> eval (f v2) ...)
```

Finally, we have to pass the original continuation to this evaluation.

```
eval (App e1 e2) k =
    eval e1 (\(Lam f) -> eval e2 (\v2 -> eval (f v2) k))
```

The remaining cases in evaluation are derived from the transition rules of the abstract machine in a similar manner. We do not make continuations or stacks explicit as a data structure, but represent them as functions. We show the completed code.

```
data E = Lam (E -> E)
       | App E E
       | Pair E E
       | CasePair E (E -> E -> E)
       | Unit
       | CaseUnit E E

eval :: E -> (E -> E) -> E
retn :: E -> (E -> E) -> E

eval (Lam f) k = retn (Lam f) k
eval (App e1 e2) k = eval e1 (\(Lam f) ->
                         eval e2 (\v2 -> eval (f v2) k))
eval (Pair e1 e2) k = eval e1 (\v1 ->
                          eval e2 (\v2 -> retn (Pair v1 v2) k))
eval (CasePair e f) k = eval e (\(Pair v1 v2) -> eval (f v1 v2)
eval (Unit) k = retn (Unit) k
eval (CaseUnit e f) k = eval e (\(Unit) -> eval f k)

retn v k = k v
```

This interpreter can fail with an error because we have not implemented a type-checker. Such as error could arise because pattern-matching against `(Lam f)`, `(Pair v1 v2)`, and `(Unit)` in the cases for `App`, `CasePair`, and `CaseUnit` may fail to match the value returned *if the expression is not well-typed*. Writing a type-checker on this representation is a bit tricky, and we might discuss it at a future lecture.

A more complete implementation, including fixed points, recursive types, and sums can be found on the course schedule page.

This form of continuation-passing interpreter has been proposed by Reynolds [Rey72] as a means of language definition. The K machine can be seen as a "defunctionalization" of such a higher-order interpreter.

References

[Har16] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, second edition, April 2016.

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, pages 717–740, Boston, Massachusetts, August 1972. ACM Press. Reprinted in *Higher-Order and Symbolic Computation*, 11(4), pp.363–397, 1998.

Lecture Notes on Modeling Store

15-814: Types and Programming Languages
Frank Pfenning

Lecture 16
October 30, 2018

1 Introduction

The dynamics we have constructed so far treat both expressions and values as abstract terms, while in an actual machine architecture both expressions and values need to be stored in memory. In this lecture we introduce a store, arriving at the *S machine*. The idea is for the store to hold *values*. We leave *expressions* as terms with binders that we interpret directly. In the next lecture we'll look at expressions in (slightly) more detail.

We present the dynamics with store in the form of a *substructural operational semantics* [Pfe04, PS09, Sim12]. In this form of presentation the state is a collection of semantic objects which are rewritten following transition rules describing the semantics. We *can* think of them as inference rules, but unlike our original dynamics they would not have any premises.

2 Semantic Objects in the S Machine

At the heart of the S machine are *destinations* d (also called *addresses*) to hold values in the store. The only operation on them is to generate fresh ones—in a low-level implementation a system function such as `malloc` may be called. We assume the memory at a destination is not initialized until is written to.

The state of the S machine consists of the following objects:

eval e d**.** Evaluate expression e, storing the result in destination d. The destination d here is an address in the store which we assume has been allocated with enough memory to hold the value of e.

$\mathsf{!cell}\ d\ c$. Cell d has contents c. Because a value (such as a list) may be large, each cell contains only part of the value, and we use c to describe what (small) data may be stored in a cell. The exclamation mark '!' indicates that cells are *persistent*, which means the value of a cell can never change and will be available during the whole remainder of the computation.

$\mathsf{cont}\ d\ k\ d'$. Continuation k receives a value in destination in d and puts result into d'.

As before, we will develop the semantics incrementally to see what cells might contain, and which continuations we might need.

2.1 Unit

Evaluating the unit element immediately just stores it in memory at the given destination. We write:

$$\mathsf{eval}\ \langle\,\rangle\ d \quad\mapsto\quad \mathsf{!cell}\ d\ \langle\,\rangle$$

The whole state of the S machine is a whole collection of objects, but we leave them implicit here because every rule is intended to apply to a subset of the objects, replacing those matching the left-hand side of the rule by the right-hand side. More explicit would be

$$S, \mathsf{eval}\ \langle\,\rangle\ d \quad\mapsto\quad S, \mathsf{!cell}\ d\ \langle\,\rangle$$

Second, if we have a case over a value of unit element we begin by evaluating the subject of the case, and remember in the continuation that we are waiting on this value.

$$\mathsf{eval}\ (\mathbf{case}\ e\ \{\langle\,\rangle \Rightarrow e'\})\ d' \quad\mapsto\quad \mathsf{eval}\ e\ d, \mathsf{cont}\ d\ (\mathbf{case}\ \_\ \{\langle\,\rangle \Rightarrow e'\})\ d' \quad (d\ \text{fresh})$$

Let's read this. We create a fresh destination d to hold the value of e. The object $\mathsf{cont}\ d\ (\mathsf{eval}\ \_\ \{\langle\,\rangle \Rightarrow e'\})\ d'$ waits on the destination d before proceeding. Once the cell d holds a value (which must be $\langle\,\rangle$), the continuation must evaluate e' with destination d'.

$$\mathsf{!cell}\ d\ \langle\,\rangle, \mathsf{cont}\ d\ (\mathbf{case}\ \_\ \{\langle\,\rangle \Rightarrow e'\})\ d' \quad\mapsto\quad \mathsf{eval}\ e'\ d'$$

In this rule, the *persistent* !cell object on the left-hand side remains in the store, even though it is not explicitly mentioned on the right-hand side. The continuation on the other hand is *ephemeral*, that is, it is consumed in the application of the rule and replaced by the eval object on the right-hand side.

As a simple example, consider the evaluation of $\mathbf{case}\ \langle\,\rangle\ \{\langle\,\rangle \Rightarrow \langle\,\rangle\}$ with some initial destination d_0, showing the whole state each time.

$$\begin{array}{ll} & \mathsf{eval}\ (\mathbf{case}\ \langle\,\rangle\ \{\langle\,\rangle \Rightarrow \langle\,\rangle\})\ d_0 \\ \mapsto & \mathsf{eval}\ \langle\,\rangle\ d_1, \mathsf{cont}\ d_1\ (\mathbf{case}\ \_\ \{\langle\,\rangle \Rightarrow \langle\,\rangle\})\ d_0 \qquad (d_1 \text{ fresh}) \\ \mapsto & !\mathsf{cell}\ d_1\ \langle\,\rangle, \mathsf{cont}\ d_1\ (\mathbf{case}\ \_\ \{\langle\,\rangle \Rightarrow \langle\,\rangle\})\ d_0 \\ \mapsto & !\mathsf{cell}\ d_1\ \langle\,\rangle, \mathsf{eval}\ \langle\,\rangle\ d_0 \\ \mapsto & !\mathsf{cell}\ d_1\ \langle\,\rangle, !\mathsf{cell}\ d_0\ \langle\,\rangle \end{array}$$

We see that in the final state the initial destination d_0 holds the unit value $\langle\,\rangle$. In addition, there is some "junk" in the configuration, namely the cell d_1. This could safely be garbage-collected, although in this lecture we are not concerned with the definition and process of garbage collection.

In this example it may look like that the two objects that interact in the rules for continuations have to be next to each other, which is not the case in general. Even though we tend to write the state of the S machine in a sort-of canonical order with the store (cell objects) farthest to the left, then the eval object, if there is one, and then a sequence of continuations (cont objects) with the most recently created leftmost, this is not technically required.

2.2 Functions

Functions are relatively complex and thereby a good sample for how to design an abstract machine. λ-expressions are values, so the first rule is straightforward.

$$\mathsf{eval}\ (\lambda x.\, e)\ d \quad \mapsto \quad !\mathsf{cell}\ d\ (\lambda x.\, e)$$

Some sound objection might be raised to this rule, since allocated memory should have fixed size but the a λ-expression may not. In this lecture, we ask you to suspend this objection; in the next lecture we will present one way to make this aspect of the S machine more realistic.

As usual in a call-by-value language, an application is evaluated by first evaluating the function, then the argument, and then perform a β-reduction. We will reuse the continuations previously created for this purpose in the K machine.

$$\begin{array}{l} \mathsf{eval}\ (e_1\, e_2)\ d \ \mapsto\ \mathsf{eval}\ e_1\ d_1, \mathsf{cont}\ d_1\ (\_\, e_2)\ d \qquad (d_1 \text{ fresh}) \\ !\mathsf{cell}\ d_1\ c_1, \mathsf{cont}\ d_1\ (\_\, e_2)\ d \ \mapsto\ \mathsf{eval}\ e_2\ d_2, \mathsf{cont}\ d_2\ (d_1\, \_)\ d \qquad (d_2 \text{ fresh}) \\ !\mathsf{cell}\ d_1\ (\lambda x.\, e_1'), !\mathsf{cell}\ d_2\ c_2, \mathsf{cont}\ d_2\ (d_1\, \_)\ d \ \mapsto\ \mathsf{eval}\ ([d_2/x]e_1')\ d \end{array}$$

The first two rules should be expected, since they are a a straighforward rewrite of the K machine's transition rules. Note that in the second rule we check that the cell d_1 holds a value (c_1), but we actually do not use the contents. Nevertheless, this check is necessary to ensure that operation of the S machine is deterministic: there always is a unique next step, assuming we start in state

$$\mathsf{eval}\ e\ d_0$$

and stop when there are no eval or cont cells left.

The interesting aspect of the last rule is that it we substitute not a *value* (as we have done in the dynamics so far, including the K machine), but the *address* d_2 of a value. This necessitates a further rule, namely how to evaluate a destination! The destination amounts to a reference to the store, so we have to copy the contents at one address to another. Since we imagine the size of storage locations to be fixed and small, this is a reasonable operation.

$$!\mathsf{cell}\ d\ c, \mathsf{eval}\ d\ d' \quad \mapsto \quad !\mathsf{cell}\ d'\ c$$

There is an alternative line of thought where we store in the cell d' not a copy of the value c, but a reference to the value c. Then, of course, we would have to follow chains of references and rules that need to access the contents of cells would become more complicated.

Because fixed points are usually used for functions, the simple and straightforward rule just unrolls the recursion.

$$\mathsf{eval}\ (\mathbf{fix}\, x.\, e)\ d \quad \mapsto \quad \mathsf{eval}\ ([\mathbf{fix}\, x.\, e/x]e)\ d$$

In the next lecture we will look at a different semantics for fixed points since we want to avoid substitution into expressions.

3 A Simple Example

As a simple example, we consider the evaluation of $((\lambda x.\, \lambda y.\, x)\, 7)\, 5$ with an initial destination d_0. Here, 7 and 5 stand in for values that can be directly stored in memory, to simplify the example.

$$
\begin{array}{lll}
 & \mathsf{eval}\ (((\lambda x.\, \lambda y.\, x)\, 7)\, 5)\ d_0 & \\
\mapsto & \mathsf{eval}\ ((\lambda x.\, \lambda y.\, x)\, 7)\ d_1, \mathsf{cont}\ d_1\ (\_\, 5)\ d_0 & (d \\
\mapsto & \mathsf{eval}\ (\lambda x.\, \lambda y.\, x)\ d_2, \mathsf{cont}\ d_2\ (\_\, 7)\ d_1, \mathsf{cont}\ d_1\ (\_\, 5)\ d_0 & (d \\
\mapsto & \mathsf{!cell}\ d_2\ (\lambda x.\, \lambda y.\, x), \mathsf{cont}\ d_2\ (\_\, 7)\ d_1, \mathsf{cont}\ d_1\ (\_\, 5)\ d_0 & \\
\mapsto & \mathsf{!cell}\ d_2\ (\lambda x.\, \lambda y.\, x), \mathsf{eval}\ 7\ d_3, \mathsf{cont}\ d_3\ (d_2\, \_)\ d_1, \mathsf{cont}\ d_1\ (\_\, 5)\ d_0 & (d \\
\mapsto & \mathsf{!cell}\ d_2\ (\lambda x.\, \lambda y.\, x), \mathsf{!cell}\ d_3\ 7, \mathsf{cont}\ d_3\ (d_2\, \_)\ d_1, \mathsf{cont}\ d_1\ (\_\, 5)\ d_0 & \\
\mapsto & \mathsf{!cell}\ d_2\ (\lambda x.\, \lambda y.\, x), \mathsf{!cell}\ d_3\ 7, \mathsf{eval}\ (\lambda y.\, d_3)\ d_1, \mathsf{cont}\ d_1\ (\_\, 5)\ d_0 & \\
\mapsto & \mathsf{!cell}\ d_2\ (\lambda x.\, \lambda y.\, x), \mathsf{!cell}\ d_3\ 7, \mathsf{!cell}\ d_1\ (\lambda y.\, d_3), \mathsf{cont}\ d_1\ (\_\, 5)\ d_0 & \\
\mapsto & \mathsf{!cell}\ d_2\ (\lambda x.\, \lambda y.\, x), \mathsf{!cell}\ d_3\ 7, \mathsf{!cell}\ d_1\ (\lambda y.\, d_3), \mathsf{eval}\ 5\ d_4, \mathsf{cont}\ d_4\ (d_1\, \_)\ d_0 & (d \\
\mapsto & \mathsf{!cell}\ d_2\ (\lambda x.\, \lambda y.\, x), \mathsf{!cell}\ d_3\ 7, \mathsf{!cell}\ d_1\ (\lambda y.\, d_3), \mathsf{!cell}\ d_4\ 5, \mathsf{cont}\ d_4\ (d_1\, \_)\ d_0 & \\
\mapsto & \mathsf{!cell}\ d_2\ (\lambda x.\, \lambda y.\, x), \mathsf{!cell}\ d_3\ 7, \mathsf{!cell}\ d_1\ (\lambda y.\, d_3), \mathsf{!cell}\ d_4\ 5, \mathsf{eval}\ d_3\ d_0 & \\
\mapsto & \mathsf{!cell}\ d_2\ (\lambda x.\, \lambda y.\, x), \mathsf{!cell}\ d_3\ 7, \mathsf{!cell}\ d_1\ (\lambda y.\, d_3), \mathsf{!cell}\ d_4\ 5, \mathsf{!cell}\ d_0\ 7 &
\end{array}
$$

4 Eager Pairs

Eager pairs are somewhat similar to functions, but we construct a pair in memory as soon as the two components are evaluated. An interesting aspect of the S machine is that we form a new cell containing just a pair of destinations, indicating where the components of the pair are stored.

$$
\begin{array}{lll}
\mathsf{eval}\ \langle e_1, e_2\rangle\ d & \mapsto & \mathsf{eval}\ e_1\ d_1, \mathsf{cont}\ d_1\ \langle \_, e_2\rangle\ d \qquad (d_1 \text{ fresh}) \\
\mathsf{!cell}\ d_1\ c_1, \mathsf{cont}\ d_1\ \langle \_, e_2\rangle\ d & \mapsto & \mathsf{eval}\ e_2\ d_2, \mathsf{cont}\ d_2\ \langle d_1, \_\rangle\ d \qquad (d_2 \text{ fresh}) \\
\mathsf{!cell}\ d_2\ c_2, \mathsf{cont}\ d_2\ \langle d_1, \_\rangle\ d & \mapsto & \mathsf{!cell}\ d\ \langle d_1, d_2\rangle
\end{array}
$$

In lecture, it was pointed out is might make sense to also check that cell d_1 holds a value, with a another persistent !cell d_1 c_1 on the left-hand side. This is redundant because in a sequential semantics the continuation $\langle d_1, \_\rangle$ only makes sense if d_1 already holds a value. The difference between the rules is therefore just a matter of style.

In the rule for the destructor of eager pairs we perform again a substitution of destinations in an expression, as already seen for functions.

$$
\begin{array}{lll}
\mathsf{eval}\ (\mathbf{case}\ e\ \{\langle x_1, x_2\rangle \Rightarrow e'\})\ d' & \mapsto & \mathsf{eval}\ e\ d, \mathsf{cont}\ d\ (\mathbf{case}\ \_\ \{\langle x_1, x_2\rangle \Rightarrow e'\})\ d' \\
 & & \qquad (d \text{ fresh} \\
\mathsf{!cell}\ d\ \langle d_1, d_2\rangle, \mathsf{cont}\ d\ (\mathbf{case}\ \_\ \{\langle x_1, x_2\rangle \Rightarrow e'\})\ d' & \mapsto & \mathsf{eval}\ ([d_1/x_2, d_2/x_2]e')\ d'
\end{array}
$$

5 Typing the Store

First, a summary of the three types we have considered so far.

$$
\begin{array}{llcll}
\text{Expressions} & e & ::= & x & \text{(variables)} \\
 & & \mid & d & \text{(destination} \\
 & & \mid & \lambda x.\, e \mid e_1\, e_2 & (\rightarrow) \\
 & & \mid & \langle\,\rangle \mid \textbf{case}\ e\ \{\langle\,\rangle \Rightarrow e'\} & (1) \\
 & & \mid & \langle e_1, e_2\rangle \mid \textbf{case}\ e\ \{\langle x_1, x_2\rangle \Rightarrow e'\} & (\otimes) \\[1ex]
\text{Continuations} & k & ::= & (\_\ e_2) \mid (d_1\ \_) & (\rightarrow) \\
 & & \mid & \textbf{case}\ \_\ \{\langle\,\rangle \Rightarrow e'\} & (1) \\
 & & \mid & \langle \_, e_2\rangle \mid \langle d_1, \_\rangle \mid \textbf{case}\ \_\ \{\langle x_1, x_2\rangle \Rightarrow e'\} & (\otimes) \\[1ex]
\text{Cell Contents} & c & ::= & \langle\,\rangle \mid \langle d_1, d_2\rangle \mid \lambda x.\, e &
\end{array}
$$

From these examples we can readily extrapolate the rest of the S machine. Continuations haven't really changed from the K machine except we only use a small piece at a time and not whole stacks. We just show the possible cell contents, organized by type, thereby describing the possible shapes of memory.

$$
\begin{array}{llcll}
\text{Cell Contents} & c & ::= & \langle\,\rangle & (1) \\
 & & \mid & \langle d_1, d_2\rangle & (\otimes) \\
 & & \mid & \ell \cdot d & (+) \\
 & & \mid & \textbf{fold}\ d & (\rho) \\
 & & \mid & \langle\!| e_1, e_2 |\!\rangle & (\&) \\
 & & \mid & \lambda x.\, e & (\rightarrow)
\end{array}
$$

We assign types to the store by typing each destination and then checking for consistent usage. We use

$$
\text{Store Typing} \quad \Sigma \quad ::= \quad d_1 : \tau_1, \ldots, d_n : \tau_n
$$

In a store typing, all destinations must be distinct. Notice the difference to the usual typing context Γ that types *variables*, while Σ assign types to destinations. At runtime, we only execute expression without free variables, but several rules (for example, for function calls) will substitute a destination for a variable. Therefore, we type expressions with $\Sigma, \Gamma \vdash e : \tau$ with the additional rule

$$
\frac{d : \tau \in \Sigma}{\Sigma, \Gamma \vdash d : \tau}\ \mathsf{Dest}
$$

while in all other rules we just add Σ and propagate it from the conclusion to all premises.

Next we move on to typing objects. For uniformity we write $\Sigma \vdash d : \tau$ if $d : \tau \in \Sigma$. We type each object P with the judgment $\Sigma \vdash P\ \mathit{obj}$. From this, the typings are rather straightforward.

$$\frac{\Sigma \vdash d : \tau \quad \Sigma \vdash e : \tau}{\Sigma \vdash (\mathsf{eval}\ e\ d)\ \mathit{obj}} \qquad \frac{\Sigma \vdash d : \tau \quad \Sigma \vdash c :: \tau}{\Sigma \vdash (!\mathsf{cell}\ d\ c)\ \mathit{obj}}$$

$$\frac{\Sigma \vdash d_1 : \tau_1 \quad \Sigma \vdash d_2 : \tau_2 \quad \Sigma \vdash k \div \tau_1 \Rightarrow \tau_2}{\Sigma \vdash (\mathsf{cont}\ d_1\ k\ d_2)\ \mathit{obj}}$$

A state is well-typed with respect to store typing Σ if each object in it is a valid object. This form of typing is inadequate in several respects and, in particular, it does not guarantee progress. An initial state has the form eval $e\ d_0$ for a destination d_0 and a final state consists solely of memory cells !cell $d_i\ c_i$ (which should include d_0). However, a state such as *cont* $d_2\ \langle d_1, \_\rangle\ d_0$ is a perfectly valid state for the store typing

$$d_0 : \tau_1 \otimes \tau_2, d_1 : \tau_1, d_2 : \tau_2$$

for any types τ_1, τ_2, but cannot make a transition. We may address the question how to obtain a more precise typing for states of the machine with store in a later lecture.

We still owe the rules for the contents of the store. They do not present any difficulty. In the rules for the eager constructs ((C-1), (C-$\otimes$), (C-$+$), (C-ρ)) we refer only directly to the types of other destinations, while for the lazy ones ((C-&), (C-$\to$)) we have to type the embedded expressions.

$$\frac{}{\Sigma \vdash \langle\,\rangle :: 1}\ (\text{C-}1) \qquad \frac{\Sigma \vdash d_1 : \tau_1 \quad \Sigma \vdash d_2 : \tau_2}{\Sigma \vdash \langle d_1, d_2\rangle :: \tau_1 \otimes \tau_2}\ (\text{C-}\otimes)$$

$$\frac{\Sigma \vdash d : \tau_i \quad (i \in L)}{\Sigma \vdash i \cdot d :: \Sigma_{\ell \in L}(\ell : \tau_\ell)}\ (\text{C-}+) \qquad \frac{\Sigma \vdash d : [\rho\alpha.\,\tau/\alpha]\tau}{\Sigma \vdash \mathbf{fold}\ d :: \rho\alpha.\,\tau}\ (\text{C-}\rho)$$

$$\frac{\Sigma \vdash e_1 : \tau_1 \quad \Sigma \vdash e_2 : \tau_2}{\Sigma \vdash \langle\!|e_1, e_2|\!\rangle :: \tau_1 \mathbin{\&} \tau_2}\ (\text{C-}\&) \qquad \frac{\Sigma, x : \tau_1 \vdash e : \tau_2}{\Sigma \vdash \lambda x.\, e :: \tau_1 \to \tau_2}\ (\text{C-}\to)$$

6 Concurrency/Parallelism

Both in the K machine and the S machine we ensured that evaluation was *sequential*: there was always a unique next step to take. Our dynamics formalism is general enough to support parallel or concurrent evaluation. Consider, for example, an eager pair. We can evaluate the components of a pair independently, each with a new separate destination. Moreover, we can immediately fill the destination with a pair so that further computation can proceed before either component finishes!

$$\mathsf{eval}\ \langle e_1, e_2\rangle\ d \quad\mapsto\quad !\mathsf{cell}\ d\ \langle d_1, d_2\rangle, \mathsf{eval}\ e_1\ d_1, \mathsf{eval}\ e_2\ d_2$$

Recall the rules for the pair destructor.

$$\begin{array}{rcl}
\mathsf{eval}\ (\mathbf{case}\ e\ \{\langle x_1, x_2\rangle \Rightarrow e'\})\ d' & \mapsto & \mathsf{eval}\ e\ d, \mathsf{cont}\ d\ (\mathbf{case}\ \_\ \{\langle x_1, x_2\rangle \Rightarrow e'\})\ d' \\
 & & (d\ \text{fresh}) \\
!\mathsf{cell}\ d\ \langle d_1, d_2\rangle, \mathsf{cont}\ d\ (\mathbf{case}\ \_\ \{\langle x_1, x_2\rangle \Rightarrow e'\})\ d' & \mapsto & \mathsf{eval}\ ([d_1/x_2, d_2/x_2]e')\ d'
\end{array}$$

We see that the body of the **case** construct can evaluate as soon as the cell d has been filled with a pair of destinations, but before either of these destinations has been filled. This enables a lot of fine-grained parallelism, so much so, that if we try to do everything in parallel in many programs there would simply be too many threads of control to execute efficiently.

We also observe that the distinction between *eager* (or *strict*) and *lazy* is difficult to apply to this situation. Both components of the pair are evaluated, but we don't wait for them to finish. If only one component is needed in the body of the **case**, the other might not terminate and yet we may have filled the initial destination d_0.

We may return to a closer examination of a language supporting parallelism or concurrency in a future lecture.

References

[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.

Lecture Notes on
Closures

15-814: Types and Programming Languages
Frank Pfenning

Lecture 17
November 1, 2018

1 Introduction

In the S machine, we still freely substitute into expressions, which goes somewhat against the idea that expressions should be compiled. Also, we directly store expressions in memory cells, even though their space requirements are not clear and not small.

In this lecture we first review the S machine and then update it to avoid substitution into expressions. Instead we construct *environments* to hold the bindings for the variables in an expression and then *closures* to pair up an environment with an expression as a closed value.

2 Semantic Objects in the S Machine

We briefly summarize the S machine from the previous lecture. At its core are *destinations* d (also called *addresses*) to hold values in the store. The only operation on them is to generate fresh ones. The state of the S machine consists of the following objects:

eval $e\ d$**.** Evaluate expression e, storing the result in destination d.

!cell $d\ c$**.** Cell d has contents c. The exclamation mark '!' indicates that cells are *persistent*, which means the value of a cell can never change and will be available during the whole remainder of the computation.

cont $d\ k\ d'$**.** Continuation k receives a value in destination in d and puts result into d'.

First, a summary of the three types we have considered so far.

$$
\begin{array}{llcll}
\text{Expressions} & e & ::= & x & \text{(variables)} \\
 & & \mid & d & \text{(destination} \\
 & & \mid & \lambda x.\, e \mid e_1\, e_2 & (\rightarrow) \\
 & & \mid & \langle\,\rangle \mid \textbf{case}\ e\ \{\langle\,\rangle \Rightarrow e'\} & (1) \\
 & & \mid & \langle e_1, e_2\rangle \mid \textbf{case}\ e\ \{\langle x_1, x_2\rangle \Rightarrow e'\} & (\otimes) \\[1em]
\text{Continuations} & k & ::= & (\_\ e_2) \mid (d_1\ \_) & (\rightarrow) \\
 & & \mid & \textbf{case}\ \_\ \{\langle\,\rangle \Rightarrow e'\} & (1) \\
 & & \mid & \langle \_, e_2\rangle \mid \langle d_1, \_\rangle \mid \textbf{case}\ \_\ \{\langle x_1, x_2\rangle \Rightarrow e'\} & (\otimes) \\[1em]
\text{Cell Contents} & c & ::= & \langle\,\rangle \mid \langle d_1, d_2\rangle \mid \lambda x.\, e &
\end{array}
$$

From these examples we can readily extrapolate the rest of the S machine. We just show the possible cell contents, organized by type, thereby describing the possible shapes of memory.

$$
\begin{array}{llcll}
\text{Cell Contents} & c & ::= & \langle\,\rangle & (1) \\
 & & \mid & \langle d_1, d_2\rangle & (\otimes) \\
 & & \mid & \ell \cdot d & (+) \\
 & & \mid & \textbf{fold}\ d & (\rho) \\
 & & \mid & \langle\!| e_1, e_2 |\!\rangle & (\&) \\
 & & \mid & \lambda x.\, e & (\rightarrow)
\end{array}
$$

We assign types to the store by typing each destination and then checking for consistent usage. We use

$$
\text{Store Typing} \quad \Sigma \quad ::= \quad d_1 : \tau_1, \ldots, d_n : \tau_n
$$

where all the destinations d_i are distinct. We type semantics objects as

$$
\frac{\Sigma \vdash d : \tau \quad \Sigma \vdash e : \tau}{\Sigma \vdash (\mathsf{eval}\ e\ d)\ \mathit{obj}}
\qquad
\frac{\Sigma \vdash d : \tau \quad \Sigma \vdash c :: \tau}{\Sigma \vdash (\mathsf{!cell}\ d\ c)\ \mathit{obj}}
$$

$$
\frac{\Sigma \vdash d_1 : \tau_1 \quad \Sigma \vdash d_2 : \tau_2 \quad \Sigma \vdash k \div \tau_1 \Rightarrow \tau_2}{\Sigma \vdash (\mathsf{cont}\ d_1\ k\ d_2)\ \mathit{obj}}
$$

and the contents of cells with the following rules:

$$\frac{}{\Sigma \vdash \langle\,\rangle :: 1}\ (\text{C-}1) \qquad \frac{\Sigma \vdash d_1 : \tau_1 \quad \Sigma \vdash d_2 : \tau_2}{\Sigma \vdash \langle d_1, d_2 \rangle :: \tau_1 \otimes \tau_2}\ (\text{C-}\otimes)$$

$$\frac{\Sigma \vdash d : \tau_i \quad (i \in L)}{\Sigma \vdash i \cdot d :: \Sigma_{\ell \in L}(\ell : \tau_\ell)}\ (\text{C-}+) \qquad \frac{\Sigma \vdash d : [\rho\alpha.\, \tau/\alpha]\tau}{\Sigma \vdash \mathbf{fold}\ d :: \rho\alpha.\, \tau}\ (\text{C-}\rho)$$

$$\frac{\Sigma \vdash e_1 : \tau_1 \quad \Sigma \vdash e_2 : \tau_2}{\Sigma \vdash \langle\!|e_1, e_2|\!\rangle :: \tau_1 \mathbin{\&} \tau_2}\ (\text{C-}\&) \qquad \frac{\Sigma, x : \tau_1 \vdash e : \tau_2}{\Sigma \vdash \lambda x.\, e :: \tau_1 \to \tau_2}\ (\text{C-}\to)$$

The dynamics is given with the following rules:

$$\mathsf{!cell}\ d\ c, \mathsf{eval}\ d\ d' \quad \mapsto \quad \mathsf{!cell}\ d'\ c$$

$$\begin{array}{l}
\mathsf{eval}\ \langle\,\rangle\ d \quad \mapsto \quad \mathsf{!cell}\ d\ \langle\,\rangle \\
\mathsf{eval}\ (\mathbf{case}\ e\ \{\langle\,\rangle \Rightarrow e'\})\ d' \quad \mapsto \quad \mathsf{eval}\ e\ d, \mathsf{cont}\ d\ (\mathbf{case}\ \_\ \{\langle\,\rangle \Rightarrow e'\})\ d' \quad (d \text{ fresh}) \\
\mathsf{!cell}\ d\ \langle\,\rangle, \mathsf{cont}\ d\ (\mathbf{case}\ \_\ \{\langle\,\rangle \Rightarrow e'\})\ d' \quad \mapsto \quad \mathsf{eval}\ e'\ d'
\end{array}$$

$$\begin{array}{l}
\mathsf{eval}\ (\lambda x.\, e)\ d \quad \mapsto \quad \mathsf{!cell}\ d\ (\lambda x.\, e) \\
\mathsf{eval}\ (e_1\ e_2)\ d \quad \mapsto \quad \mathsf{eval}\ e_1\ d_1, \mathsf{cont}\ d_1\ (\_\ e_2)\ d \qquad (d_1 \text{ fresh}) \\
\mathsf{!cell}\ d_1\ c_1, \mathsf{cont}\ d_1\ (\_\ e_2)\ d \quad \mapsto \quad \mathsf{eval}\ e_2\ d_2, \mathsf{cont}\ d_2\ (d_1\ \_)\ d \qquad (d_2 \text{ fresh}) \\
\mathsf{!cell}\ d_1\ (\lambda x.\, e_1'), \mathsf{!cell}\ d_2\ c_2, \mathsf{cont}\ d_2\ (d_1\ \_)\ d \quad \mapsto \quad \mathsf{eval}\ ([d_2/x]e_1')\ d
\end{array}$$

$$\begin{array}{l}
\mathsf{eval}\ \langle e_1, e_2 \rangle\ d \quad \mapsto \quad \mathsf{eval}\ e_1\ d_1, \mathsf{cont}\ d_1\ \langle \_, e_2 \rangle\ d \qquad (d_1 \text{ fresh}) \\
\mathsf{!cell}\ d_1\ c_1, \mathsf{cont}\ d_1\ \langle \_, e_2 \rangle\ d \quad \mapsto \quad \mathsf{eval}\ e_2\ d_2, \mathsf{cont}\ d_2\ \langle d_1, \_ \rangle\ d \qquad (d_2 \text{ fresh}) \\
\mathsf{!cell}\ d_2\ c_2, \mathsf{cont}\ d_2\ \langle d_1, \_ \rangle\ d \quad \mapsto \quad \mathsf{!cell}\ d\ \langle d_1, d_2 \rangle \\
\mathsf{eval}\ (\mathbf{case}\ e\ \{\langle x_1, x_2 \rangle \Rightarrow e'\})\ d' \quad \mapsto \quad \mathsf{eval}\ e\ d, \mathsf{cont}\ d\ (\mathbf{case}\ \_\ \{\langle x_1, x_2 \rangle \Rightarrow e'\})\ d \\
\hfill (d \text{ fresh}
\end{array}$$

$$\mathsf{!cell}\ d\ \langle d_1, d_2 \rangle, \mathsf{cont}\ d\ (\mathbf{case}\ \_\ \{\langle x_1, x_2 \rangle \Rightarrow e'\})\ d' \quad \mapsto \quad \mathsf{eval}\ ([d_1/x_2, d_2/x_2]e')\ d'$$

$$\mathsf{eval}\ (\mathbf{fix}\, x.\, e)\ d \quad \mapsto \quad \mathsf{eval}\ ([\mathbf{fix}\, x.\, e/x]e)\ d$$

3 Environments

For the eager constructs of the language, this representation of values in the store is adequate, if perhaps consuming a bit too much space. For example, the value 1 at destination d_0 would be

$$\begin{array}{l} !\mathsf{cell}\ d_0\ (\textbf{fold}\ d_1), \\ !\mathsf{cell}\ d_1\ (\mathsf{s} \cdot d_2), \\ !\mathsf{cell}\ d_2\ (\textbf{fold}\ d_3), \\ !\mathsf{cell}\ d_3\ (\mathsf{z} \cdot d_4), \\ !\mathsf{cell}\ d_4\ \langle\,\rangle \end{array}$$

Up to a constant factor, this is what one might expect.

However, expressions such as the values $\lambda x.\, e$ and $\langle\!| e_1, e_2 |\!\rangle$ are treated not quite in the way we might envision in a lower-level semantics. Functions should be compiled to efficient machine code, which is justified in part by saying that we can not observe their internal forms. Moreover, in the dynamics of the S machine we substitute destinations into expressions to obtain new ones that we then evaluate. In a lower-level implementation, such a substitution is unrealistic. Instead, we compile variables so they reference the store, either on a stack or in the heap. While we don't model this distinction here, we would still like to model that code is essentially immutable, and the values held in variables are stored in memory.

The first key idea is not to substitute into an expression, but instead maintain an *environment* that maps variables to values. In the case of the K machine, these values would be the same as we had in our original, high-level semantics. In the case of the S machine, the values are simply store addresses where the value is represented.

$$\text{Environments} \quad \eta \quad ::= \quad d_1/x_1, \ldots, d_n/x_n$$

We require that all the variables x_i are distinct so that the value of each variable is uniquely determined. The destinations d_i however do **not** need to be distinct: it is perfectly possible that two different program variables contain references to the same storage cell.

Previously we were careful to evaluate only *closed* expressions. Now we evaluate expressions in an environment that substitutes destinations for all of its free variables. Of course, the type of the destination must match the type of the variables it substitutes for. To ensure this we use the typing judgment $\Sigma \vdash \eta : \Gamma$ defined by the two rules

$$\frac{}{\Sigma \vdash (\cdot) : (\cdot)} \qquad \frac{\Sigma \vdash \eta : \Gamma \quad \Sigma \vdash d : \tau}{\Sigma \vdash (\eta, d/x) : (\Gamma, x : \tau)}$$

Now evaluation depends on an environment

$$\frac{\Sigma \vdash d : \tau \quad \Sigma \vdash \eta : \Gamma \quad \Gamma \vdash e : \tau}{\Sigma \vdash \mathsf{eval}\ \eta\ e\ d\ \mathit{obj}}$$

Compared to the S machine in the previous lecture, expressions now no longer contain destinations, so the typing judgments for expressions reverts to be pure, $\Gamma \vdash e : \tau$.

4 Evaluation with Environments

Now we revisit the rules of the S machine in the presence of environments. Let's call this new machine the S_η machine. Previously we had

$$\mathsf{!cell}\ d\ c, \mathsf{eval}\ d\ d' \quad \mapsto \quad \mathsf{!cell}\ d'\ c \quad \text{(S machine)}$$

Now, this becomes a rule for *variables* which must be defined in the environment

$$\mathsf{!cell}\ d\ c, \mathsf{eval}\ \eta\ x\ d' \quad \mapsto \quad \mathsf{!cell}\ d'\ c \quad (d/x \in \eta)$$

For functions, we had

$$\mathsf{eval}\ (\lambda x.\, e)\ d \quad \mapsto \quad \mathsf{!cell}\ d\ (\lambda x.\, e) \quad \text{(S machine)}$$

Now we have to pair up the environment with the λ-abstraction in order to form a *closure*. It is called a closure because it "closes up" the expression e all of whose free variables are defined in η.

$$\mathsf{eval}\ \eta\ (\lambda x.\, e)\ d \quad \mapsto \quad \mathsf{!cell}\ d\ \langle \eta, \lambda x.\, e\rangle$$

For an application $e_1\, e_2$ we have to evaluate e_1, but we also have to remember the environment in which e_2 makes sense. In a another implementation, this might be captured in an environment stack. Here, we just keep track of the environment in the continuation, building a temporary closure $\langle \eta, e_2\rangle$. After evaluation of e_1 we continue the evaluation of e_2 in the saved environment.

$$\begin{array}{lcll} \mathsf{eval}\ \eta\ (e_1\, e_2)\ d & \mapsto & \mathsf{eval}\ \eta\ e_1\ d_1, \mathsf{cont}\ d_1\ (\_\ \langle \eta, e_2\rangle)\ d & (d_1 \text{ fresh}) \\ \mathsf{!cell}\ d_1\ c_1, \mathsf{cont}\ d_1\ (\_\ \langle \eta, e_2\rangle)\ d & \mapsto & \mathsf{eval}\ \eta\ e_2\ d_2, \mathsf{cont}\ d_2\ (d_1\ \_)\ d & (d_2 \text{ fresh}) \end{array}$$

The most interesting rule is the one where we actually pass the argument to the function. Previously, we just substituted the address of the argument value; now we add it to the environment.

$$\mathsf{!cell}\ d_1\ \langle \eta, \lambda x.\, e_1'\rangle, \mathsf{!cell}\ d_2\ c_2, \mathsf{cont}\ d_2\ (d_1\ \_)\ d \quad \mapsto \quad \mathsf{eval}\ (\eta, d_2/x)\ e_1'\ d$$

It is easy to see that this environment is the correct one. On the left-hand side, given the store typing Σ, we have

$$\Sigma \vdash \eta : \Gamma \quad \text{and} \quad \Gamma \vdash \lambda x.\, e_1' : \tau$$

for some Γ and τ. By inversion, we also have

$$\Gamma, x : \tau_2 \vdash e_1' : \tau_1$$

with $\tau = \tau_2 \to \tau_1$. Also

$$\Sigma \vdash (\eta, d_2/x) : (\Gamma, x : \tau_2)$$

since $\Sigma \vdash d_1 : \tau_2 \to \tau_1$ and $\Sigma \vdash d_2 : \tau_2$ from inversion on the continuation typing.

There is nothing much interesting in the remaining rules, but we will show those for lazy pairs because they also involve closures precisely because they are lazy.

$$\begin{array}{lll}\mathsf{eval}\ \eta\ \langle\!|e_\ell|\!\rangle_{\ell \in L}\ d & \mapsto & !\mathsf{cell}\ d\ \langle \eta, \langle\!|e_\ell|\!\rangle_{\ell \in L} \rangle \\ \mathsf{eval}\ \eta\ (e \cdot i)\ d & \mapsto & \mathsf{eval}\ \eta\ e\ d_1, \mathsf{cont}\ d_1\ (\_ \cdot i)\ d \quad (d_1 \text{ fresh}) \\ !\mathsf{cell}\ d_1\ \langle \eta, \langle\!|e_\ell|\!\rangle_{\ell \in L} \rangle, \mathsf{cont}\ d_1\ (\_ \cdot i)\ d & \mapsto & \mathsf{eval}\ \eta\ e_i\ d\end{array}$$

At this point we might ask if we have actually satisfied our goal of storing only data of fixed size. We imagine that in an implementation the code is compiled, with variables becoming references into an environment. Then the expression part of a closure is a pointer to code that in turn expect to be passed the address of the environment. As such, it is only the size of the environment which is of variable size. However, it can be predicted at the time of compilation. In our simple model, it consists of bindings for all variables that *might* occur free in e, that is, all variable in Γ if e was checked with $\Gamma \vdash e : \tau$. We can slightly improve on this, keeping only the variables of Γ that actually occur free in e. Thus, while the space for different closures is of different size, we can calculate it at compile time, and it is proportional to the number of free variables in e.

5 Fixed Points

Fixed points are interesting. The rule of the S machine

$$\mathsf{eval}\ (\mathbf{fix}\, x.\, e)\ d \quad \mapsto \quad \mathsf{eval}\ ([\mathbf{fix}\, x.\, e/x]e)\ d$$

(and also the corresponding rule of the K machine) substitutes an *expression* for a variable, while all other rules in our call-by-value language just substitute either values (K machine) or destinations (S machine). Since one of our goals is to eliminate substitution into expressions, we should change this rule somehow. First idea might be to just add the expression to the environment, but a rule such as

$$\mathsf{eval}\ \eta\ (\mathbf{fix}\, x.\, e)\ d \quad\mapsto\quad \mathsf{eval}\ (\eta, (\mathbf{fix}\, x.\, e)/x)\ e\ d \quad ??$$

would add expressions to the environment, upsetting our carefully constructed system. In particular, looking up a variable doesn't necessarily result in a destination. Perhaps even worse, the expression **fix** $x.\, e$ is not closed, so at the very least we'd have to construct another closure.

$$\mathsf{eval}\ \eta\ (\mathbf{fix}\, x.\, e)\ d \quad\mapsto\quad \mathsf{eval}\ (\eta, \langle \eta, \mathbf{fix}\, x.\, e\rangle/x)\ e\ d \quad ??$$

We pursue here a different approach, namely evaluating the body of the fixed point *as if* d *already held a value*!

$$\mathsf{eval}\ \eta\ (\mathbf{fix}\, x.\, e)\ d \quad\mapsto\quad \mathsf{eval}\ (\eta, d/x)\ e\ d$$

This upsets another invariant of our semantics so far, namely that any destination in the environment is defined in the store. This new rule speculates that d will contain a value by the time e might reference the variable x. This is not a trivial matter. Consider the expression **fix** $x.\, x$ in the empty environment $(\cdot)$.

$$\mathsf{eval}\ (\cdot)\ (\mathbf{fix}\, x.\, x)\ d_0 \quad\mapsto\quad \mathsf{eval}\ (d_0/x)\ x\ d_0$$

At this point the rule for variables

$$!\mathsf{cell}\ d\ c, \mathsf{eval}\ \eta\ x\ d' \quad\mapsto\quad !\mathsf{cell}\ d'\ c \quad (d/x \in \eta)$$

cannot be applied because destination d_0 does yet hold a value. In other words, the progress property fails!

This situation can indeed arise in Haskell where it is called a *black hole*. It is actually detected at runtime and a "black hole" is reported during execution. For example,

```
blackHole :: Int
blackHole = blackHole

main = print blackHole
```

compiles, but running it reports

```
black_hole: <<loop>>
```

We can imagine how this may be done: when the fixed point is executed we actually allocate a destination for its value and initialize it with a recognizable value indicating it has not yet been written. We may then modify the progress theorem to account for a black hole as a third form of outcome of the computation, besides a value or divergence.

In a call-by-value language there is a different solution: we can restrict the body of the fixed point expression to be a value, where the fixed point variable x does **not** count as a value. We believe[1] that this guarantees that the destination of the fixed point will always be defined before the fixed point variable x is encountered. The revised rule then reads

$$\mathsf{eval}\ \eta\ (\mathbf{fix}\, x.\, v)\ d \quad\mapsto\quad \mathsf{eval}\ (\eta, d/x)\ v\ d$$

where we have to be careful not to count x as a value. Evaluating the expression v will construct its representation in the store.

As an example, consider the following definition of natural number streams:

$$\begin{array}{lcl} \mathit{nat} & \simeq & (\mathsf{z} : 1) + (\mathsf{s} : \mathit{nat}) \\ \mathit{zero} & = & \mathbf{fold}\ (\mathsf{z} \cdot \langle\,\rangle) \\ \\ \mathit{stream} & \simeq & \mathit{nat}\ \&\ \mathit{stream} \\ \\ \mathit{zeros} & : & \mathit{stream} \\ \mathit{zeros} & = & \mathbf{fix}\, x.\, \mathbf{fold}\ \langle\!|\mathit{zero}, x|\!\rangle \end{array}$$

The stream *zeros* corresponds to a potentially unbounded number of zeros, computed lazily. We see that $\mathbf{fold}\ \langle\!|\mathit{zero}, x|\!\rangle$ is a value even if x is not, since any lazy pair is a value. Starting with the empty environment and initial destination d_0, we evaluate *zeros* as follows:

$$\begin{array}{ll} & \mathsf{eval}\ (\cdot)\ (\mathbf{fix}\ x.\, \mathbf{fold}\ \langle\!|\mathit{zero}, x|\!\rangle)\ d_0 \\ \mapsto & \mathsf{eval}\ (d_0/x)\ (\mathbf{fold}\ \langle\!|\mathit{zero}, x|\!\rangle)\ d_0 \\ \mapsto & \mathsf{eval}\ (d_0/x)\ \langle\!|\mathit{zero}, x|\!\rangle\ d_1, \mathsf{cont}\ d_1\ (\mathbf{fold}\ \_)\ d_0 \qquad (d_1\ \text{fresh}) \\ \mapsto & !\mathsf{cell}\ d_1\ \langle (d_0/x), \langle\!|\mathit{zero}, x|\!\rangle \rangle, \mathsf{cont}\ d_1\ (\mathbf{fold}\ \_)\ d_0 \\ \mapsto & !\mathsf{cell}\ d_1\ \langle (d_0/x), \langle\!|\mathit{zero}, x|\!\rangle \rangle, !\mathsf{cell}\ d_0\ (\mathbf{fold}\ d_1) \end{array}$$

At this point we have constructed a store with a circular chain of references: cell d_0 contains a reference to d_1, and d_1 contains a reference to d_0 in the

[1] but have not proven

environment stored with the closure. If we define

$$\begin{array}{lcl} hd & : & \mathit{stream} \to \mathit{stream} \\ hd & = & \lambda s.\,(\textbf{unfold}\,s)\cdot l \\[1ex] tl & : & \mathit{stream} \to \mathit{stream} \\ tl & = & \lambda s.\,(\textbf{unfold}\,s)\cdot r \end{array}$$

we should be able to check that *hd zeros* returns (a representation of) *zero*, while *tail zeros* returns (a representation of) *zeros*.

It is a good exercise to check that the *ascending* function below behaves as expected, where *ascending* $\ulcorner n \urcorner$ computes an ascending stream of natural numbers starting at $\ulcorner n \urcorner$.

$$\begin{array}{lcl} \mathit{succ} & : & \mathit{nat} \to \mathit{nat} \\ \mathit{succ} & = & \lambda n.\,\textbf{fold}\,(\mathsf{s}\cdot n) \\[1ex] \mathit{ascending} & : & \mathit{nat} \to \mathit{stream} \\ \mathit{ascending} & = & \lambda n.\,\textbf{fold}\,\langle\!|\, n, \mathit{ascending}\,(\mathit{succ}\ n) \,|\!\rangle \end{array}$$

Lecture Notes on Quotation

15-814: Types and Programming Languages
Frank Pfenning

Lecture 18
November 6, 2018

1 Introduction

One of the features that appear to be more prevalent in dynamically than statically typed languages is that of quotation and evaluation. In this lecture we will make sense of quotation type-theoretically and see that as a programming language construct it is closely related to prior work in philosophy on *modal logic* aimed at capturing similar phenomena in logical reasoning.

Our concrete aim and underlying intuition is to model *runtime code generation* [LL98]. This has actually become a staple of many programming language implementations in the guise of *just-in-time* compilation (see, for example, [KWM$^+$08]). Languages such as Java may be compiled to bytecode, which is then interpreted during execution. As long as we have interpreters for various machine architectures, this makes the code portable, but for efficiency reasons we may still want to compile the bytecode down to actual machine code "just in time" (essentially: as it runs). In our model, the programmer (rather than the environment) is in full control over whether and when code is generated, so it differs in this respect from much of the work on just-in-time compilation and is more similar to the quote and eval constructs of languages such as Lisp, Scheme, and Racket.

The approach of using a modal type system [DP01, PD01] has made its way into statically typed languages such as MetaOCaml [Kis14], although some of the technical details differ from what we present in this lecture.

2 A Type for Closed Source Expressions

Early attempts at runtime code generation for functional languages were based on the simple idea that a curried function such as $f : \tau_1 \to (\tau_2 \to \sigma)$ could take an argument of type τ_1 and then generate code for a residual function $f' : \tau_2 \to \sigma$. The problem with this approach was that, often, the program was written in such a way that the best that could be done is to generate a closure. Since generating code at runtime is expensive, in many cases programs would get slower. If we had a closed source expression for $\tau_2 \to \sigma$ we would be sure it no longer depended on the argument v_1 of type τ_1 and we could generate specialized code for this particular v_1.

As a start, let's capture closed expression of type τ in a type $\Box\tau$. The constructor is easy, using **box** e as the notation for a quoted expression e.

$$\frac{\cdot \vdash e : \tau}{\Gamma \vdash \mathbf{box}\ e : \Box\tau}\ (\text{I-}\Box)$$

No variables that are declared in Γ may appear in e, because we erase it in the premise.

The elimination rule is difficult. Most immediate attempts will be too weak to write interesting programs or are unsound. In the end, there seem to be essentially two approaches [DP01, PD01] that are equivalent in expressive power. We choose the simpler one, introducing a new kind of variable u that stands only for *source expressions*. We generalize our judgment to

$$\Psi \,;\, \Gamma \vdash e : \tau$$

where Ψ consists of expression variables $u_i : \tau_i$ and Γ consists of value variables $x_j : \tau_j$. Then the rule

$$\frac{\Psi \,;\, \Gamma \vdash e : \Box\tau \quad \Psi, u : \tau \,;\, \Gamma \vdash e' : \tau'}{\Psi \,;\, \Gamma \vdash \mathbf{case}\ e\ \{\mathbf{box}\ u \Rightarrow e'\} : \tau'}\ (\text{E-}\Box)$$

introduces a new expression variable $u : \tau$ with scope e'. The next key insight is that expression variables may appear under **box** constructors, because they will always be bound to source code. We revise our introduction rule:

$$\frac{\Psi \,;\, \cdot \vdash e : \tau}{\Psi \,;\, \Gamma \vdash \mathbf{box}\ e : \Box\tau}\ (\text{I-}\Box)$$

In the dynamics, every quoted expression is simply a value.

$$\frac{}{\mathbf{box}\ e\ \mathit{val}}\ (\text{V-}\Box)$$

We have to keep in mind, however, that it is different from lazy evaluation in that e must also be available in source form (at least conceptually, if not in an actual implementation). While an equivalent-looking lazy expression $\langle\!| e, \langle\,\rangle |\!\rangle : \tau \;\&\; 1$ can only be awakened for evaluation by the left projection, a quoted expression that be unwrapped and substituted into another quoted expression.

$$\frac{}{\textbf{case}\ (\textbf{box}\ e)\ \{\textbf{box}\ u \Rightarrow e'\} \mapsto [\![e/u]\!] e'}\ (\text{R-}\square)$$

We have used a different notation for substitution here to remind ourselves that we are substituting a source expression for an expression variable, which may have a very different implementation than substituting a value for an ordinary value variable.

We also have a standard congruence rule for the elimination construct.

$$\frac{e_0 \mapsto e_0'}{\textbf{case}\ e_0\ \{\textbf{box}\ u \Rightarrow e'\} \mapsto \textbf{case}\ e_0'\ \{\textbf{box}\ u \Rightarrow e'\}}\ (\text{CE-}\square)$$

3 An Example: Exponentiation

As an example, we consider exponentiation on natural numbers in unary representation. We allow pattern matching (knowing how it is elaborated into multiple **case** expressions) and assume multiplication can be written in infix notation $e_1 * e_2$.

We define a function $\mathit{pow}\ n\ b = b^n$, with the exponent as the first argument since it is defined recursively over this argument.

$$\begin{array}{lcl} \mathit{pow} & : & \mathit{nat} \to (\mathit{nat} \to \mathit{nat}) \\ \mathit{pow}\ Z\ b & = & S\,Z \\ \mathit{pow}\ (S\,n)\ b & = & b * \mathit{pow}\ n\ b \end{array}$$

We would now like to rewrite this code to another function exp such that $\mathit{exp}\ n$ returns code to compute b^n. It's type should be

$$\mathit{exp} : \mathit{nat} \to \square(\mathit{nat} \to \mathit{nat})$$

The case for $n = Z$ is easy:

$$\begin{array}{lcl} \mathit{exp}\ Z & = & \textbf{box}\ (\lambda b.\, S\,Z) \\ \mathit{exp}\ (S\,n) & = & \ldots \end{array}$$

The case for sucessor, however is tricky. We can **not** write something of the form

$$\begin{array}{lcl} \mathit{exp}\ Z & = & \mathbf{box}\ (\lambda b.\, S\, Z) \\ \mathit{exp}\ (S\, n) & = & \mathbf{box}\ (\lambda b.\ \ldots) \end{array}$$

because the value variable n is not available in the scope of the **box**. Clearly, though, the result will need to depend on n.

Instead, we make a recursive call to obtain the code for a function that computes $\lambda b.\, b^n$.

$$\begin{array}{lcl} \mathit{exp}\ Z & = & \mathbf{box}\ (\lambda b.\, S\, Z) \\ \mathit{exp}\ (S\, n) & = & \mathbf{case}\ (\mathit{exp}\ n)\ \{\mathbf{box}\ u \Rightarrow \underbrace{\qquad\qquad}_{:\, \Box(\mathit{nat} \to \mathit{nat})}\ \} \end{array}$$

Because u is an expression variable we can now employ quotation

$$\begin{array}{lcl} \mathit{exp}\ Z & = & \mathbf{box}\ (\lambda b.\, S\, Z) \\ \mathit{exp}\ (S\, n) & = & \mathbf{case}\ (\mathit{exp}\ n)\ \{\mathbf{box}\ u \Rightarrow \mathbf{box}\ (\ \underbrace{\qquad\qquad}_{:\, \mathit{nat} \to \mathit{nat}}\ )\} \end{array}$$

Instead of the recursive call *exp* n we use u to construct the code we'd like to return.

$$\begin{array}{lcl} \mathit{exp}\ Z & = & \mathbf{box}\ (\lambda b.\, S\, Z) \\ \mathit{exp}\ (S\, n) & = & \mathbf{case}\ (\mathit{exp}\ n)\ \{\mathbf{box}\ u \Rightarrow \mathbf{box}\ (\lambda b.\, b * (u\, b))\} \end{array}$$

Let's consider this code in action by computing *exp* $(S\ (S\ Z))$. Ideally, we might want something like

$$\mathit{exp}\ (S\ (S\ Z)) \mapsto^* \lambda b.\, b * b$$

but let's compute:

$$\begin{array}{l} \mathit{exp}\ (S\, (S\, Z)) \mapsto^* \mathbf{case}\ (\mathit{exp}\ (S\, Z))\{\mathbf{box}\ u \Rightarrow \mathbf{box}\ (\lambda b.\, b * (u\, b))\} \\ \mathit{exp}\ (S\, Z) \mapsto^* \mathbf{case}\ (\mathit{exp}\ Z)\{\mathbf{box}\ u \Rightarrow \mathbf{box}\ (\lambda b.\, b * (u\, b))\} \\ \mathit{exp}\ Z \mapsto^* \mathbf{box}\ (\lambda b.\, S\, Z) \end{array}$$

Substituting back (including some renaming) and continuing computation:

$$\begin{array}{l} \mathit{exp}\ (S\, Z) \mapsto^* \mathbf{case}\ \mathbf{box}\ (\lambda b_0.\, S\, Z)\{\mathbf{box}\ u \Rightarrow \mathbf{box}\ (\lambda b_1.\, b_1 * (u\, b_1))\} \\ \qquad\qquad \mapsto \mathbf{box}\ (\lambda b_1.\, b_1 * ((\lambda b_0.\, S\, Z)\, b_1)) \end{array}$$

And one more back-substitution:

$$\begin{aligned}
\mathit{exp}\ (S\,(S\,Z)) &\mapsto^* \textbf{case}\ (\mathit{exp}\ (S\,Z))\{\textbf{box}\ u \Rightarrow \textbf{box}\ (\lambda b_2.\, b_2 * (u\, b_2))\} \\
&\mapsto^* \textbf{case}\ \textbf{box}\ (\lambda b_1.\, b_1 * ((\lambda b_0.\, S\,Z)\, b_1))\ \{\textbf{box}\ u \Rightarrow \textbf{box}\ (\lambda b_2.\, b_2 * (\\
&\mapsto \textbf{box}\ (\lambda b_2.\, b_2 * ((\lambda b_1.\, b_1 * ((\lambda b_0.\, S\,Z)\, b_1))\, b_2))
\end{aligned}$$

This is not quite what we had hoped for. But we can perform a simple optimization, substituting variables for variables (noting that $(\lambda x.\, e)\, y \simeq [y/x]e$):

$$\mathit{exp}\ (S\,(S\,Z)) \mapsto^* v \quad \text{with } v \simeq \lambda b_2.\, b_2 * (b_2 * S\,Z)$$

We could eliminate the multiplication by 1 by introducing another case into the definition of the function

$$\begin{array}{lcl}
\mathit{exp}\ Z & = & \textbf{box}\ (\lambda b.\, S\,Z) \\
\mathit{exp}\,(S\,Z) & = & \textbf{box}\ (\lambda b.\, b) \\
\mathit{exp}\ (S\,(S\,n)) & = & \textbf{case}\ (\mathit{exp}\ (S\,n))\ \{\textbf{box}\ u \Rightarrow \textbf{box}\ (\lambda b.\, b * (u\, b))\}
\end{array}$$

But the variable for variable reduction is more difficult to eliminate. If we don't want to rely on the smarts of the compiler to perform this kind of inlining, we can further generalize the type $\Box\tau$ to $\Box_\Gamma\,\tau$ by allowing the free variables in Γ to appear in $e : \Box_\Gamma\,\tau$. This is a subject of *contextual modal types* [NPP08].

4 Evaluation

We have now seen an example of how we build a complex quoted expression. But how do we actually run it? For example, how do we compute 5^2 using the staged exponential function? We can define

$$\begin{array}{lcl}
\mathit{exp}' & : & \mathit{nat} \to \mathit{nat} \to \mathit{nat} \\
\mathit{exp}' & = & \lambda n.\, \lambda b.\, \textbf{case}\ \mathit{exp}\ n\ \{\textbf{box}\, u \Rightarrow u\, b\}
\end{array}$$

and then *pow* $\ulcorner 2 \urcorner\, \ulcorner 5 \urcorner \mapsto \ulcorner 25 \urcorner$.

We see that the *pow* function computes the quoted expression of type $\Box(\mathit{nat} \to \mathit{nat})$, binds u to the quoted function, and then applies that function. The way this differs from what we wrote in the definition of *exp* is that the expression variable u appears *outside* another **box** constructor. It is such an occurrence that causes the expression to be actually evaluated. In fact, we can define a polymorphic function (with parentheses in the type for emphasis):

$$\begin{array}{lcl}
\mathit{eval} & : & (\Box\alpha) \to \alpha \\
\mathit{eval} & = & \lambda x.\, \textbf{case}\ x\ \{\textbf{box}\ u \Rightarrow u\}
\end{array}$$

Critically, we can **not** define a (terminating) polymorphic function

$$\mathit{quote} : \alpha \to (\Box\alpha) \quad \text{(impossible)}$$

Intuitively, that's because we cannot complete

$$\mathit{quote} = \lambda x.\, \mathbf{box}\ ???$$

because underneath the **box** operator we cannot mention the value variable x. We see it is critical that **box** is a language primitive and not a *function*. This mirrors that fact that in modal logic we have an *inference rule* of necessitation

$$\frac{\vdash A}{\vdash \Box A}\ \text{NEC}$$

for an arbitrary proposition A but we cannot prove $\vdash A \supset \Box A$.

What else can we write? We can certainly quote a quoted expression. And we can take a quoted function and a quoted argument and synthesize a quoted application. We express these functions via pattern matching, but it should be clear how to decompose this into individual case expressions.

$$\begin{array}{l}
\mathit{eval} : \Box\alpha \to \alpha \\
\mathit{eval}\ (\mathbf{box}\ u) = u \\
\\
\mathit{quote} : \Box\alpha \to \Box\Box\alpha \\
\mathit{quote}\ (\mathbf{box}\ u) = \mathbf{box}\ (\mathbf{box}\ u) \\
\\
\mathit{apply} : \Box(\alpha \to \beta) \to \Box\alpha \to \Box\beta \\
\mathit{apply}\ (\mathbf{box}\ u)\ (\mathbf{box}\ w) = \mathbf{box}\ (u\, w)
\end{array}$$

If we view these types as axioms in a logic

$$\begin{array}{l}
\vdash \Box A \supset A \\
\vdash \Box A \supset \Box\Box A \\
\vdash \Box(A \supset B) \supset \Box A \supset \Box B
\end{array}$$

then together with the rule of necessitation these are characteristic of the intuitionistic modal logic S4 [PD01]. This is not an accident and we will elaborate further on the connection between logics and type systems in the next lecture.

One limitation: while pattern matching is convenient, we cannot match against the structure of expressions underneath the **box** constructor. Allowing this requires yet another big leap (see, for example, the work on

Beluga [PC15]). Not being able to do this allows us to implement runtime code generation efficiently, because we compile a value **box** e of type $\Box\,\tau$ to a *code generator* for e. Then substitution for expression variables $[\![e/u]\!]e'$ composes code generators, and using an expression variable u outside a **box** will finally call the code generator then jump to the code it produces (see, for example, [WLP98]).

5 Lifting

Not all programs can be restaged in the neat way of the exponentiation function, but there are many examples that work more or less well. Here are some hinted at that can be found in the literature:

$$\mathit{parse} : \mathit{grammar} \to \Box(\mathit{string} \to \mathit{tree}\ \textbf{option})$$

The staged version of a parser is a *parser generator* which takes a grammar and returns a parsing function from strings to parse trees (when they exist).

$$\mathit{mvmult} : \mathit{matrix} \to \Box(\mathit{vector} \to \mathit{vector})$$

The staged version that multiplies a matrix with a vector returns the source of a function that embodies the matrix values. This is generally a bad idea (the code could be very large) unless we know that the matrix is sparse. For spare matrices, however, it can pay off because we can eliminate multiplication by 0 and potentially get code that approximates the efficiency of specialized code for sparse matrix multiplication.

In general, in these example we sometimes have include *observable values* from one stage into the next stage, for example, integers. We recall from earlier that *purely positive types* have observable values. Ignoring universal and existential types, we have

$$\text{Purely Positive Types} \quad \tau^+ \quad ::= \quad 1 \mid \tau_1^+ \otimes \tau_2^+ \mid 0 \mid \tau_1^+ + \tau_2^+ \mid \rho\alpha^+.\,\tau^+ \mid \alpha^+$$

Also positive, but with a negative type underneath, is $(\Box\tau^-)^+$. For positive types, we can define functions by (nested) pattern matching, but not for negative types (which have the form $\tau_1^+ \to \tau_2^-$ and $\tau_1^- \mathbin{\&} \tau_2^-$). We can also define a family of functions

$$\mathit{lift}_{\tau^+} \quad : \quad \tau^+ \to \Box\tau^+$$

but it would be defined differently for different types τ^+. In other words, the *lift* function would not be parametric! However, when included as a primitive (justified because it is definable at every positive type) we may be able to rescue some parametricity property. As an example, we consider lifting natural numbers.

$$\begin{array}{lcl} \mathit{lift}_{nat} & : & \mathit{nat} \to \Box\, \mathit{nat} \\ \mathit{lift}_{nat}\; Z & = & \mathbf{box}\; Z \\ \mathit{lift}_{nat}\; (S\, n) & = & \mathbf{case}\; \mathit{lift}_{nat}\; n\; \{\mathbf{box}\; u \Rightarrow \mathbf{box}\; (S\, u)\} \end{array}$$

It is straightforward but tedious to translate this definition into one using only the language primitives directly.

References

[DP01] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, May 2001.

[Kis14] Oleg Kiselyov. The design and implementation of BER MetaOCaml. In M. Codish and E. Sumii, editors, *12th International Symposium on Functional and Logic Programming (FLOPS 2014)*, pages 86–102. Springer LNCS 8475, 2014.

[KWM$^+$08] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the java hotspot client compiler for java 6. *ACM Transactions on Architecture and Code Optimization*, 5(1):7:1–7:32, May 2008.

[LL98] Mark Leone and Peter Lee. Dynamic specialization in the Fabius system. *Computing Surveys*, 30(3es), 1998. Published electronically.

[NPP08] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *Transactions on Computational Logic*, 9(3), 2008.

[PC15] Brigitte Pientka and Andrew Cave. Inductive Beluga: Programming proofs. In A. Felty and A. Middeldorp, editors, *25th International Conference on Automated Deduction (CADE 2015)*, pages 272–281, Berlin, Germany, August 2015. Springer LNCS 9195.

[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.

[WLP98] Philip Wickline, Peter Lee, and Frank Pfenning. Run-time code generation and modal-ML. In Keith D. Cooper, editor, *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*, pages 224–235, Montreal, Canada, June 1998. ACM Press.

Lecture Notes on Propositions as Types

15-814: Types and Programming Languages
Frank Pfenning

Lecture 19
November 8, 2018

1 Introduction

> *These lecture notes are pieced together from several lectures in an undergraduate course on* Constructive Logic*, so they are a bit more extensive than what we discussed in the lecture.*

2 Natural Deduction

The goal of this section is to develop the two principal notions of logic, namely *propositions* and *proofs*. There is no universal agreement about the proper foundations for these notions. One approach, which has been particularly successful for applications in computer science, is to understand the meaning of a proposition by understanding its proofs. In the words of Martin-Löf [ML96, Page 27]:

> *The meaning of a proposition is determined by [...] what counts as a verification of it.*

A *verification* may be understood as a certain kind of proof that only examines the constituents of a proposition. This is analyzed in greater detail by Dummett [Dum91] although with less direct connection to computer science. The system of inference rules that arises from this point of view is *natural deduction*, first proposed by Gentzen [Gen35] and studied in depth by Prawitz [Pra65].

In this chapter we apply Martin-Löf's approach, which follows a rich philosophical tradition, to explain the basic propositional connectives.

We will define the meaning of the usual connectives of propositional logic (conjunction, implication, disjunction) by rules that allow us to infer when they should be true, so-called *introduction rules*. From these, we derive rules for the use of propositions, so-called *elimination rules*. The resulting system of *natural deduction* is the foundation of intuitionistic logic which has direct connections to functional programming and logic programming.

3 Judgments and Propositions

The cornerstone of Martin-Löf's foundation of logic is a clear separation of the notions of judgment and proposition. A *judgment* is something we may know, that is, an object of knowledge. A judgment is *evident* if we in fact know it.

We make a judgment such as "*it is raining*", because we have evidence for it. In everyday life, such evidence is often immediate: we may look out the window and see that it is raining. In logic, we are concerned with situation where the evidence is indirect: we deduce the judgment by making correct inferences from other evident judgments. In other words: a judgment is evident if we have a proof for it.

The most important judgment form in logic is "*A is true*", where A is a proposition. There are many others that have been studied extensively. For example, "*A is false*", "*A is true at time t*" (from temporal logic), "*A is necessarily true*" (from modal logic), "*program M has type τ*" (from programming languages), etc.

Returning to the first judgment, let us try to explain the meaning of conjunction. We write A *true* for the judgment "*A is true*" (presupposing that A is a proposition. Given propositions A and B, we can form the compound proposition "*A and B*", written more formally as $A \wedge B$. But we have not yet specified what conjunction *means*, that is, what counts as a verification of $A \wedge B$. This is accomplished by the following inference rule:

$$\frac{A\ \mathit{true} \quad B\ \mathit{true}}{A \wedge B\ \mathit{true}}\ \wedge I$$

Here the name $\wedge I$ stands for "conjunction introduction", since the conjunction is introduced in the conclusion.

This rule allows us to conclude that $A \wedge B$ *true* if we already know that A *true* and B *true*. In this inference rule, A and B are *schematic variables*, and $\wedge I$ is the name of the rule. Intuitively, the $\wedge I$ rule says that a proof of $A \wedge B$ *true* consists of a proof of A *true* together with a proof of B *true*.

The general form of an inference rule is

$$\frac{J_1 \ \ldots \ J_n}{J} \ \mathit{name}$$

where the judgments $J_1, \ldots, J_n$ are called the *premises*, the judgment J is called the *conclusion*. In general, we will use letters J to stand for judgments, while A, B, and C are reserved for propositions.

We take conjunction introduction as specifying the meaning of $A \wedge B$ completely. So what can be deduced if we know that $A \wedge B$ is true? By the above rule, to have a verification for $A \wedge B$ means to have verifications for A and B. Hence the following two rules are justified:

$$\frac{A \wedge B \ \mathit{true}}{A \ \mathit{true}} \ \wedge E_1 \qquad\qquad \frac{A \wedge B \ \mathit{true}}{B \ \mathit{true}} \ \wedge E_2$$

The name $\wedge E_1$ stands for "first/left conjunction elimination", since the conjunction in the premise has been eliminated in the conclusion. Similarly $\wedge E_2$ stands for "second/right conjunction elimination". Intuitively, the $\wedge E_1$ rule says that A *true* follows if we have a proof of $A \wedge B$ *true*, because "we must have had a proof of A *true* to justify $A \wedge B$ *true*".

We will later see what precisely is required in order to guarantee that the formation, introduction, and elimination rules for a connective fit together correctly. For now, we will informally argue the correctness of the elimination rules, as we did for the conjunction elimination rules.

As a second example we consider the proposition "*truth*" written as $\top$. Truth should always be true, which means its introduction rule has no premises.

$$\frac{}{\top \ \mathit{true}} \ \top I$$

Consequently, we have no information if we know $\top$ *true*, so there is no elimination rule.

A conjunction of two propositions is characterized by one introduction rule with two premises, and two corresponding elimination rules. We may think of truth as a conjunction of zero propositions. By analogy it should then have one introduction rule with zero premises, and zero corresponding elimination rules. This is precisely what we wrote out above.

4 Hypothetical Judgments

Consider the following derivation, for arbitrary propositions A, B, and C:

$$\cfrac{\cfrac{A \wedge (B \wedge C)\ \mathit{true}}{B \wedge C\ \mathit{true}}\ \wedge E_2}{B\ \mathit{true}}\ \wedge E_1$$

Have we actually proved anything here? At first glance it seems that cannot be the case: B is an arbitrary proposition; clearly we should not be able to prove that it is true. Upon closer inspection we see that all inferences are correct, but the first judgment $A \wedge (B \wedge C)$ *true* has not been justified. We can extract the following knowledge:

> *From the assumption that $A \wedge (B \wedge C)$ is true, we deduce that B must be true.*

This is an example of a *hypothetical judgment*, and the figure above is an *hypothetical deduction*. In general, we may have more than one assumption, so a hypothetical deduction has the form

$$\begin{array}{ccc} J_1 & \cdots & J_n \\ & \vdots & \\ & J & \end{array}$$

where the judgments $J_1, \ldots, J_n$ are unproven assumptions, and the judgment J is the conclusion. All instances of the inference rules are hypothetical judgments as well (albeit possibly with 0 assumptions if the inference rule has no premises).

Many mistakes in reasoning arise because dependencies on some hidden assumptions are ignored. When we need to be explicit, we will write $J_1, \ldots, J_n \vdash J$ for the hypothetical judgment which is established by the hypothetical deduction above. We may refer to $J_1, \ldots, J_n$ as the antecedents and J as the succedent of the hypothetical judgment. For example, the hypothetical judgment $A \wedge (B \wedge C)$ *true* $\vdash B$ *true* is proved by the above hypothetical deduction that B *true* indeed follows from the hypothesis $A \wedge (B \wedge C)$ *true* using inference rules.

Substitution Principle for Hypotheses: We can always substitute a proof for any hypothesis J_i to eliminate the assumption. Into the above hypothetical deduction, a proof of its hypothesis J_i

$$\begin{array}{ccc} K_1 & \cdots & K_m \\ & \vdots & \\ & J_i & \end{array}$$

can be substituted in for J_i to obtain the hypothetical deduction

$$\begin{array}{ccccc} & & K_1 \quad \cdots \quad K_m & & \\ & & \vdots & & \\ J_1 \quad \cdots & & J_i & & \cdots \quad J_n \\ & & \vdots & & \\ & & J & & \end{array}$$

This hypothetical deduction concludes J from the unproven assumptions $J_1, \ldots, J_{i-1}, K_1, \ldots, K_m, J_{i+1}, \ldots, J_n$ and justifies the hypothetical judgment

$$J_1, \ldots, J_{i-1}, K_1, \ldots, K_m, J_{i+1}, \ldots, J_n \vdash J$$

That is, into the hypothetical judgment $J_1, \ldots, J_n \vdash J$, we can always substitute a derivation of the judgment J_i that was used as a hypothesis to obtain a derivation which no longer depends on the assumption J_i. A hypothetical deduction with 0 assumptions is a *proof* of its conclusion J.

One has to keep in mind that hypotheses may be used more than once, or not at all. For example, for arbitrary propositions A and B,

$$\dfrac{\dfrac{A \wedge B\ \mathit{true}}{B\ \mathit{true}}\wedge E_2 \qquad \dfrac{A \wedge B\ \mathit{true}}{A\ \mathit{true}}\wedge E_1}{B \wedge A\ \mathit{true}}\wedge I$$

can be seen a hypothetical derivation of $A \wedge B\ \mathit{true} \vdash B \wedge A\ \mathit{true}$. Similarly, a minor variation of the first proof in this section is a hypothetical derivation for the hypothetical judgment $A \wedge (B \wedge C)\ \mathit{true} \vdash B \wedge A\ \mathit{true}$ that uses the hypothesis twice.

With hypothetical judgments, we can now explain the meaning of implication "*A implies B*" or "*if A then B*" (more formally: $A \supset B$). The introduction rule reads: $A \supset B$ is true, if B is true under the assumption that A is true.

$$\dfrac{\begin{array}{c} \overline{A\ \mathit{true}}\ u \\ \vdots \\ B\ \mathit{true} \end{array}}{A \supset B\ \mathit{true}} \supset I^u$$

The tricky part of this rule is the label u and its bar. If we omit this annotation, the rule would read

$$\dfrac{\begin{array}{c} A\ \mathit{true} \\ \vdots \\ B\ \mathit{true} \end{array}}{A \supset B\ \mathit{true}} \supset I$$

which would be incorrect: it looks like a derivation of $A \supset B$ *true* from the hypothesis A *true*. But the assumption A *true* is introduced in the process of proving $A \supset B$ *true*; the conclusion should not depend on it! Certainly, whether the implication $A \supset B$ is true is independent of the question whether A itself is actually true. Therefore we label uses of the assumption with a new name u, and the corresponding inference which introduced this assumption into the derivation with the same label u.

The rule makes intuitive sense, a proof justifying $A \supset B$ *true* assumes, hypothetically, the left-hand side of the implication so that A *true*, and uses this to show the right-hand side of the implication by proving B *true*. The proof of $A \supset B$ *true* constructs a proof of B *true* from the additional assumption that A *true*.

As a concrete example, consider the following proof of $A \supset (B \supset (A \wedge B))$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A\ \mathit{true}}\,u \qquad \dfrac{}{B\ \mathit{true}}\,w}{A \wedge B\ \mathit{true}}\wedge I}{B \supset (A \wedge B)\ \mathit{true}}\supset I^{w}}{A \supset (B \supset (A \wedge B))\ \mathit{true}}\supset I^{u}$$

Note that this derivation is not hypothetical (it does not depend on any assumptions). The assumption A *true* labeled u is discharged in the last inference, and the assumption B *true* labeled w is discharged in the second-to-last inference. It is critical that a discharged hypothesis is no longer available for reasoning, and that all labels introduced in a derivation are distinct.

Finally, we consider what the elimination rule for implication should say. By the only introduction rule, having a proof of $A \supset B$ *true* means that we have a hypothetical proof of B *true* from A *true*. By the substitution principle, if we also have a proof of A *true* then we get a proof of B *true*.

$$\dfrac{A \supset B\ \mathit{true} \qquad A\ \mathit{true}}{B\ \mathit{true}}\supset E$$

This completes the rules concerning implication.

With the rules so far, we can write out proofs of simple properties concerning conjunction and implication. The first expresses that conjunction is commutative—intuitively, an obvious property.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge B\ \mathit{true}}\,u}{B\ \mathit{true}}\,{\wedge}E_2 \qquad \dfrac{\dfrac{}{A \wedge B\ \mathit{true}}\,u}{A\ \mathit{true}}\,{\wedge}E_1}{B \wedge A\ \mathit{true}}\,{\wedge}I}{(A \wedge B) \supset (B \wedge A)\ \mathit{true}}\,{\supset}I^u$$

When we construct such a derivation, we generally proceed by a combination of bottom-up and top-down reasoning. The next example is a distributivity law, allowing us to move implications over conjunctions. This time, we show the partial proofs in each step. Of course, other sequences of steps in proof constructions are also possible.

$$\begin{array}{c} \vdots \\ (A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C))\ \mathit{true} \end{array}$$

First, we use the implication introduction rule bottom-up.

$$\dfrac{\begin{array}{c} \dfrac{}{A \supset (B \wedge C)\ \mathit{true}}\,u \\ \vdots \\ (A \supset B) \wedge (A \supset C)\ \mathit{true} \end{array}}{(A \supset (B \wedge C) \supset ((A \supset B) \wedge (A \supset C))\ \mathit{true}}\,{\supset}I^u$$

Next, we use the conjunction introduction rule bottom-up, copying the available assumptions to both branches in the scope.

$$\dfrac{\dfrac{\begin{array}{c} \dfrac{}{A \supset (B \wedge C)\ \mathit{true}}\,u \\ \vdots \\ A \supset B\ \mathit{true} \end{array} \qquad \begin{array}{c} \dfrac{}{A \supset (B \wedge C)\ \mathit{true}}\,u \\ \vdots \\ A \supset C\ \mathit{true} \end{array}}{(A \supset B) \wedge (A \supset C)\ \mathit{true}}\,{\wedge}I}{(A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C))\ \mathit{true}}\,{\supset}I^u$$

We now pursue the left branch, again using implication introduction bottom-up.

$$
\dfrac{
\dfrac{
\dfrac{
\begin{array}{c}
\dfrac{}{A \supset (B \wedge C)\ \mathit{true}}\,u \quad \dfrac{}{A\ \mathit{true}}\,w \\
\vdots \\
B\ \mathit{true}
\end{array}
}{A \supset B\ \mathit{true}}\,{\supset}I^w
\qquad
\begin{array}{c}
\dfrac{}{A \supset (B \wedge C)\ \mathit{true}}\,u \\
\vdots \\
A \supset C\ \mathit{true}
\end{array}
}{(A \supset B) \wedge (A \supset C)\ \mathit{true}}\,{\wedge}I
}{(A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C))\ \mathit{true}}\,{\supset}I^u
$$

Note that the hypothesis *A true* is available only in the left branch and not in the right one: it is discharged at the inference ${\supset}I^w$. We now switch to top-down reasoning, taking advantage of implication elimination.

$$
\dfrac{
\dfrac{
\dfrac{
\begin{array}{c}
\dfrac{\dfrac{}{A \supset (B \wedge C)\ \mathit{true}}\,u \quad \dfrac{}{A\ \mathit{true}}\,w}{B \wedge C\ \mathit{true}}\,{\supset}E \\
\vdots \\
B\ \mathit{true}
\end{array}
}{A \supset B\ \mathit{true}}\,{\supset}I^w
\qquad
\begin{array}{c}
\dfrac{}{A \supset (B \wedge C)\ \mathit{true}}\,u \\
\vdots \\
A \supset C\ \mathit{true}
\end{array}
}{(A \supset B) \wedge (A \supset C)\ \mathit{true}}\,{\wedge}I
}{(A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C))\ \mathit{true}}\,{\supset}I^u
$$

Now we can close the gap in the left-hand side by conjunction elimination.

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{\dfrac{}{A \supset (B \wedge C)\ \mathit{true}}\,u \quad \dfrac{}{A\ \mathit{true}}\,w}{B \wedge C\ \mathit{true}}\,{\supset}E
}{B\ \mathit{true}}\,{\wedge}E_1
}{A \supset B\ \mathit{true}}\,{\supset}I^w
\qquad
\begin{array}{c}
\dfrac{}{A \supset (B \wedge C)\ \mathit{true}}\,u \\
\vdots \\
A \supset C\ \mathit{true}
\end{array}
}{(A \supset B) \wedge (A \supset C)\ \mathit{true}}\,{\wedge}I
}{(A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C))\ \mathit{true}}\,{\supset}I^u
$$

The right premise of the conjunction introduction can be filled in analogously. We skip the intermediate steps and only show the final derivation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \supset (B \wedge C)\ \mathit{true}}\,u \quad \dfrac{}{A\ \mathit{true}}\,w}{B \wedge C\ \mathit{true}}\,{\supset}E}{B\ \mathit{true}}\,{\wedge}E_1}{A \supset B\ \mathit{true}}\,{\supset}I^w \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A \supset (B \wedge C)\ \mathit{true}}\,u \quad \dfrac{}{A\ \mathit{true}}\,v}{B \wedge C\ \mathit{true}}\,{\supset}E}{C\ \mathit{true}}\,{\wedge}E_2}{A \supset C\ \mathit{true}}\,{\supset}I^v}{(A \supset B) \wedge (A \supset C)\ \mathit{true}}\,{\wedge}I}{(A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C))\ \mathit{true}}\,{\supset}I^u$$

5 Disjunction and Falsehood

So far we have explained the meaning of conjunction, truth, and implication. The disjunction "A *or* B" (written as $A \vee B$) is more difficult, but does not require any new judgment forms. Disjunction is characterized by two introduction rules: $A \vee B$ is true, if either A or B is true.

$$\dfrac{A\ \mathit{true}}{A \vee B\ \mathit{true}}\,{\vee}I_1 \qquad \dfrac{B\ \mathit{true}}{A \vee B\ \mathit{true}}\,{\vee}I_2$$

Now it would be incorrect to have an elimination rule such as

$$\dfrac{A \vee B\ \mathit{true}}{A\ \mathit{true}}\,{\vee}E_1?$$

because even if we know that $A \vee B$ is true, we do not know whether the disjunct A or the disjunct B is true. Concretely, with such a rule we could derive the truth of *every* proposition A as follows:

$$\dfrac{\dfrac{\dfrac{}{\top\ \mathit{true}}\,{\top}I}{A \vee \top\ \mathit{true}}\,{\vee}I_2}{A\ \mathit{true}}\,{\vee}E_1?$$

Thus we take a different approach. If we know that $A \vee B$ is true, we must consider two cases: A *true* and B *true*. If we can prove a conclusion C *true* in both cases, then C must be true! Written as an inference rule:

$$\dfrac{A \vee B\ \mathit{true} \qquad \begin{array}{c}\dfrac{}{A\ \mathit{true}}\,u \\ \vdots \\ C\ \mathit{true}\end{array} \qquad \begin{array}{c}\dfrac{}{B\ \mathit{true}}\,w \\ \vdots \\ C\ \mathit{true}\end{array}}{C\ \mathit{true}}\,{\vee}E^{u,w}$$

If we know that $A \lor B$ *true* then we also know C *true*, if that follows both in the case where $A \lor B$ *true* because A is true and in the case where $A \lor B$ *true* because B is true. Note that we use once again the mechanism of hypothetical judgments. In the proof of the second premise we may use the assumption A *true* labeled u, in the proof of the third premise we may use the assumption B *true* labeled w. Both are discharged at the disjunction elimination rule.

Let us justify the conclusion of this rule more explicitly. By the first premise we know $A \lor B$ *true*. The premises of the two possible introduction rules are A *true* and B *true*. In case A *true* we conclude C *true* by the substitution principle and the second premise: we substitute the proof of $A\ \mathit{true}$ for any use of the assumption labeled u in the hypothetical derivation. The case for B *true* is symmetric, using the hypothetical derivation in the third premise.

Because of the complex nature of the elimination rule, reasoning with disjunction is more difficult than with implication and conjunction. As a simple example, we prove the commutativity of disjunction.

$$\begin{array}{c} \vdots \\ (A \lor B) \supset (B \lor A)\ \mathit{true} \end{array}$$

We begin with an implication introduction.

$$\dfrac{\begin{array}{c} \dfrac{}{A \lor B\ \mathit{true}}\,u \\ \vdots \\ B \lor A\ \mathit{true} \end{array}}{(A \lor B) \supset (B \lor A)\ \mathit{true}}\,{\supset}I^u$$

At this point we cannot use either of the two disjunction introduction rules. The problem is that neither B nor A follow from our assumption $A \lor B$! So first we need to distinguish the two cases via the rule of disjunction elimination.

$$\dfrac{\dfrac{\dfrac{}{A \lor B\ \mathit{true}}\,u \quad \begin{array}{c} \dfrac{}{A\ \mathit{true}}\,v \\ \vdots \\ B \lor A\ \mathit{true} \end{array} \quad \begin{array}{c} \dfrac{}{B\ \mathit{true}}\,w \\ \vdots \\ B \lor A\ \mathit{true} \end{array}}{B \lor A\ \mathit{true}}\,{\lor}E^{v,w}}{(A \lor B) \supset (B \lor A)\ \mathit{true}}\,{\supset}I^u$$

The assumption labeled u is still available for each of the two proof obligations, but we have omitted it, since it is no longer needed.

Now each gap can be filled in directly by the two disjunction introduction rules.

$$\dfrac{\dfrac{\dfrac{}{A \lor B\ \mathit{true}}\ u \qquad \dfrac{\dfrac{}{A\ \mathit{true}}\ v}{B \lor A\ \mathit{true}}\ {\lor}I_2 \qquad \dfrac{\dfrac{}{B\ \mathit{true}}\ w}{B \lor A\ \mathit{true}}\ {\lor}I_1}{B \lor A\ \mathit{true}}\ {\lor}E^{v,w}}{(A \lor B) \supset (B \lor A)\ \mathit{true}}\ {\supset}I^u$$

This concludes the discussion of disjunction. Falsehood (written as $\bot$, sometimes called absurdity) is a proposition that should have no proof! Therefore there are no introduction rules.

Since there cannot be a proof of $\bot$ *true*, it is sound to conclude the truth of any arbitrary proposition if we know $\bot$ *true*. This justifies the elimination rule

$$\dfrac{\bot\ \mathit{true}}{C\ \mathit{true}}\ {\bot}E$$

We can also think of falsehood as a disjunction between zero alternatives. By analogy with the binary disjunction, we therefore have zero introduction rules, and an elimination rule in which we have to consider zero cases. This is precisely the $\bot E$ rule above.

From this is might seem that falsehood it useless: we can never prove it. This is correct, except that we might reason from contradictory hypotheses! We will see some examples when we discuss negation, since we may think of the proposition "*not* A" (written $\neg A$) as $A \supset \bot$. In other words, $\neg A$ is true precisely if the assumption A *true* is contradictory because we could derive $\bot$ *true*.

6 Summary of Natural Deduction

The judgments, propositions, and inference rules we have defined so far collectively form a system of *natural deduction*. It is a minor variant of a system introduced by Gentzen [Gen35] and studied in depth by Prawitz [Pra65]. One of Gentzen's main motivations was to devise rules that model mathematical reasoning as directly as possible, although clearly in much more detail than in a typical mathematical argument.

The specific interpretation of the truth judgment underlying these rules is *intuitionistic* or *constructive*. This differs from the *classical* or *Boolean* interpretation of truth. For example, classical logic accepts the proposition $A \lor (A \supset B)$ as true for arbitrary A and B, although in the system we have presented so far this would have no proof. Classical logic is based on the

Introduction Rules

$$\frac{A\ \mathit{true} \quad B\ \mathit{true}}{A \wedge B\ \mathit{true}}\ \wedge I$$

$$\frac{}{\top\ \mathit{true}}\ \top I$$

$$\frac{\begin{array}{c}\overline{A\ \mathit{true}}\ u\\ \vdots\\ B\ \mathit{true}\end{array}}{A \supset B\ \mathit{true}}\ \supset I^{u}$$

$$\frac{A\ \mathit{true}}{A \vee B\ \mathit{true}}\ \vee I_1 \qquad \frac{B\ \mathit{true}}{A \vee B\ \mathit{true}}\ \vee I_2$$

no $\bot I$ *rule*

Elimination Rules

$$\frac{A \wedge B\ \mathit{true}}{A\ \mathit{true}}\ \wedge E_1 \qquad \frac{A \wedge B\ \mathit{true}}{B\ \mathit{true}}\ \wedge E_2$$

no $\top E$ *rule*

$$\frac{A \supset B\ \mathit{true} \quad A\ \mathit{true}}{B\ \mathit{true}}\ \supset E$$

$$\frac{A \vee B\ \mathit{true} \quad \begin{array}{c}\overline{A\ \mathit{true}}\ u\\ \vdots\\ C\ \mathit{true}\end{array} \quad \begin{array}{c}\overline{B\ \mathit{true}}\ w\\ \vdots\\ C\ \mathit{true}\end{array}}{C\ \mathit{true}}\ \vee E^{u,w}$$

$$\frac{\bot\ \mathit{true}}{C\ \mathit{true}}\ \bot E$$

Figure 1: Rules for intuitionistic natural deduction

principle that every proposition must be true or false. If we distinguish these cases we see that $A \vee (A \supset B)$ should be accepted, because in case that A is true, the left disjunct holds; in case A is false, the right disjunct holds. In contrast, intuitionistic logic is based on explicit evidence, and evidence for a disjunction requires evidence for one of the disjuncts. We will return to classical logic and its relationship to intuitionistic logic later; for now our reasoning remains intuitionistic since, as we will see, it has a direct connection to functional computation, which classical logic lacks.

We summarize the rules of inference for the truth judgment introduced so far in Figure 1.

7 Propositions as Types

We now investigate a computational interpretation of constructive proofs and relate it to functional programming. On the propositional fragment of logic this is called the Curry-Howard isomorphism [How80]. From the very outset of the development of constructive logic and mathematics, a central idea has been that *proofs ought to represent constructions*. The Curry-Howard isomorphism is only a particularly poignant and beautiful realization of this idea. In a highly influential subsequent paper, Per Martin-Löf [ML80] developed it further into a more expressive calculus called *type theory*.

In order to illustrate the relationship between proofs and programs we introduce a new judgment:

$M : A \qquad M$ is a proof term for proposition A

We presuppose that A is a proposition when we write this judgment. We will also interpret $M : A$ as "*M is a program of type A*". These dual interpretations of the same judgment is the core of the Curry-Howard isomorphism. We either think of M as a syntactic term that represents the proof of A *true*, or we think of A as the type of the program M. As we discuss each connective, we give both readings of the rules to emphasize the analogy.

We intend that if $M : A$ then A *true*. Conversely, if A *true* then $M : A$ for some appropriate proof term M. But we want something more: every deduction of $M : A$ should correspond to a deduction of A *true* with an identical structure and vice versa. In other words we annotate the inference rules of natural deduction with proof terms. The property above should then be obvious. In that way, proof term M of $M : A$ will correspond directly to the corresponding proof of A *true*.

Conjunction. Constructively, we think of a proof of $A \wedge B$ *true* as a pair of proofs: one for A *true* and one for B *true*. So if M is a proof of A and N is a proof of B, then the pair $\langle\!| M, N |\!\rangle$ is a proof of $A \wedge B$.

$$\frac{M : A \quad N : B}{\langle\!| M, N |\!\rangle : A \wedge B} \wedge I$$

The elimination rules correspond to the projections from a pair to its first and second elements to get the individual proofs back out from a pair M.

$$\frac{M : A \wedge B}{M \cdot l : A} \wedge E_1 \qquad \frac{M : A \wedge B}{M \cdot r : B} \wedge E_2$$

Hence the conjunction $A \wedge B$ proposition corresponds to the (lazy) product type $A \mathbin{\&} B$. And, indeed, product types in functional programming languages have the same property that conjunction propositions $A \wedge B$ have. Constructing a pair $\langle\!| M, N |\!\rangle$ of type $A \mathbin{\&} B$ requires a program M of type A and a program N of type B (as in $\wedge I$). Given a pair M of type $A \mathbin{\&} B$, its first component of type A can be retrieved by the projection $M \cdot l$ (as in $\wedge E_1$), its second component of type B by the projection $M \cdot r$ (as in $\wedge E_2$).

Truth. Constructively, we think of a proof of $\top$ *true* as a unit element that carries no information.

$$\frac{}{\langle\!|\,|\!\rangle : \top}\ \top I$$

Hence $\top$ corresponds to the (lazy) unit type with one element that we haven't encountered yet explicity, but is the nullary version of the product $\& \{\}$. There is no elimination rule and hence no further proof term constructs for truth. Indeed, we have not put any information into $\langle\,\rangle$ when constructing it via $\top I$, so cannot expect to get any information back out when trying to eliminate it.

Implication. Constructively, we think of a proof of $A \supset B$ *true* as a function which transforms a proof of A *true* into a proof of B *true*.

We now use the notation of λ-abstraction to annotate the rule of implication introduction with proof terms.

$$\frac{\begin{array}{c}\overline{u : A}\ u \\ \vdots \\ M : B\end{array}}{\lambda u.\, M : A \supset B}\ \supset I^u$$

The hypothesis label u acts as a variable, and any use of the hypothesis labeled u in the proof of B corresponds to an occurrence of u in M. Notice how a constructive proof of B *true* from the additional assumption A *true* to establish $A \supset B$ *true* also describes the transformation of a proof of A *true* to a proof of B *true*. But the proof term $\lambda u.\, M$ explicitly represents this transformation syntactically as a function, instead of leaving this construction implicit by inspection of whatever the proof does.

As a concrete example, consider the (trivial) proof of $A \supset A$ *true*:

$$\dfrac{\dfrac{}{A\ \mathit{true}}\ u}{A \supset A\ \mathit{true}}\ {\supset}I^u$$

If we annotate the deduction with proof terms, we obtain

$$\dfrac{\dfrac{}{u : A}\ u}{(\lambda u.\, u) : A \supset A}\ {\supset}I^u$$

So our proof corresponds to the identity function *id* at type A which simply returns its argument. It can be defined with the identity function $\mathit{id}(u) = u$ or $\mathrm{id} = (\lambda u.\, u)$.

Constructively, a proof of $A \supset B$ *true* is a function transforming a proof of A *true* to a proof of B *true*. Using $A \supset B$ *true* by its elimination rule ${\supset}E$, thus, corresponds to providing the proof of A *true* that $A \supset B$ *true* is waiting for to obtain a proof of B *true*. The rule for implication elimination corresponds to function application.

$$\dfrac{M : A \supset B \quad N : A}{M\, N : B}\ {\supset}E$$

What is the meaning of $A \supset B$ as a type? From the discussion above it should be clear that it can be interpreted as a function type $A \to B$. The introduction and elimination rules for implication can also be viewed as formation rules for functional abstraction $\lambda u.\, M$ and application $M\, N$. Forming a functional abstraction $\lambda u.\, M$ corresponds to a function that accepts input parameter u of type A and produces M of type B (as in ${\supset}I$). Using a function $M : A \to B$ corresponds to applying it to a concrete input argument N of type A to obtain an output $M\, N$ of type B.

Note that we obtain the usual introduction and elimination rules for implication if we erase the proof terms. This will continue to be true for all rules in the remainder of this section and is immediate evidence for the soundness of the proof term calculus, that is, if $M : A$ then A *true*.

As a second example we consider a proof of $(A \wedge B) \supset (B \wedge A)$ *true*.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge B\ \mathit{true}}\ u}{B\ \mathit{true}}\ {\wedge}E_2 \qquad \dfrac{\dfrac{}{A \wedge B\ \mathit{true}}\ u}{A\ \mathit{true}}\ {\wedge}E_1}{B \wedge A\ \mathit{true}}\ {\wedge}I}{(A \wedge B) \supset (B \wedge A)\ \mathit{true}}\ {\supset}I^u$$

When we annotate this derivation with proof terms, we obtain the swap function which takes a pair $\langle M, N\rangle$ and returns the reverse pair $\langle N, M\rangle$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{u : A \wedge B}\,u}{u \cdot r : B}\,\wedge E_2 \qquad \dfrac{\dfrac{}{u : A \wedge B}\,u}{u \cdot l : A}\,\wedge E_1}{\llparenthesis u \cdot r, u \cdot l \rrparenthesis : B \wedge A}\,\wedge I}{(\lambda u.\, \llparenthesis u \cdot r, u \cdot l \rrparenthesis) : (A \wedge B) \supset (B \wedge A)}\,\supset I^u$$

Disjunction. Constructively, we think of a proof of $A \vee B$ *true* as either a proof of A *true* or B *true*. Disjunction therefore corresponds to a disjoint sum type $A + B$ that either store something of type A or something of type B. The two introduction rules correspond to the left and right injection into a sum type.

$$\dfrac{M : A}{l \cdot M : A \vee B}\,\vee I_1 \qquad \dfrac{N : B}{r \cdot N : A \vee B}\,\vee I_2$$

When using a disjunction $A \vee B$ *true* in a proof, we need to be prepared to handle A *true* as well as B *true*, because we don't know whether $\vee I_1$ or $\vee I_2$ was used to prove it. The elimination rule corresponds to a case construct which discriminates between a left and right injection into a sum types.

$$\dfrac{M : A \vee B \qquad \begin{array}{c} \dfrac{}{u : A}\,u \\ \vdots \\ N : C \end{array} \qquad \begin{array}{c} \dfrac{}{w : B}\,w \\ \vdots \\ P : C \end{array}}{\textbf{case}\ M\ \{l \cdot u \Rightarrow N \mid r \cdot w \Rightarrow P\} : C}\,\vee E^{u,w}$$

Recall that the hypothesis labeled u is available only in the proof of the second premise and the hypothesis labeled w only in the proof of the third premise. This means that the scope of the variable u is N, while the scope of the variable w is P.

Falsehood. There is no introduction rule for falsehood ($\bot$). We can therefore view it as the empty type $\mathbf{0}$. The corresponding elimination rule allows a term of $\bot$ to stand for an expression of any type when wrapped in a case with no alternatives. There can be no valid reduction rule for falsehood, which means during computation of a valid program we will never try to evaluate a term of the form **case** $M\,\{\,\}$.

$$\dfrac{M : \bot}{\textbf{case}\ M\,\{\,\} : C}\,\bot E$$

Interaction Laws. This completes our assignment of proof terms to the logical inference rules. Now we can interpret the interaction laws we introduced early as programming exercises. Consider the following distributivity law:

(L11a) $(A \supset (B \wedge C)) \supset (A \supset B) \wedge (A \supset C)$ *true*

Interpreted constructively, this assignment can be read as:

> Write a function which, when given a function from A to pairs of type $B \wedge C$, returns two functions: one which maps A to B and one which maps A to C.

This is satisfied by the following function:

$$\lambda u.\, \llparenthesis(\lambda w.\,(u\,w)\cdot l), (\lambda v.\,(u\,v)\cdot r)\rrparenthesis$$

The following deduction provides the evidence:

$$
\cfrac{
  \cfrac{
    \cfrac{
      \cfrac{
        \cfrac{
          \cfrac{}{u : A \supset (B \wedge C)}\,u
          \qquad
          \cfrac{}{w : A}\,w
        }{u\,w : B \wedge C}\,{\supset}E
      }{(u\,w)\cdot l : B}\,{\wedge}E_1
    }{\lambda w.\,(u\,w)\cdot l : A \supset B}\,{\supset}I^w
    \qquad
    \cfrac{
      \cfrac{
        \cfrac{
          \cfrac{}{u : A \supset (B \wedge C)}\,u
          \qquad
          \cfrac{}{v : A}\,v
        }{u\,v : B \wedge C}\,{\supset}E
      }{(u\,v)\cdot r : C}\,{\wedge}E_2
    }{\lambda v.\,(u\,v)\cdot r : A \supset C}\,{\supset}I^v
  }{\llparenthesis(\lambda w.\,(u\,w)\cdot l), (\lambda v.\,(u\,v)\cdot r)\rrparenthesis : (A \supset B) \wedge (A \supset C)}\,{\wedge}I
}{\lambda u.\, \llparenthesis(\lambda w.\,(u\,w)\cdot l), (\lambda v.\,(u\,v)\cdot r)\rrparenthesis : (A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C))}\,{\supset}I^u
$$

Programs in constructive propositional logic are somewhat uninteresting in that they do not manipulate basic data types such as natural numbers, integers, lists, trees, etc. We introduce such data types later in this course, following the same method we have used in the development of logic.

Summary. To close this section we recall the *guiding principles behind the assignment of proof terms to deductions*.

1. For every deduction of A *true* there is a proof term M and deduction of $M : A$.

2. For every deduction of $M : A$ there is a deduction of A *true*

3. The correspondence between proof terms M and deductions of A *true* is a bijection.

8 Reduction

In the preceding section, we have introduced the assignment of proof terms to natural deductions. If proofs are programs then we need to explain how proofs are to be executed, and which results may be returned by a computation.

We explain the operational interpretation of proofs in two steps. In the first step we introduce a judgment of *reduction* written $M \longrightarrow M'$ and read "*M reduces to M'*". In the second step, a computation then proceeds by a sequence of reductions $M \longrightarrow M_1 \longrightarrow M_2 \ldots$, according to a fixed strategy, until we reach a value which is the result of the computation.

As in the development of propositional logic, we discuss each of the connectives separately, taking care to make sure the explanations are independent. This means we can consider various sublanguages and we can later extend our logic or programming language without invalidating the results from this section. Furthermore, it greatly simplifies the analysis of properties of the reduction rules.

In general, we think of the proof terms corresponding to the introduction rules as the *constructors* and the proof terms corresponding to the elimination rules as the *destructors*.

Conjunction. The constructor forms a pair, while the destructors are the left and right projections. The reduction rules prescribe the actions of the projections.

$$\begin{array}{rcl} \llparenthesis M, N \rrparenthesis \cdot l & \longrightarrow & M \\ \llparenthesis M, N \rrparenthesis \cdot r & \longrightarrow & N \end{array}$$

These (computational) reduction rules directly corresponds to the proof term analogue of the logical reductions for the local soundness detailed in Section 11. For example:

$$\dfrac{\dfrac{M : A \quad N : B}{\llparenthesis M, N \rrparenthesis : A \wedge B}\,\wedge I}{\llparenthesis M, N \rrparenthesis \cdot l : A}\,\wedge E_1 \quad \longrightarrow \quad M : A$$

Truth. The constructor just forms the unit element, $\langle\,\rangle$. Since there is no destructor, there is no reduction rule.

Implication. The constructor forms a function by λ-abstraction, while the destructor applies the function to an argument. The notation for the substitution of N for occurrences of u in M is $[N/u]M$. We therefore write the reduction rule as

$$(\lambda u.\, M)\, N \quad \longrightarrow \quad [N/u]M$$

We have to be somewhat careful so that substitution behaves correctly. In particular, no variable in N should be bound in M in order to avoid conflict. We can always achieve this by renaming bound variables—an operation which clearly does not change the meaning of a proof term. Again, this computational reduction directly relates to the logical reduction from the local soundness using the substitution notation for the right-hand side:

$$\dfrac{\dfrac{\begin{array}{c}\overline{u : A}\;u\\ \vdots \\ M : B\end{array}}{\lambda u.\, M : A \supset B}\;{\supset}I^u \qquad N : A}{(\lambda u.\, M)\, N : B}\;{\supset}E \qquad \longrightarrow \qquad [N/u]M$$

Disjunction. The constructors inject into a sum types; the destructor distinguishes cases. We need to use substitution again.

$$\begin{array}{rcl}\textbf{case}\ l \cdot M\ \{l \cdot u \Rightarrow N \mid r \cdot w \Rightarrow P\} & \longrightarrow & [M/u]N\\ \textbf{case}\ r \cdot M\ \{l \cdot u \Rightarrow N \mid r \cdot w \Rightarrow P\} & \longrightarrow & [M/w]P\end{array}$$

The analogy with the logical reduction again works, for example:

$$\dfrac{\dfrac{M : A}{l \cdot M : A \vee B}\;{\vee}I_1 \qquad \begin{array}{c}\overline{u : A}\;u\\ \vdots\\ N : C\end{array} \qquad \begin{array}{c}\overline{w : B}\;w\\ \vdots\\ P : C\end{array}}{\textbf{case}\ l \cdot M\ \{l \cdot u \Rightarrow N \mid r \cdot w \Rightarrow P\} : C}\;{\vee}E^{u,w} \qquad \longrightarrow \qquad [M/u]N$$

Falsehood. Since there is no constructor for the empty type there is no reduction rule for falsehood. There is no computation rule and we will not try to evaluate **case** M $\{\,\}$.

This concludes the definition of the reduction judgment. Observe that the construction principle for the (computational) reductions is to investigate what happens when a destructor is applied to a corresponding constructor.

This is in correspondence with how (logical) reductions for local soundness consider what happens when an elimination rule is used in succession on the output of an introduction rule (when reading proofs top to bottom).

9 Summary of Proof Terms

Judgments.

$M : A$	M is a proof term for proposition A, see Figure 2
$M \longrightarrow M'$	M reduces to M', see Figure 3

10 Summary of the Curry-Howard Correspondence

The Curry-Howard correspondence we have elaborated in this lecture has three central components:

- Propositions are interpreted as types
- Proofs are interpreted as programs
- Proof reductions are interpreted as computation

This correspondence goes in both directions, but it does not capture everything we have been using so far.

Proposition	**Type**
$A \wedge B$	$\tau \mathbin{\&} \sigma$
$A \supset B$	$\tau \to \sigma$
$A \vee B$	$\tau + \sigma$
$\top$	$\& \{ \}$
$\bot$	0
?	$A \otimes B$
?	1
??	$\rho\alpha.\, \tau$

For $A \otimes B$ and 1 we obtain other forms of logical conjunction and truth that hav the same introduction rules as $A \wedge B$ and $\top$, respectively, but other elimination rules:

$$\dfrac{A \otimes B \qquad \begin{array}{c} \overline{A}^{\,u} \quad \overline{B}^{\,w} \\ \vdots \\ C \end{array}}{C} \otimes E^{u,w} \qquad\qquad \dfrac{1 \quad C}{C}\ 1E$$

Constructors	Destructors
$\dfrac{M : A \quad N : B}{\langle\!\vert M, N \vert\!\rangle : A \wedge B}\ {\wedge}I$	$\dfrac{M : A \wedge B}{M \cdot l : A}\ {\wedge}E_1$ $\dfrac{M : A \wedge B}{M \cdot r : B}\ {\wedge}E_2$
$\dfrac{}{\langle\,\rangle : \top}\ {\top}I$	no destructor for $\top$
$\dfrac{\begin{array}{c} \overline{u : A}\ u \\ \vdots \\ M : B \end{array}}{\lambda u.\, M : A \supset B}\ {\supset}I^u$	$\dfrac{M : A \supset B \quad N : A}{M\, N : B}\ {\supset}E$
$\dfrac{M : A}{l \cdot M : A \vee B}\ {\vee}I_1$ $\dfrac{N : B}{r \cdot N : A \vee B}\ {\vee}I_2$	$\dfrac{M : A \vee B \quad \begin{array}{c} \overline{u : A}\ u \\ \vdots \\ N : C \end{array} \quad \begin{array}{c} \overline{w : B}\ w \\ \vdots \\ P : C \end{array}}{\textbf{case}\ M\ \{l \cdot u \Rightarrow N \mid r \cdot w \Rightarrow P\} : C}\ {\vee}E^{u,w}$
no constructor for $\bot$	$\dfrac{M : \bot}{\textbf{case}\ M\ \{\,\} : C}\ {\bot}E$

Figure 2: Proof term assignment for natural deduction

$$
\begin{array}{rcl}
\langle\!| M, N |\!\rangle \cdot l & \longrightarrow & M \\
\langle\!| M, N |\!\rangle \cdot r & \longrightarrow & N \\
\\
\text{no reduction for } \langle\,\rangle & & \\
\\
(\lambda u.\, M)\, N & \longrightarrow & [N/u]M \\
\\
\textbf{case}\ l \cdot M\ \{l \cdot u \Rightarrow N \mid r \cdot w \Rightarrow P\} & \longrightarrow & [M/u]N \\
\textbf{case}\ r \cdot\ M\ \{l \cdot u \Rightarrow N \mid r \cdot w \Rightarrow P\} & \longrightarrow & [M/w]O \\
\\
\text{no reduction for } \textbf{case}\ M\ \{\,\} & &
\end{array}
$$

Figure 3: Proof term reductions

These are logically equivalent to existing connectives ($A \otimes B \equiv A \wedge B$ and $1 \equiv \top$), so they are not usually used in a treatment of intuitionistic logic, but their operational interpretations are different (eager vs. lazy).

As for general recursive types $\rho\alpha.\, \tau$, there aren't any good propositional analogues on the logical side in general. The overarching study of type theory (encompassing both logic and its computational interpretation) treats the so-called inductive and coinductive types as special cases. Similarly, the fixed point construction **fix** $x.\, e$ does not have a good logical analogue, only special cases of it do.

11 Harmony

> *This is bonus material only touched upon in lecture. It elaborates on how proof reduction arises in the study of logic.*

In the verificationist definition of the logical connectives via their introduction rules we have briefly justified the elimination rules. We now study the balance between introduction and elimination rules more closely.

We elaborate on the verificationist point of view that logical connectives are defined by their introduction rules. We show that for intuitionistic logic as presented so far, the elimination rules are in harmony with the introduction rules in the sense that they are neither too strong nor too weak. We demonstrate this via local reductions and expansions, respectively.

In order to show that introduction and elimination rules are in harmony we establish two properties: *local soundness* and *local completeness*.

Local soundness shows that the elimination rules are not too strong: no matter how we apply elimination rules to the result of an introduction we cannot gain any new information. We demonstrate this by showing that we can find a more direct proof of the conclusion of an elimination than one that first introduces and then eliminates the connective in question. This is witnessed by a *local reduction* of the given introduction and the subsequent elimination.

Local completeness shows that the elimination rules are not too weak: there is always a way to apply elimination rules so that we can reconstitute a proof of the original proposition from the results by applying introduction rules. This is witnessed by a *local expansion* of an arbitrary given derivation into one that introduces the primary connective.

Connectives whose introduction and elimination rules are in harmony in the sense that they are locally sound and complete are properly defined from the verificationist perspective. If not, the proposed connective should be viewed with suspicion. Another criterion we would like to apply uniformly is that both introduction and elimination rules do not refer to other propositional constants or connectives (besides the one we are trying to define), which could create a dangerous dependency of the various connectives on each other. As we present correct definitions we will occasionally also give some counterexamples to illustrate the consequences of violating the principles behind the patterns of valid inference.

In the discussion of each individual connective below we use the notation

$$\begin{array}{c}\mathcal{D}\\ A\ \mathit{true}\end{array} \Longrightarrow_R \begin{array}{c}\mathcal{D}'\\ A\ \mathit{true}\end{array}$$

for the local reduction of a deduction $\mathcal{D}$ to another deduction $\mathcal{D}'$ of the same judgment A *true*. In fact, $\Longrightarrow_R$ can itself be a higher level judgment relating two proofs, $\mathcal{D}$ and $\mathcal{D}'$, although we will not directly exploit this point of view. Similarly,

$$\begin{array}{c}\mathcal{D}\\ A\ \mathit{true}\end{array} \Longrightarrow_E \begin{array}{c}\mathcal{D}'\\ A\ \mathit{true}\end{array}$$

is the notation of the local expansion of $\mathcal{D}$ to $\mathcal{D}'$.

Conjunction. We start with local soundness, i.e., locally reducing an elimination of a conjunction that was just introduced. Since there are two elimination rules and one introduction, we have two cases to consider, because there are two different elimination rules $\wedge E_1$ and $\wedge E_2$ that could follow the

$\wedge I$ introduction rule. In either case, we can easily reduce.

$$\dfrac{\dfrac{\begin{array}{cc}\mathcal{D} & \mathcal{E}\\ A\ \mathit{true} & B\ \mathit{true}\end{array}}{A \wedge B\ \mathit{true}}\wedge I}{A\ \mathit{true}}\wedge E_1 \quad \Longrightarrow_R \quad \begin{array}{c}\mathcal{D}\\ A\ \mathit{true}\end{array}$$

$$\dfrac{\dfrac{\begin{array}{cc}\mathcal{D} & \mathcal{E}\\ A\ \mathit{true} & B\ \mathit{true}\end{array}}{A \wedge B\ \mathit{true}}\wedge I}{B\ \mathit{true}}\wedge E_2 \quad \Longrightarrow_R \quad \begin{array}{c}\mathcal{E}\\ B\ \mathit{true}\end{array}$$

These two reductions justify that, after we just proved a conjunction $A \wedge B$ to be true by the introduction rule $\wedge I$ from a proof $\mathcal{D}$ of A *true* and a proof $\mathcal{E}$ of B *true*, the only thing we can get back out by the elimination rules is something that we have put into the proof of $A \wedge B$ *true*. This makes $\wedge E_1$ and $\wedge E_2$ locally sound, because the only thing we get out is A *true* which already has the direct proof $\mathcal{D}$ as well as B *true* which has the direct proof $\mathcal{E}$. The above two reductions make $\wedge E_1$ and $\wedge E_2$ locally sound.

Local completeness establishes that we are not losing information from the elimination rules. Local completeness requires us to apply eliminations to an arbitrary proof of $A \wedge B$ *true* in such a way that we can reconstitute a proof of $A \wedge B$ from the results.

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

This local expansion shows that, collectively, the elimination rules $\wedge E_1$ and $\wedge E_2$ extract all information from the judgment $A \wedge B$ *true* that is needed to reprove $A \wedge B$ *true* with the introduction rule $\wedge I$. Remember that the hypothesis $A \wedge B$ *true*, once available, can be used multiple times, which is very apparent in the local expansion, because the proof $\mathcal{D}$ of $A \wedge B$ *true* can simply be repeated on the left and on the right premise.

As an example where local completeness fails, consider the case where we "forget" the second/right elimination rule $\wedge E_2$ for conjunction. The remaining rule is still locally sound, because it proves something that was put into the proof of $A \wedge B$ *true*, but not locally complete because we cannot extract a proof of B from the assumption $A \wedge B$. Now, for example, we cannot prove $(A \wedge B) \supset (B \wedge A)$ even though this should clearly be true.

Substitution Principle. We need the defining property for hypothetical judgments before we can discuss implication. Intuitively, we can always substitute a deduction of A *true* for any use of a hypothesis A *true*. In order to avoid ambiguity, we make sure assumptions are labelled and we substitute for all uses of an assumption with a given label. Note that we can only substitute for assumptions that are not discharged in the subproof we are considering. The substitution principle then reads as follows:

> If
>
> $$\begin{array}{c}\dfrac{}{A\ \mathit{true}}\,u\\ \mathcal{E}\\ B\ \mathit{true}\end{array}$$
>
> is a hypothetical proof of B *true* under the undischarged hypothesis A *true* labelled u, and
>
> $$\begin{array}{c}\mathcal{D}\\ A\ \mathit{true}\end{array}$$
>
> is a proof of A *true* then
>
> $$\begin{array}{c}\dfrac{\mathcal{D}}{A\ \mathit{true}}\,u\\ \mathcal{E}\\ B\ \mathit{true}\end{array}$$
>
> is our notation for substituting $\mathcal{D}$ for all uses of the hypothesis labelled u in $\mathcal{E}$. This deduction, also sometime written as $[\mathcal{D}/u]\mathcal{E}$ no longer depends on u.

Implication. To witness local soundness, we reduce an implication introduction followed by an elimination using the substitution operation.

$$\dfrac{\dfrac{\begin{array}{c}\dfrac{}{A\ \mathit{true}}\,u\\ \mathcal{E}\\ B\ \mathit{true}\end{array}}{A \supset B\ \mathit{true}}\supset\!I^u \qquad \begin{array}{c}\mathcal{D}\\ A\ \mathit{true}\end{array}}{B\ \mathit{true}}\supset\!E \quad \Longrightarrow_R \quad \begin{array}{c}\dfrac{\mathcal{D}}{A\ \mathit{true}}\,u\\ \mathcal{E}\\ B\ \mathit{true}\end{array}$$

The conditions on the substitution operation is satisfied, because u is introduced at the $\supset\!I^u$ inference and therefore not discharged in $\mathcal{E}$.

Local completeness is witnessed by the following expansion.

$$\begin{array}{c}\mathcal{D}\\ A \supset B\ \mathit{true}\end{array} \quad \Longrightarrow_E \quad \dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ A \supset B\ \mathit{true}\end{array} \quad \dfrac{}{A\ \mathit{true}}\,u}{B\ \mathit{true}}\,{\supset}E}{A \supset B\ \mathit{true}}\,{\supset}I^u$$

Here u must be chosen fresh: it only labels the new hypothesis A *true* which is used only once.

Disjunction. For disjunction we also employ the substitution principle because the two cases we consider in the elimination rule introduce hypotheses. Also, in order to show local soundness we have two possibilities for the introduction rule, in both situations followed by the only elimination rule.

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ A\ \mathit{true}\end{array}}{A \vee B\ \mathit{true}}\,{\vee}I_L \quad \begin{array}{c}\overline{A\ \mathit{true}}\,u\\ \mathcal{E}\\ C\ \mathit{true}\end{array} \quad \begin{array}{c}\overline{B\ \mathit{true}}\,w\\ \mathcal{F}\\ C\ \mathit{true}\end{array}}{C\ \mathit{true}}\,{\vee}E^{u,w} \quad \Longrightarrow_R \quad \begin{array}{c}\dfrac{\mathcal{D}}{A\ \mathit{true}}\,u\\ \mathcal{E}\\ C\ \mathit{true}\end{array}$$

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ B\ \mathit{true}\end{array}}{A \vee B\ \mathit{true}}\,{\vee}I_R \quad \begin{array}{c}\overline{A\ \mathit{true}}\,u\\ \mathcal{E}\\ C\ \mathit{true}\end{array} \quad \begin{array}{c}\overline{B\ \mathit{true}}\,w\\ \mathcal{F}\\ C\ \mathit{true}\end{array}}{C\ \mathit{true}}\,{\vee}E^{u,w} \quad \Longrightarrow_R \quad \begin{array}{c}\dfrac{\mathcal{D}}{B\ \mathit{true}}\,w\\ \mathcal{F}\\ C\ \mathit{true}\end{array}$$

An example of a rule that would not be locally sound is

$$\dfrac{A \vee B\ \mathit{true}}{A\ \mathit{true}}\,{\vee}E_1?$$

and, indeed, we would not be able to reduce

$$\dfrac{\dfrac{B\ \mathit{true}}{A \vee B\ \mathit{true}}\,{\vee}I_R}{A\ \mathit{true}}\,{\vee}E_1?$$

In fact we can now derive a contradiction from no assumption, which means the whole system is incorrect.

$$\dfrac{\dfrac{\dfrac{}{\top\ \mathit{true}}\,{\top}I}{\bot \vee \top\ \mathit{true}}\,{\vee}I_R}{\bot\ \mathit{true}}\,{\vee}E_1?$$

Local completeness of disjunction distinguishes cases on the known $A \lor B$ *true*, using $A \lor B$ *true* as the conclusion.

$$\begin{array}{c} \mathcal{D} \\ A \lor B \; \mathit{true} \end{array} \quad \Longrightarrow_E \quad \dfrac{\begin{array}{c} \mathcal{D} \\ A \lor B \; \mathit{true} \end{array} \quad \dfrac{\dfrac{}{A \; \mathit{true}}\,u}{A \lor B \; \mathit{true}}\,\lor I_L \quad \dfrac{\dfrac{}{B \; \mathit{true}}\,w}{A \lor B \; \mathit{true}}\,\lor I_R}{A \lor B \; \mathit{true}}\,\lor E^{u,w}$$

Visually, this looks somewhat different from the local expansions for conjunction or implication. It looks like the elimination rule is applied last, rather than first. Mostly, this is due to the notation of natural deduction: the above represents the step from using the knowledge of $A \lor B$ *true* and eliminating it to obtain the hypotheses A *true* and B *true* in the two cases.

Truth. The local constant $\top$ has only an introduction rule, but no elimination rule. Consequently, there are no cases to check for local soundness: any introduction followed by any elimination can be reduced, because $\top$ has no elimination rules.

However, local completeness still yields a local expansion: Any proof of $\top$ *true* can be trivially converted to one by $\top I$.

$$\begin{array}{c} \mathcal{D} \\ \top \; \mathit{true} \end{array} \quad \Longrightarrow_E \quad \dfrac{}{\top \; \mathit{true}}\,\top I$$

Falsehood. As for truth, there is no local reduction because local soundness is trivially satisfied since we have no introduction rule.

Local completeness is slightly tricky. Literally, we have to show that there is a way to apply an elimination rule to any proof of $\bot$ *true* so that we can reintroduce a proof of $\bot$ *true* from the result. However, there will be zero cases to consider, so we apply no introductions. Nevertheless, the following is the right local expansion.

$$\begin{array}{c} \mathcal{D} \\ \bot \; \mathit{true} \end{array} \quad \Longrightarrow_E \quad \dfrac{\begin{array}{c} \mathcal{D} \\ \bot \; \mathit{true} \end{array}}{\bot \; \mathit{true}}\,\bot E$$

Reasoning about situation when falsehood is true may seem vacuous, but is common in practice because it corresponds to reaching a contradiction. In intuitionistic reasoning, this occurs when we prove $A \supset \bot$ which is often abbreviated as $\neg A$. In classical reasoning it is even more frequent, due to the rule of proof by contradiction.

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.

[How80] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980. Hitherto unpublished note of 1969, rearranged, corrected, and annotated by Howard.

[ML80] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, pages 153–175. North-Holland, 1980.

[ML96] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996. Notes for three lectures given in Siena, April 1983.

[Pra65] Dag Prawitz. *Natural Deduction*. Almquist & Wiksell, Stockholm, 1965.

Lecture Notes on
Sequent Calculus

15-814: Types and Programming Languages
Frank Pfenning

Lecture 20
November 13, 2018

1 Introduction

So far, we have presented logical inference in the style of *natural deduction*. Propositions corresponded to types, proofs to programs, and proof reduction to computation.

In this lecture we develop an alternative presentation of logical inference using the *sequent calculus*, also due to Gentzen [Gen35]. From a logical perspective, we change the direction of proof construction, without changing what can be proved. From a computational perspective, this opens up new avenues for capturing computational phenomena, namely *message-passing concurrency* (as we will see in the next lecture).

2 Sequent Calculus Constructs Natural Deductions

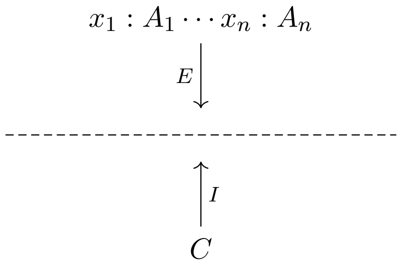
As we have seen in the last lecture, during proof construction we

1. Use *introduction rules* from the bottom up. For example, to prove $A \wedge B$ *true* we reduce it to the subgoals of proving A *true* and B *true*, using $\wedge I$

2. Use *elimination rules* from the top down. For example, if we know $A \wedge B$ *true* we may conclude A *true* using $\wedge E_1$.

The two directions of inference "meet in the middle", when something we have inferred by eliminations matches the conclusion we are trying to prove.

Schematically (and somewhat oversimplified), proving conclusion C from assumptions $x_1 : A_1, \ldots, x_n : A_n$ labeled with variables looks like

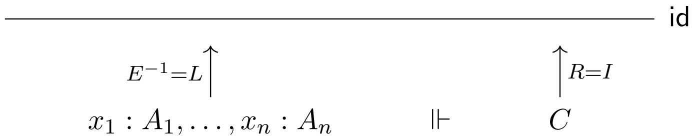


where I indicates introduction rules, E indicates elimination rules, and the dashed line is where proof construction meets in the middle.

This bidirectional reasoning can be awkward, especially if we are trying to establish metatheoretic properties such as consistency of a logical system, that is, that it cannot prove a contradiction $\bot$. Gentzen's idea was to write down the current state of proof construction in a *sequent*

$$x_1 : A_1, \ldots, x_n : A_n \Vdash C$$

and have *right rules* decomposing the *succedent* C while *left rules* decompose the *antecedents* A_i. In this transformation, the *right rules* correspond very directly to the *introduction rules* of natural deduction, because they proceed in the same direction (bottom-up). On the other hand, the left rules correspond to the *inverted elimination rules* because we have to changes their direction from top-down to bottom-up. Schematically:



Rather then meeting in the middle, we now complete the proof construction when we have inferred an antecedent that exactly matches the succedent with the *identity rule*.

$$\frac{}{\Gamma, x : A \Vdash A}\ \mathsf{id}$$

For this and the following rules to make sense, we assume the antecedents are unordered (can be freely exchanged) and all variables x_i are distinct.

Let's use our basic intuition to derive some rules, starting with conjunction.

$$\frac{\Gamma \Vdash A \quad \Gamma \Vdash B}{\Gamma \Vdash A \wedge B} \wedge R \quad \frac{\Gamma, x : A \wedge B, y : A \Vdash C}{\Gamma, x : A \wedge B \Vdash C} \wedge L_1 \quad \frac{\Gamma, x : A \wedge B, z : B \Vdash C}{\Gamma, x : A \wedge B \Vdash C} \wedge L_2$$

The right rule corresponds direction to the introduction rule and the two left rules to the two elimination rules (read upside down) with the twist that the antecedent $x : A \wedge B$ persists in the premise. All of our left rules in this lecture will preserve the antecedent to which we apply the rule so we can use it again, even though it some cases that may seem redundant. As usual, we assume that all antecedent labels x_i are distinct, so that y (in $\wedge L_1$) and z (in $\wedge L_2$) are not aready declared in Γ and different from x.

The right rule for implication is also straightforward.

$$\frac{\Gamma, x : A \Vdash B}{\Gamma \Vdash A \supset B} \supset R$$

How do we use the knowledge of $A \supset B$ in a proof of C? If we can also supply a proof of A we are allowed to assume B in the proof of C.

$$\frac{\Gamma, x : A \supset B \Vdash A \quad \Gamma, x : A \supset B, y : B \Vdash C}{\Gamma, x : A \supset B \Vdash C} \supset L$$

This rule looks a little clunky because we repeat x in both premises. If we leave this implicit

$$\frac{\Gamma \Vdash A \quad \Gamma, y : B \Vdash C}{\Gamma, x : A \supset B \Vdash C} \supset L^*$$

it looks better, but only if we understand that $x : A \supset B$ actually persists in both premises.

In lecture, a student asked the excellent question why we only extract A or B from $A \wedge B$ with the two left rules in the antecedent, but not both together? One answer that we want to faithfully model proof construction in natural deduction, and there happen to be two separate rules to extract the two components. Another answer is: yes, let's do this! What we obtain is actually a *different* logical connective!

$$\frac{\Gamma, x : A \otimes B, y : A, z : B \Vdash C}{\Gamma, x : A \otimes B \Vdash C} \otimes L$$

The corresponding right rule is actually familiar:

$$\frac{\Gamma \Vdash A \quad \Gamma \Vdash B}{\Gamma \Vdash A \otimes B} \otimes R$$

When we reverse-engineer the corresponding natural deduction rules we have

$$\dfrac{A\ \mathit{true} \quad B\ \mathit{true}}{A \otimes B\ \mathit{true}}\ \otimes I \qquad \dfrac{A \otimes B\ \mathit{true} \qquad \begin{array}{c} \overline{A\ \mathit{true}}^{\,y} \quad \overline{B\ \mathit{true}}^{\,z} \\ \vdots \\ C\ \mathit{true} \end{array}}{C\ \mathit{true}}\ \otimes E^{y,z}$$

When looking at this from the lense of proof terms, we realize that $A \wedge B$ corresponds to *lazy pairs* $\tau \mathbin{\&} \sigma$, while $A \otimes B$ corresponds to *eager pairs* $\tau \otimes \sigma$. So even though, purely logically, $A \wedge B \equiv A \otimes B$, they have a different computational meaning. This meaning will diverge even further in the next lecture when we refine the logic and the two connectives are no longer equivalent.

We have left out disjunction, truth, and falsehood, but the rules for them are easy to complete.

However, there is still one rule we need, which is the *converse* of the identity rule. Identity

$$\dfrac{}{\Gamma, x : A \Vdash A}\ \mathsf{id}$$

expresses that if we assume A we can conclude A. The converse would say if we conclude A we can assume A. Expressed as a rule this is called *cut*:

$$\dfrac{\Gamma \Vdash A \quad \Gamma, x : A \Vdash C}{\Gamma \Vdash C}\ \mathsf{cut}$$

Mathematically, this corresponds to introducing the lemma A into a proof of C. We have to prove the lemma (first premise) but then we can use it to prove our succedent (second premise). Generally, in mathematics, finding the right lemma (such as: a generalization of the induction hypothesis) is a critical part of finding proofs. Here, in pure logic with only the usual connective, this rule turns out to be *redundant*. That is, any sequent $\Gamma \Vdash C$ we can derive *with* the rule of cut we can also derive *without* the rule of cut. This is of fundamental *logical* significance because it allows us to establish easily that the system is consistent. All other rules break down either a succedent or an antecedent, and there is no rule to break down falsehood $\bot$, and therefore the cannot be a cut-free proof of $\cdot \Vdash \bot$.

3 Soundness of the Sequent Calculus

By soundness we mean: whenever $\Gamma \Vdash A$ in the sequent calculus then also $\Gamma \vdash A$ in natural deduction. In other words, if we view natural deduction as defining the meaning of the logical connectives, then the sequent calculus let's us draw only correct conclusions. In the next section we prove that the other direction also holds.

Theorem 1 (Soundness of the Sequent Calculus) *If* $\Gamma \Vdash A$ *then* $\Gamma \vdash A$*.*

Proof: The proof is by rule induction over the given sequent calculus derivation. In constructing the natural deduction proof we write all the hypothesis as $x : A$ to the left of the turnstile instead of the assumption $\overline{A}^{\,x}$ in the usual two-dimensional form. We show only two cases.

Case:

$$\frac{\Gamma, x : A \Vdash B}{\Gamma \Vdash A \supset B}\ {\supset}R$$

Then

$\Gamma, x : A \vdash B$ By i.h.
$\Gamma \vdash A \supset B$ By rule ${\supset}I^x$

Case:

$$\frac{\Gamma, x : A \supset B \Vdash A \quad \Gamma, x : A \supset B, y : B \Vdash C}{\Gamma, x : A \supset B \Vdash C}\ {\supset}L$$

Then

$\Gamma, x : A \supset B \vdash A \supset B$ By rule var
$\Gamma, x : A \supset B \vdash A$ By i.h. on first premise
$\Gamma, x : A \supset B \vdash B$ By rule ${\supset}E$
$\Gamma, x : A \supset B, y : B \vdash C$ By i.h. on second premise
$\Gamma, x : A \supset B \vdash C$ By substitution

In the last step we use the substitution property on the two lines just above, substituting the proof of B for the hypothesis $y : B$ in the proof of C.

□

A perhaps more insightful way to present this proof is to annotate the sequent derivation with proof terms drawn from natural deduction. We want to synthesize

$$\Gamma \Vdash M : A$$

such that

$$\Gamma \vdash M : A$$

that is, M is a well-typed (natural deduction) proof term of A. If we can annotate each sequent derivation in this manner, then it will be sound. Fortunately, this is not very difficult. We just have to call upon substitution in the right places. Consider identity and cut.

$$\frac{x : A \in \Gamma}{\Gamma \Vdash x : A}\ \mathsf{id} \qquad \frac{\Gamma \Vdash M : A \quad \Gamma, x : A \Vdash N : C}{\Gamma \Vdash [M/x]N : C}\ \mathsf{cut}$$

Identity just uses a variable, while cut corresponds to substitution. Note that if $M : A$ we can substitute it for the variable $x : A$ appearing in N.

Next consider implication. The right rule (as usual) just mirrors the introduction rule. Intuitively, we obtain M from the induction hypothesis (for an induction we are not spelling out in detail).

$$\frac{\Gamma, x : A \Vdash M : B}{\Gamma \Vdash \lambda x.\, M : A \supset B}\ {\supset}R$$

The left rule is trickier (also as usual!)

$$\frac{\Gamma, x : A \supset B \Vdash M : A \quad \Gamma, x : A \supset B, y : B \Vdash N : C}{\Gamma, x : A \supset B \Vdash\ ?? : C}\ {\supset}L$$

We assume we can annotate the premises, so we have M and N. But how to we construct a proof term for C that does not depend on y? The explicit proof that we have done before tells is it has to be by substitution for $y : B$ and the term will be x (of type $A \supset B$) applied to M (of type A):

$$\frac{\Gamma, x : A \supset B \Vdash M : A \quad \Gamma, x : A \supset B, y : B \Vdash N : C}{\Gamma, x : A \supset B \Vdash [(x\, M)/y]N : C}\ {\supset}L$$

The rules for conjunction are even simpler: in the left rule the additional antecedent y or z is justified by the first and second projection of x.

$$\frac{\Gamma \Vdash M : A \quad \Gamma \Vdash N : B}{\Gamma \Vdash \langle\!| M, N |\!\rangle : A \wedge B}\ {\wedge}R$$

$$\frac{\Gamma, x : A \wedge B, y : A \Vdash N : C}{\Gamma, x : A \wedge B \Vdash [(x \cdot l)/y]N : C}\ {\wedge}L_1 \quad \frac{\Gamma, x : A \wedge B, z : B \Vdash N : C}{\Gamma, x : A \wedge B \Vdash [(x \cdot r)/z]N : C}\ {\wedge}L_2$$

Finally, the other (eager) form of conjunction. No substitution is required here because the case-like elimination construct already matches the sequent calculus rule.

$$\frac{\Gamma \Vdash M : A \quad \Gamma \Vdash N : B}{\Gamma \Vdash \langle M, N \rangle : A \otimes B} \otimes R \qquad \frac{\Gamma, x : A \otimes B, y : A, z : B \Vdash N : C}{\Gamma, x : A \otimes B \Vdash \textbf{case } x \ \{\langle y, z \rangle \Rightarrow N\} : C} \otimes L$$

4 Completeness of the Sequent Calculus

Now we would like to go the other direction: anything we can prove with natural deduction we can also prove in the sequent calculus.

Theorem 2 (Completeness of the Sequent Calculus) *If* $\Gamma \vdash A$ *then* $\Gamma \Vdash A$.

Proof: By rule induction on the deduction of $\Gamma \vdash A$. We show only two representative cases.

Case:

$$\frac{\Gamma, x : A \vdash B}{\Gamma \vdash A \supset B} \supset I$$

Then we construct

$$\frac{\overset{\text{i.h.}}{\Gamma, x : A \Vdash B}}{\Gamma \Vdash A \supset B} \supset R$$

Case:

$$\frac{\Gamma \vdash A \supset B \quad \Gamma \vdash A}{\Gamma \vdash B} \supset E$$

This case requires some thought. From the induction hypothesis we obtain $\Gamma \Vdash A \supset B$ and $\Gamma \Vdash A$ and we need to conclude $\Gamma \Vdash B$. The left rules of the sequent calculus, however, go in the wrong direction, so we cannot easily use the knowledge of $A \supset B$.

In order to create an implication on the left-hand side, we can use the rule of cut, which says that if we know A we can assume A for any proposition A. That is,

$$\frac{\overset{\text{i.h.}}{\Gamma \Vdash A \supset B} \quad \Gamma, x : A \supset B \Vdash ??}{\Gamma \Vdash ??} \textsf{cut}$$

Since we are trying to prove $\Gamma \Vdash B$, using B for $??$ appears to be the obvious choice.

$$\dfrac{\overset{\text{i.h.}}{\Gamma \Vdash A \supset B} \quad \Gamma, x : A \supset B \Vdash B}{\Gamma \Vdash B}\,\textsf{cut}$$

Now we can use the $\supset L$ rules as intended and use the proof of A we have by induction hypothesis.

$$\dfrac{\overset{\text{i.h.}}{\Gamma \Vdash A \supset B} \quad \dfrac{\overset{\text{i.h.}}{\Gamma \Vdash A} \quad \Gamma, y : B \Vdash B}{\Gamma, x : A \supset B \Vdash B}\,{\supset}L}{\Gamma \Vdash B}\,\textsf{cut}$$

The final unproved goal now just follows by the identity.

$$\dfrac{\overset{\text{i.h.}}{\Gamma \Vdash A \supset B} \quad \dfrac{\overset{\text{i.h.}}{\Gamma \Vdash A} \quad \dfrac{}{\Gamma, y : B \Vdash B}\,\textsf{id}}{\Gamma, x : A \supset B \Vdash B}\,{\supset}L}{\Gamma \Vdash B}\,\textsf{cut}$$

Here, we have omitted some unneeded antecedents, particularly $x : A \supset B$ in the premises of $\supset L$. They easily be restored by adding them to the antecedents of every sequent in the deduction. We do not prove this obvious property called *weakening*.

□

Before we investigate what this translation means on proof terms, we revise our language of proof terms for the sequent calculus.

5 Proof Terms for Sequent Calculus

In the soundness proof, we have simply assigned natural deduction proof terms to sequent deductions. This served the purpose perfectly, but such terms do not contain sufficient information to actually reconstruct a sequent proof. For example, in

$$\dfrac{\Gamma \Vdash M : A \quad \Gamma, x : A \Vdash N : C}{\Gamma \Vdash [M/x]N : C}\,\textsf{cut}$$

we would know only the result of substituting M for x in N, which is clearly not enough information to extract M, N, or even A. We restate the rules, this time giving informative proof terms.

$$\frac{x : A \in \Gamma}{\Gamma \Vdash x : A}\ \mathsf{id} \qquad \frac{\Gamma \Vdash M : A \quad \Gamma, x : A \Vdash N : C}{\Gamma \Vdash \textbf{let}\ x : A = M\ \textbf{in}\ N : C}\ \mathsf{cut}$$

$$\frac{\Gamma, x : A \Vdash M : B}{\Gamma \Vdash \lambda x.\, M : A \supset B}\ {\supset}R \qquad \frac{\Gamma, x : A \supset B \Vdash M : A \quad \Gamma, x : A \supset B, y : B \Vdash N : C}{\Gamma, x : A \supset B \Vdash \textbf{let}\ y = x\, M\ \textbf{in}\ N : C}$$

$$\frac{\Gamma \Vdash M : A \quad \Gamma \Vdash N : B}{\Gamma \Vdash \llparenthesis M, N \rrparenthesis : A \wedge B}\ {\wedge}R$$

$$\frac{\Gamma, x : A \wedge B, y : A \Vdash N : C}{\Gamma, x : A \wedge B \Vdash \textbf{let}\ y = x \cdot l\ \textbf{in}\ N : C}\ {\wedge}L_1 \qquad \frac{\Gamma, x : A \wedge B, z : B \Vdash N : C}{\Gamma, x : A \wedge B \Vdash \textbf{let}\ z = x \cdot r\ \textbf{in}\ N : C}$$

$$\frac{\Gamma \Vdash M : A \quad \Gamma \Vdash N : B}{\Gamma \Vdash \langle M, N \rangle : A \otimes B}\ {\otimes}R \qquad \frac{\Gamma, x : A \otimes B, y : A, z : B \Vdash N : C}{\Gamma, x : A \otimes B \Vdash \textbf{case}\ x\ \{\langle y, z \rangle \Rightarrow N\} : C}\ \otimes$$

Just like continuation-passing style, this form of proof term names intermediate values, but it does not make a continuation explicit. We could now rewrite our dynamics on these terms and the rules would be more streamlined since they already anticipate the order in which expressions are evaluated. We can also easily translate from this form to natural deduction terms by replacing all constructs $\textbf{let}\ x = M\ \textbf{in}\ N$ by $[M/x]N$. More formally, we write $M^\dagger$:

$$\begin{array}{lcl}
(x)^\dagger & = & x \\
(\textbf{let}\ x : A = M\ \textbf{in}\ N)^\dagger & = & [M^\dagger/x]N^\dagger \\[1ex]
(\lambda x.\, M)^\dagger & = & \lambda x.\, M^\dagger \\
(\textbf{let}\ y = x\, M\ \textbf{in}\ N)^\dagger & = & [x\, M^\dagger/y]N^\dagger \\[1ex]
\llparenthesis M, N \rrparenthesis^\dagger & = & \llparenthesis M^\dagger, N^\dagger \rrparenthesis \\
(\textbf{let}\ y = x \cdot l\ \textbf{in}\ N)^\dagger & = & [x \cdot l/y]N^\dagger \\
(\textbf{let}\ z = x \cdot r\ \textbf{in}\ N)^\dagger & = & [x \cdot r/z]N^\dagger \\[1ex]
\langle M, N \rangle^\dagger & = & \langle M^\dagger, N^\dagger \rangle \\
(\textbf{case}\ x\ \{\langle y, z \rangle \Rightarrow N\})^\dagger & = & \textbf{case}\ x\ \{\langle y, z \rangle \Rightarrow N^\dagger\}
\end{array}$$

One question is how we translate in the other direction, from natural deduction to these new forms of terms. We write this as M^*. Our proof of the completeness of the sequent calculus holds the key. We read off:

$$\begin{array}{lcl}
(x)^* & = & x \\
(\lambda x.\, M)^* & = & \lambda x.\, M^* \\
(M\, N)^* & = & \mathbf{let}\ x = M^*\ \mathbf{in}\ \mathbf{let}\ y = x\, N^*\ \mathbf{in}\ y
\end{array}$$

Here, we have omitted the type of x (that is, the type of M) in the last line since, computationally, we are not interested in this type. We only tracked it in order to be able to reconstruct the sequent derivation uniquely. Completing this translation is straightforward, keep in mind the proof term language we assigned to the sequent calculus.

$$\begin{array}{lcl}
\langle\!| M, N |\!\rangle^* & = & \langle\!| M^*, N^* |\!\rangle \\
(M \cdot l)^* & = & \mathbf{let}\ x = M^*\ \mathbf{in}\ \mathbf{let}\ y = x \cdot l\ \mathbf{in}\ y \\
(M \cdot r)^* & = & \mathbf{let}\ x = M^*\ \mathbf{in}\ \mathbf{let}\ z = x \cdot r\ \mathbf{in}\ z \\
 & & \\
\langle M, N\rangle^* & = & \langle M^*, N^*\rangle \\
(\mathbf{case}\ M\ \{\langle y, z\rangle \Rightarrow N\})^* & = & \mathbf{let}\ x = M^*\ \mathbf{in}\ \mathbf{case}\ x\ \{\langle y, z\rangle \Rightarrow N^*\}
\end{array}$$

A remarkable property of these translations is that if we translate from natural deduction to sequent calculus and then back we obtain the original term. This does not immediately entail the operational correctness of these translations in the presence of recursion and recursive types, but it does show that the sequent calculus really is a calculus of proof search for natural deduction. If there is a natural deduction proof term M we can find a sequent proof term M' that translates back to M—we have "found" M by construction M'. In general, there will be many different sequent terms M' which all map to the same natural deduction term M, because M' tracks some details on the order which rules were applied that are not visible in natural deduction.

6 Cut Elimination

Gentzen's goal was to prove the consistency of logic as captured in natural deduction. One step in his proof was to show that it is equivalent to the sequent calculus. Now we can ask if the sequent calculus is enough to show that we cannot prove a contradiction. For that purpose we give the rules for $\bot$:

$$\text{no } \bot R \text{ rule} \qquad \frac{}{\Gamma, x : \bot \Vdash C}\ \bot L$$

Ideally, we would like to show that there is there **cannot** be a proof of

$$\cdot \Vdash \bot$$

This, however, is not immediately apparent, because we may just need to find the right "lemma" A and prove

$$\frac{\cdot \Vdash A \quad x : A \Vdash \bot}{\cdot \Vdash \bot}\ \mathsf{cut}$$

Then Gentzen showed a remarkable property: the rule of cut, so essentially in everyday mathematics (Which proof gets by without needing a lemma?) is redundant here in pure logic. That is:

Theorem 3 (Cut Elimination [Gen35]) *If $\Gamma \Vdash A$ then there is a proof of $\Gamma \Vdash A$ without using the rule of cut.*

This immediately implies consistency by inversion: there is no rule with a conclusion matching $\cdot \Vdash \bot$.

The proof of cut elimination is deep and interesting, and there are many resources to understand it.[1] From a computational perspective, however, it is only the so-called *cut reductions* we will discuss in the next lecture that are relevant. This is because in programming languages we impose a particular strategy of evaluation, and, moreover, one that does not evaluate underneath λ-abstractions or inside lazy pairs. In cut elimination, we obey no such restrictions. Plus, in realistic languages we have recursion and recursive types and cut elimination either no longer holds, or holds only for some restricted fragments.

In the next lecture we explore the computational consequences of the sequent calculus from the programming language perspective.

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.

[1] For example, Lecture Notes on Cut Elimination

Lecture Notes on Message-Passing Concurrency

15-814: Types and Programming Languages
Frank Pfenning

Lecture 21
November 15, 2018

1 Introduction

In the last lecture we have seen the sequent calculus as a calculus of proof search for natural deduction. The "informative" proof term assignment decomposed the computation into smaller steps. Today, we will take a leap and provide an interpretation of the sequent calculus based on processes that execute concurrently and pass messages between each other.

2 Destinations

Let's reconsider for a moment the informative proof terms assigned to the sequent calculus, just looking at identity and cut.

$$\frac{}{\Gamma, x : A \Vdash x : A}\ \mathsf{id} \qquad \frac{\Gamma \Vdash M : A \quad \Gamma, x : A \Vdash N : C}{\Gamma \Vdash \mathbf{let}\ x : A = M\ \mathbf{in}\ N : C}\ \mathsf{cut}$$

We can almost give this a store semantics, maybe simplifying the S machine, if we think of every variable standing in for a location at runtime. The only missing piece is that there is no destination for the result of the computation. We can fix that by also naming the right-hand side (statically with a variable, and dynamically with a destination):

$$\frac{}{\Gamma, x : A \Vdash ?? :: (y : A)}\ \mathsf{id} \qquad \frac{\Gamma \Vdash ?? :: (x : A) \quad \Gamma, x : A \Vdash ?? :: (z : C)}{\Gamma \Vdash ?? :: (z : C)}\ \mathsf{cut}$$

The proof term for the identity should copy x to y, which is also the operational interpretation of a destination expression in the S machine.

$$\frac{}{\Gamma, x : A \Vdash (y \leftarrow x) :: (y : A)}\ \mathsf{id}$$

The cut rule creates a new destination for x then runs M to fill it and N to use it.

$$\frac{\Gamma \Vdash M :: (x : A) \quad \Gamma, x : A \Vdash N :: (z : C)}{\Gamma \Vdash \mathbf{let}\ x : A = M\ \mathbf{in}\ N :: (z : C)}\ \mathsf{cut}$$

If M and N run in sequentially, this fits within the model for a functional language we have introduced so far. If M and N run in parallel, then this is the behavior of a future [Hal85]. We can develop the dynamics of the remaining proof terms under this interpretation. The proof terms represent a kind of low-level language for instructions of the S machine.

Instead of pursuing this further, we make a deceptively small change in the sequent calculus to obtain an alternate interpretation as message passing.

3 Linearity

The key reinterpretation of the judgment

$$x_1 : A_1, \ldots, x_n : A_n \Vdash P :: (z : C)$$

is that the x_i and z are *channels for communication* and P is a process. We say P *provides* channel z and *uses* channels x_i. The propositions A_i and C describe a protocol for interaction along the channel x_i and x, respectively.

The first fundamental invariant we want to preserve throughout computation is:

> ***Linearity:*** *Every channel has exactly one provider and exactly one client.*

The second one enables us to identify processes with the channels they provide:

> ***Uniqueness:*** *Every process provides exactly one channel.*

It is possible to relax both of these, but in this lecture we are concerned with the core of the computational interpretation of the (linear) sequent calculus.

Let's reconsider identity and cut in light of these invariants.

$$\frac{}{\Gamma, y : A \Vdash ?? :: (x : A)}\ \mathsf{id}$$

The process $??$ is obligated to *provide* a service following protocol A along x. It also *uses* a channel y of the same type A. One way to fulfill its obligation is to *forward* between x and y and terminate. We can also say that this process *identifies* x and y so that further communication along x will go to the provider of y, and further communication along x will go the client of y. We write this as $x \leftarrow y$ and read it as "*x is implemented by y*".

Since this process terminates by forwarding, it cannot be using any other channels. If it did, those channels would be left without a client, violating linearity! So our final rule is

$$\frac{}{y : A \Vdash (x \leftarrow y) :: (x : A)}\ \mathsf{id}$$

Let's move on to cut, not yet committing to the process expression/proof term for it.

$$\frac{\Gamma \Vdash P :: (x : A) \quad \Gamma, x : A \Vdash Q :: (z : C)}{\Gamma \Vdash ?? :: (z : C)}\ \mathsf{cut}$$

We can observe a few things about this rule. Since channels must be distinct, x is not already declared in Γ. Moreover, P provides a service of type A along x and Q is the client. Also, whatever $??$ turns out to be, it provides along z, the same channel as Q. So $??$ *spawns* a new process P that provides along a fresh channel x and continues with Q. We write this as

$$x \leftarrow P \ ; Q$$

Both P and Q depend on x, P being the provider and Q being the client. Before we can complete the rule, we should consider Γ. In the current form, every channel in Γ suddenly would have two clients, namely P and Q. This violates linearity, so instead we need to "split up" the context: some of the channels should be used by P and others by Q. We use the notation Δ_1, Δ_2 for joining two contexts with no overlapping names. Then we have

$$\frac{\Delta_1 \Vdash P :: (x : A) \quad \Delta_2, x : A \Vdash Q :: (z : C)}{\Delta_1, \Delta_2 \Vdash (x \leftarrow P \ ; Q) :: (z : C)}\ \mathsf{cut}$$

We use Δ as our notation for contexts of channels that should be used *linearly*, that is, with exactly one provider and exactly one client.

In summary, we have

$$\frac{}{y : A \Vdash (x \leftarrow y) :: (x : A)}\ \mathsf{id}$$

$$\frac{\Delta_1 \Vdash P :: (x : A) \quad \Delta_2, x : A \Vdash Q :: (z : C)}{\Delta_1, \Delta_2 \Vdash (x \leftarrow P \ ; Q) :: (z : C)}\ \mathsf{cut}$$

4 Intuitionistic Linear Logic

The sequent calculus we have started derives from *intuitionistic linear logic* [GL87, CCP03]. It is "intuitionistic" because the right-hand side of the sequents are singletons, thereby maintaining our uniqueness invariant. Classical linear logic [Gir87] has symmetric sequents, which has some advantages and some disadvantages for our purposes.

All of the rules we will provide in the remainder of this lecture are indeed also logical rules when one ignores the process expressions. In linear logic, a sequent $\Delta \Vdash A$ expresses that A can be proved from Δ using each antecedent in Δ *exactly once*. Often, this is explained by thinking of the antecedents as abstract *resources* that must be consumed in a proof.

In order to recover the usual expressive power of logic (in our case, intuitionistic logic), linear logic adds a modality $!A$. Only antecedents of this form may be reused or discarded in a proof. We do not develop this modality in this lecture, but might return to it in one of the remaining lectures.

5 Internal Choice

As a first *logical connective* we consider a form of disjunction, written in linear logic as $A \oplus B$. From the computational perspective, a provider of $x : A \oplus B$ should send either l or r. If the provider sends l, communication should the continue following the type A; if it sends r it should continue following B.

$$\frac{\Delta \Vdash P :: (x : A)}{\Delta \Vdash (x.l \ ; P) :: (x : A \oplus B)}\ \oplus R_1 \qquad \frac{\Delta \Vdash P :: (x : B)}{\Delta \Vdash (x.r \ ; P) :: (x : A \oplus B)}\ \oplus R_2$$

The proposition $A \oplus B$ is called *internal choice* because the provider decides whether to choose A (by sending l) or B (by sending r). Conversely, the client must be ready to receive either l or r and then continue communication at type A or B, respectively.

$$\frac{\Delta, x : A \Vdash Q :: (z : C) \quad \Delta, x : B \Vdash R :: (z : C)}{\Delta, x : A \oplus B \Vdash (\mathbf{case}\ x\ \{l \Rightarrow Q \mid r \Rightarrow R\}) :: (z : C)}\ \oplus L$$

At this point you might, and probably *should* object: didn't we say that each antecedent in Δ should be used exactly once in a proof? Or, computational, each channel in Δ should have exactly one client? Here, it looks as if Δ is duplicated to that each channel has two clients: Q and R.

Thinking about the operational semantics clarifies why the rule must be as shown. Imagine the provider of $(x.l\ ;\ P) :: (x : A \oplus B)$ sends l to a client of x, say $\mathbf{case}\ x\ \{l \Rightarrow Q \mid r \Rightarrow R\}$. The provider continues with $P :: (x : A)$ and the client continues with Q. Now each channel used by the original client is used by Q, precisely because we have propagated all of Δ to the first branch. If the provider sends r, then the continuation R is the one that will use all these channels. So linearity is preserved in both cases. If we had split Δ into two, linearity could in fact have been violated because in each case some of the providers could be left without clients.

To formally describe the dynamics we use semantic objects of the form $\mathsf{proc}\ P\ c$ which means that process P executes providing along channel c. Just as in destination-passing style, we do not explicit record the channels that P uses—they simply occur (free) in P. In the S machine we also needed memory cells $!\mathsf{cell}\ d\ v$ and continuations $\mathsf{cont}\ d\ k\ d'$ which turn out not to be required here. In linear logic, every semantic object is in fact a process.

The possible interactions for internal choice then are described by the following two rules:

$$\begin{array}{lllll} (\oplus C_1) & \mathsf{proc}\ (c.l\ ;\ P)\ c, \mathsf{proc}\ (\mathbf{case}\ c\ \{l \Rightarrow Q \mid r \Rightarrow R\})\ d & \mapsto & \mathsf{proc}\ P\ c, \mathsf{proc}\ Q \\ (\oplus C_2) & \mathsf{proc}\ (c.r\ ;\ P)\ c, \mathsf{proc}\ (\mathbf{case}\ c\ \{l \Rightarrow Q \mid r \Rightarrow R\})\ d & \mapsto & \mathsf{proc}\ P\ c, \mathsf{proc}\ R \end{array}$$

Returning to identity and cut, we get the following rules, writing out formally what we described informally.

$$\begin{array}{llll} (\mathsf{id}C) & \mathsf{proc}\ P\ d, \mathsf{proc}\ (c \leftarrow d)\ c & \mapsto & \mathsf{proc}\ ([c/d]P)\ c \\ (\mathsf{cut}C) & \mathsf{proc}\ (x \leftarrow P\ ;\ Q)\ d & \mapsto & \mathsf{proc}\ ([c/x]P)\ c, \mathsf{proc}\ ([c/x]Q)\ d \qquad (c\ \text{fresh}) \end{array}$$

6 An Example: Bit Streams

Already in the fragment with identity, cut, and internal choice, we can write some interesting programs provided we have recursion, both at the level of types and the level of processes. We add this here intuitively, to be formalized later.

Consider a type for a processes sending an infinite stream of bits 0 and 1.

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

For simplicity, we consider this as an equality (so-called *equirecursive types*) rather then an isomorphism (*isorecursive types*), which allows us to avoid sending **fold** or **unfold** messages. We use here the generalized form of internal choice

$$\oplus\{\ell : A_\ell\}_{\ell \in L}$$

for a finite set of labels L. We have $A \oplus B = \oplus\{l : A, r : B\}$, so this is the same idea as behind disjoint sums.

We can write a process (a form of transducer) that receives a bit stream along some channel x it uses and sends a bit stream along the channel y is provides, negating every bit.

$$x : \mathit{bits} \Vdash \mathit{neg} :: (y : \mathit{bits})$$

$$\mathit{neg} = \ldots$$

The first thing *neg* has to do is to receive one bit along x, which corresponds

$$\begin{array}{l} \mathit{neg} = \mathbf{case}\ x\ (\ \mathsf{b0} \Rightarrow \ldots \\ \qquad\qquad\qquad\ \mid \mathsf{b1} \Rightarrow \ldots\ ) \end{array}$$

If we receive b0 we output b1 along y and recurse (to process the remaining stream); if we receive b1 we output b0 and recurse.

$$\begin{array}{l} \mathit{neg} = \mathbf{case}\ x\ (\ \mathsf{b0} \Rightarrow y.\mathsf{b1}\ ;\ \mathit{neg} \\ \qquad\qquad\qquad\ \mid \mathsf{b1} \Rightarrow y.\mathsf{b0}\ ;\ \mathit{neg}\ ) \end{array}$$

What about a process *and* that takes the conjunction of the two bits from corresponding streams? In each phase we have to read the two bits from the two channels, output one bit, and recurse.

$$x : \mathit{bits}, y : \mathit{bits} \Vdash \mathit{and} :: (z : \mathit{bits})$$

$$\begin{array}{l} \mathit{and} = \mathbf{case}\ x\ (\ \mathsf{b0} \Rightarrow \mathbf{case}\ y\ (\ \mathsf{b0} \Rightarrow z.\mathsf{b0}\ ;\ \mathit{and} \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{b1} \Rightarrow z.\mathsf{b0}\ ;\ \mathit{and}\ ) \\ \qquad\qquad\qquad\ \mid \mathsf{b1} \Rightarrow \mathbf{case}\ y\ (\ \mathsf{b0} \Rightarrow z.\mathsf{b0}\ ;\ \mathit{and} \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{b1} \Rightarrow z.\mathsf{b1}\ ;\ \mathit{and}\ )\ ) \end{array}$$

An interesting twist here is that we already know, after receiving b0 that the output will also be b0, so we can output it right away. We just need to be careful to still consume one bit along channel y, or the two input streams fall out of synch.

$$x : \mathit{bits}, y : \mathit{bits} \Vdash \mathit{and} :: (z : \mathit{bits})$$

$$\begin{array}{lllll}
\mathit{and} = \textbf{case}\ x\ (\ \mathsf{b0} \Rightarrow z.\mathsf{b0}\ ;\ \textbf{case}\ y\ (\ \mathsf{b0} \Rightarrow \mathit{and} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{b1} \Rightarrow \mathit{and}\ ) \\
\qquad\qquad\qquad\ \mid \mathsf{b1} \Rightarrow \textbf{case}\ y\ (\ \mathsf{b0} \Rightarrow z.\mathsf{b0}\ ;\ \mathit{and} \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \mid \mathsf{b1} \Rightarrow z.\mathsf{b1}\ ;\ \mathit{and}\ )\ )
\end{array}$$

As a final example along similar lines we consider a process *compress* that compresses consecutives zeros into just one zero. The case where we see a b1 is easy: we just output it and recurse.

$$x : \mathit{bits} \Vdash \mathit{compress} :: (y : \mathit{bits})$$

$$\begin{array}{l}
\mathit{compress} = \textbf{case}\ x\ (\ \mathsf{b0} \Rightarrow \ldots \\
\qquad\qquad\qquad\qquad\ \mid \mathsf{b1} \Rightarrow y.\mathsf{b1}\ ;\ \mathit{compress}\ )
\end{array}$$

When we see a b0 we don't know how many b0's are still to come. So we can output the first b0, but then we need to continue to ignore all following b0's until we see a b1. We need another process definition ignore for this purpose.

$$\begin{array}{l}
x : \mathit{bits} \Vdash \mathit{compress} :: (y : \mathit{bits}) \\
x : \mathit{bits} \Vdash \mathit{ignore} :: (y : \mathit{bits})
\end{array}$$

$$\begin{array}{l}
\mathit{compress} = \textbf{case}\ x\ (\ \mathsf{b0} \Rightarrow \ldots \\
\qquad\qquad\qquad\qquad\ \mid \mathsf{b1} \Rightarrow y.\mathsf{b1}\ ;\ \mathit{compress}\ )
\end{array}$$

$$\begin{array}{l}
\mathit{ignore} = \textbf{case}\ x\ (\ \mathsf{b0} \Rightarrow \mathit{ignore} \\
\qquad\qquad\qquad\ \mid \mathsf{b1} \Rightarrow y.\mathsf{b1}\ ;\ \mathit{compress}\ )
\end{array}$$

At this point it only remains to fill the call to *ignore* after an output of the first b0 seen.

$$\begin{array}{l}
x : \mathit{bits} \Vdash \mathit{compress} :: (y : \mathit{bits}) \\
x : \mathit{bits} \Vdash \mathit{ignore} :: (y : \mathit{bits})
\end{array}$$

$$\begin{array}{l}
\mathit{compress} = \textbf{case}\ x\ (\ \mathsf{b0} \Rightarrow y.\mathsf{b0}\ ;\ \mathit{ignore} \\
\qquad\qquad\qquad\qquad\ \mid \mathsf{b1} \Rightarrow y.\mathsf{b1}\ ;\ \mathit{compress}\ )
\end{array}$$

$$\begin{array}{l}
\mathit{ignore} = \textbf{case}\ x\ (\ \mathsf{b0} \Rightarrow \mathit{ignore} \\
\qquad\qquad\qquad\ \mid \mathsf{b1} \Rightarrow y.\mathsf{b1}\ ;\ \mathit{compress}\ )
\end{array}$$

7 Ending a Session

Viewed as types, the propositions of linear logic are called *session types*, as pioneered by Honda [Hon93]. The logical origins of session types had been in the air (see, for example, Gay and Vasconcelos [GV10]) but wasn't formally spelled out and proved until 2010 [CP10, CPT16]. The concept of a *session* is a sequence of interactions between two processes (for us, provider and client) as specified by a session type.

In the examples so far, all sessions are infinite, which is common and expected in the theory of processes. But we should also have a way to end a session after finitely many interactions. This is the role played by the type 1. As a propositions, it means the "empty" resource (or the absence of resources). Computationally, a provider of $x : 1$ can end a session and terminate, while a client waits for the session to be ended. We can also think of this as *closing* a channel of communication. To preserve our linearity invariant, the process that ends the session cannot use any other channels.

$$\frac{}{\cdot \Vdash \mathbf{close}\, x :: (x : 1)}\ 1R \qquad \frac{\Delta \Vdash Q :: (z : C)}{\Delta, x : 1 \Vdash (\mathbf{wait}\, x \ ; Q) :: (z : C)}\ 1L$$

The reduction:

$$(1C) \quad \mathsf{proc}\ (\mathbf{close}\, c)\ c, \mathsf{proc}\ (\mathbf{wait}\ c\ ; Q)\ d \ \mapsto\ \mathsf{proc}\ Q\ d$$

A few words on our conventions:

- Even though the semantic objects in a configuration are unordered, we always write the provider of a channel to the left of its client.
- We use P for providers, and Q for clients.
- We use x, y, z for expression variables that stand for channels, while c, d, e are used for channels as they exist as processes execute. This is the same distinction we make between variables in a functional program and destinations or memory addresses at runtime.

8 An Example: Binary Numbers

As another example we use numbers in binary form represented as a sequence of messages. This is almost like bit streams, but they can be terminated by ϵ, which represents 0 as the empty string of bits.

$$\mathit{bin} = \oplus\{\mathsf{b0} : \mathit{bin}, \mathsf{b1} : \mathit{bin}, \epsilon : 1\}$$

A process *zero* producing the representation of 0 is easy. After sending the label ϵ we have to end the session by closing the channel because the type of x at this point in the session has become 1.

$$\begin{array}{l}
\cdot \Vdash \mathit{zero} :: (x : \mathit{bin}) \\
\mathit{zero} = x.\epsilon \;;\; \mathbf{close}\, x
\end{array}$$

A process that computes the successor of a binary number is more complicated.

$$x : \mathit{bin} \Vdash \mathit{succ} :: (y : \mathit{bin})$$

$$\begin{array}{ll}
\mathit{succ} = \mathbf{case}\; x & (\,\mathsf{b0} \Rightarrow \ldots \\
& \mid \mathsf{b1} \Rightarrow \ldots \\
& \mid \epsilon \Rightarrow \ldots)
\end{array}$$

Let's start with the last case. The label ϵ represents 0, so we have to send along y the representation of 1, which is $\mathsf{b1}$ followed by ϵ.

$$x : \mathit{bin} \Vdash \mathit{succ} :: (y : \mathit{bin})$$

$$\begin{array}{ll}
\mathit{succ} = \mathbf{case}\; x & (\,\mathsf{b0} \Rightarrow \ldots \\
& \mid \mathsf{b1} \Rightarrow \ldots \\
& \mid \epsilon \Rightarrow y.\mathsf{b1} \;;\; y.\epsilon \;;\; \ldots)
\end{array}$$

At this point we have $x : 1$ in the context and the successor process must provide $y : 1$. We could accomplish this by forwarding $y \leftarrow x$ or by waiting for x to close and then close y. Let's use the latter version.

$$x : \mathit{bin} \Vdash \mathit{succ} :: (y : \mathit{bin})$$

$$\begin{array}{ll}
\mathit{succ} = \mathbf{case}\; x & (\,\mathsf{b0} \Rightarrow \ldots \\
& \mid \mathsf{b1} \Rightarrow \ldots \\
& \mid \epsilon \Rightarrow y.\mathsf{b1} \;;\; y.\epsilon \;; \\
& \qquad\quad \mathbf{wait}\; x \;;\; \mathbf{close}\; y\,)
\end{array}$$

In the case of $\mathsf{b0}$, *succ* just outputs $\mathsf{b1}$. Then the remaining string of bits is unchanged, so we just forward.

$$x : \mathit{bin} \Vdash \mathit{succ} :: (y : \mathit{bin})$$

$$\begin{array}{ll}
\mathit{succ} = \mathbf{case}\; x & (\,\mathsf{b0} \Rightarrow y.\mathsf{b1} \;;\; y \leftarrow x \\
& \mid \mathsf{b1} \Rightarrow \ldots \\
& \mid \epsilon \Rightarrow y.\mathsf{b1} \;;\; y.\epsilon \;; \\
& \qquad\quad \mathbf{wait}\; x \;;\; \mathbf{close}\; y\,)
\end{array}$$

When the first (lowest) bit of the input is b1 we have to output b0, but we still need to take care of the carry bit. We can do this simply by calling *succ* recursively.

$$x : \mathit{bin} \Vdash \mathit{succ} :: (y : \mathit{bin})$$

$$\begin{array}{lll}
\mathit{succ} = \mathbf{case}\ x\ (& \mathsf{b0} \Rightarrow y.\mathsf{b1}\ ;\ y \leftarrow x \\
 & \mid \mathsf{b1} \Rightarrow y.\mathsf{b0}\ ;\ \mathit{succ} \\
 & \mid \epsilon \Rightarrow y.\mathsf{b1}\ ;\ y.\epsilon\ ; \\
 & \qquad \mathbf{wait}\ x\ ;\ \mathbf{close}\ y\ )
\end{array}$$

9 External Choice

In internal choice $A \oplus B$ it is the provider who gets to choose. External choice $A \mathbin{\&} B$ let's the client choose, which means the provider has to be ready with two different branches.

$$\frac{\Delta \Vdash P_1 :: (x : A) \quad \Delta \Vdash P_2 :: (x : B)}{\Delta \Vdash (\mathbf{case}\ x\ \{l \Rightarrow P_1 \mid r \Rightarrow P_2\}) :: (x : A \mathbin{\&} B)}\ \&R$$

$$\frac{\Delta, x : A \Vdash Q :: (z : C)}{\Delta, x : A \mathbin{\&} B \Vdash x.l\ ;\ Q :: (z : C)}\ \&L_1 \qquad \frac{\Delta, x : B \Vdash Q :: (z : C)}{\Delta, x : A \mathbin{\&} B \Vdash x.r\ ;\ Q :: (z : C)}\ \&L_2$$

We see that internal choice and external choice are quite symmetric in the linear sequent calculus, while in natural deduction (and functional programming) they look much further apart. The transition rules follow the pattern of internal choice, with the role of provider and client swapped.

$$\begin{array}{lllll}
(\&C_1) & \mathsf{proc}\ (\mathbf{case}\ c\ \{l \Rightarrow P_1 \mid r \Rightarrow P_2\})\ c, \mathsf{proc}\ (c.l\ ;\ Q)\ d & \mapsto & \mathsf{proc}\ P_1\ c, \mathsf{proc} \\
(\&C_2) & \mathsf{proc}\ (\mathbf{case}\ c\ \{l \Rightarrow P_1 \mid r \Rightarrow P_2\})\ c, \mathsf{proc}\ (c.r\ ;\ Q)\ d & \mapsto & \mathsf{proc}\ P_2\ c, \mathsf{proc}
\end{array}$$

With external choice we can implement a *counter* that can take two labels: increment (inc) that increments its internal value and val after which it streams the bits making up the current value of the counter. In the latter case, the counter is also destroyed, so with this interface we can request its value only once.

$$\mathit{ctr} = \&\{\mathsf{inc} : \mathit{ctr}, \mathsf{val} : \mathit{bin}\}$$

We implement the counter as a process that holds a binary number as an internal data structure, which is implemented as a process of type *bin*.

$$y : \mathit{bin} \Vdash \mathit{counter} :: (x : \mathit{ctr})$$

We say y represents an internal data structure because *counter* is the *only* client of it (by linearity), so no other process can access it.

The counter distinguishes cases based on the label received along x. After all, it is an external choice so we need to know what the client requests.

$$\begin{array}{l}
y : \mathit{bin} \Vdash \mathit{counter} :: (x : \mathit{ctr}) \\
\mathit{counter} = \mathbf{case}\ x\ (\,\mathsf{inc} \Rightarrow \ldots \\
\qquad\qquad\qquad\quad \mid \mathsf{val} \Rightarrow \ldots\,)
\end{array}$$

We increment such a counter by using the *succ* process from the previous example. We can do this by spawning a new successor process without actually receiving anything from the stream y.

$$\begin{array}{l}
y : \mathit{bin} \Vdash \mathit{counter} :: (x : \mathit{ctr}) \\
\mathit{counter} = \mathbf{case}\ x\ (\,\mathsf{inc} \Rightarrow y' \leftarrow \mathit{succ} \leftarrow y\ ; \\
\qquad\qquad\qquad\qquad\qquad\quad \ldots \\
\qquad\qquad\qquad\quad \mid \mathsf{val} \Rightarrow \ldots\,)
\end{array}$$

In order to spawn a new process and not become confused with different variables called y, we use the notation

$$x \leftarrow f \leftarrow y_1, \ldots, y_n\ ;\ P$$

for a cut, passing channels $y_1, \ldots, y_n$ to process f that provides along the fresh channel x that can be used in P. Note that due to linearity, $y_1, \ldots, y_n$ will no longer be available since now the freshly spawned instance of f is their client.

We use this same notation for the recursive call to *counter*.

$$\begin{array}{l}
y : \mathit{bin} \Vdash \mathit{counter} :: (x : \mathit{ctr}) \\
\mathit{counter} = \mathbf{case}\ x\ (\,\mathsf{inc} \Rightarrow y' \leftarrow \mathit{succ} \leftarrow y\ ; \\
\qquad\qquad\qquad\qquad\qquad\quad x \leftarrow \mathit{counter} \leftarrow y' \\
\qquad\qquad\qquad\quad \mid \mathsf{val} \Rightarrow \ldots\,)
\end{array}$$

Just so we don't make a mess of the bound variables, we define *counter* as depending on two channels, x and y. When *counter* is called, x will be created fresh since a new process is spanned, and y will be passed to this this new process.

$$
\begin{array}{l}
y : \mathit{bin} \Vdash \mathit{counter} :: (x : \mathit{ctr}) \\[1ex]
x \leftarrow \mathit{counter} \leftarrow y = \\
\quad \textbf{case}\ x\ (\,\mathsf{inc} \Rightarrow y' \leftarrow \mathit{succ} \leftarrow y\ ; \\
\qquad\qquad\qquad\quad x \leftarrow \mathit{counter} \leftarrow y' \\
\qquad\quad \mid \mathsf{val} \Rightarrow \ldots\,)
\end{array}
$$

Now we can fill in the last case, where we just forward to the privately held binary number, thereby terminating and communicating the number back to the client.

$$
\begin{array}{l}
y : \mathit{bin} \Vdash \mathit{counter} :: (x : \mathit{ctr}) \\[1ex]
\mathit{counter} = \textbf{case}\ x\ (\,\mathsf{inc} \Rightarrow y' \leftarrow \mathit{succ} \leftarrow y\ ; \\
\qquad\qquad\qquad\qquad\quad x \leftarrow \mathit{counter} \leftarrow y' \\
\qquad\qquad\qquad \mid \mathsf{val} \Rightarrow x \leftarrow y\,)
\end{array}
$$

To assure you that this is type-correct, we see that the type of counter, after seeing the val becomes *bin*, which is exactly the type of y.

We can create a new counter with some initial value by calling this process and passing it a process holding the initial value. For example,

$$
\begin{array}{l}
z \leftarrow \mathit{zero}\ ; \\
c \leftarrow \mathit{counter} \leftarrow z\ ; \\
P
\end{array}
$$

creates a new counter c that can be used in P. The channel z, on the other hand, is not accessible there because it has been passed to the instance of *counter*.

More formally, if we see a type declaration and a definition

$$
\begin{array}{l}
y_1 : B_1, \ldots, y_n : B_n \Vdash f :: (x : A) \\
x \leftarrow f \leftarrow y_1, \ldots, y_n = P
\end{array}
$$

then we check

$$y_1 : B_1, \ldots, y_n : B_n \Vdash P :: (x : A)$$

and every call

$$x' \leftarrow f \leftarrow y'_1, \ldots, y'_n$$

is executed as

$$x' \leftarrow [x'/x, y'_1/y_1, \ldots, y'_n/y_n]P$$

A tail call (which has no continuation)

$$x \leftarrow f \leftarrow y'_1, \ldots, y'_n$$

is syntactically expanded into a call, followed by a forward

$$\begin{array}{l} x' \leftarrow f \leftarrow y'_1, \ldots y'_n \ ; \\ x \leftarrow x' \end{array}$$

We can now rewrite the earlier definitions in this style, for consistency. We only show this for the processes on binary numbers.

$$\mathit{bin} = \oplus\{\mathsf{b0} : \mathit{bin}, \mathsf{b1} : \mathit{bin}, \epsilon : 1\}$$

$$\cdot \Vdash \mathit{zero} :: (x : \mathit{bin})$$

$$x \leftarrow \mathit{zero} = x.\epsilon \ ; \ \mathbf{close}\, x$$

$$x : \mathit{bin} \Vdash \mathit{succ} :: (y : \mathit{bin})$$

```
$y \leftarrow \mathit{succ} \leftarrow x =$
   $\mathbf{case}\ x\ (\,\mathsf{b0} \Rightarrow y.\mathsf{b1}\ ;\ y \leftarrow x$
              $\mid \mathsf{b1} \Rightarrow y.\mathsf{b0}\ ;\ \mathit{succ}$
              $\mid \epsilon \Rightarrow y.\mathsf{b1}\ ;\ y.\epsilon\ ;$
                     $\mathbf{wait}\ x\ ;\ \mathbf{close}\ y\,)$
```

Taking stock, we see that *external choice* provides an object-oriented style of concurrent programming where we send messages to objects and may (or may not) receive replies. In contrast, *internal choice* looks more like functional programming, done concurrently: instead of representing data in memory, they are represented via messages. However, nonterminating process such as transducers make perfect sense because we care about the interactive behavior of processes and not just a final value.

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.

[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. Special Issue on Behavioural Types.

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

[GL87] Jean-Yves Girard and Yves Lafont. Linear logic and lazy computation. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, volume 2, pages 52–66, Pisa, Italy, March 1987. Springer-Verlag LNCS 250.

[GV10] Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, January 2010.

[Hal85] Robert H. Halstead. Multilisp: A language for parallel symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–539, October 1985.

[Hon93] Kohei Honda. Types for dyadic interaction. In *4th International Conference on Concurrency Theory*, CONCUR'93, pages 509–523. Springer LNCS 715, 1993.

Lecture Notes on Session Types

15-814: Types and Programming Languages
Frank Pfenning

Lecture 22
November 27, 2018

1 Introduction

Some of the material in this lecture, specifically, the discussion of external choice and the implementation of a counter, are already provided in the notes for Lecture 21 on *Message-Passing Concurrency*. First, we have identity (forwarding) and cut (spawn), which work parametrically over all types:

$c \leftarrow d$	implement c by d and terminate
$x \leftarrow P \; ; \; Q$	spawn $[c/x]P$, providing a fresh c, with client $[c/x]Q$

Here is a summary table of the message-passing constructs in our process language so far, organized by type.

Type	**Provider**	**Client**	**Continuation Ty**
$c : \oplus\{\ell : A_\ell\}_{\ell \in L}$	$(c.k \; ; \; P)$	$\mathbf{case}\; c \; \{\ell \Rightarrow Q_\ell\}_{\ell \in L}$	$c : A_k$
$c : \&\{\ell : A_\ell\}_{\ell \in L}$	$\mathbf{case}\; c \; \{\ell \Rightarrow P_\ell\}_{\ell \in L}$	$(c.k \; ; \; Q)$	$c : A_k$
$c : 1$	$\mathbf{close}\; c$	$\mathbf{wait}\; c \; ; \; Q$	*(none)*

Note that there is a complete symmetry here between internal and external choice, only the role of provider and client are swapped. Compare that to the functional world, where disjoint sums $\tau + \sigma$ and lazy pairs $\tau \;\&\; \sigma$ exhibit a number of differences. Partly, the additional simplicity gained is due to the sequent calculus as compared to natural deduction. In particular, in the sequent calculus all connectives have uniform right- and left-rules, while in natural deduction the elimination rules for positive connectives $(1, \tau \otimes \sigma, \tau + \sigma)$ all use **case** constructs and are therefore much different from those for negative connectives $(\tau \rightarrow \sigma, \tau \;\&\; \sigma)$. The other reason for the simplicity here is linearity.

2 Passing Channels

Even though the examples in this course do not use them, we can also ask what the message-passing counterparts to $\tau \to \sigma$ and $\tau \otimes \sigma$ are. The first one is easy to guess: $A \multimap B$ corresponds to receiving a channel d of type A and continuing with type B. Conversely, $A \otimes B$ corresponds to *sending* a channel d of type A. From this we can straightforwardly reconstruct the typing rules, but we refer the interested reader, for example, to Balzer et al. [BP17].

Type	Provider	Client	Continuation Ty
$c : \oplus\{\ell : A_\ell\}_{\ell \in L}$	$(c.k\ ;\ P)$	$\textbf{case}\ c\ \{\ell \Rightarrow Q_\ell\}_{\ell \in L}$	$c : A_k$
$c : \&\{\ell : A_\ell\}_{\ell \in L}$	$\textbf{case}\ c\ \{\ell \Rightarrow P_\ell\}_{\ell \in L}$	$(c.k\ ;\ Q)$	$c : A_k$
$c : 1$	$\textbf{close}\ c$	$\textbf{wait}\ c\ ;\ Q$	*(none)*
$c : A \multimap B$	$x \leftarrow \textbf{recv}\ c\ ;\ P$	$\textbf{send}\ c\ d$	B
$c : A \otimes B$	$\textbf{send}\ c\ d$	$x \leftarrow \textbf{recv}\ c\ ;\ Q$	B

Again, we see a symmetry between $A \multimap B$ and $A \otimes B$, while in a functional language, functions $\tau \to \sigma$ and eager pairs $\tau \otimes \sigma$ are quite different.

3 Session Types in Concurrent C0

In the remainder of this lecture we demonstrate the robustness and practicality of these somewhat abstract ideas about message-passing concurrent computation by presenting a concrete instantiation of the ideas in Concurrent C0 [WPP16, BP17].

Instead of the notations of linear logic, Concurrent C0 uses more traditional notation of *session types* [Hon93]. Concurrent C0 is based on C0 [1], a type-safe and memory-safe subset of C0 augmented with contracts. C0 is used in the freshman-level introductory computer science class at CMU. Many of the syntax decision are motivated by consistency with C and should be viewed in this light.

First, session types are enclosed in angle brackets `< ... >` to make lexing and parsing them conservative over C0 (and C). Any *sending* interaction from the provider perspective is written as `!_` while a receiving interaction is written as `?_`.

A *choice*, whether it is *internal* ($\oplus$) or *external* ($\&$), must be declared with a name. This declaration is modeled after a `struct` declaration in C. So

$$\{\ell_1 : A_1, \ldots, \ell_n : A_n\}$$

[1] http://c0.typesafety.net

is written as

```
choice cname {
  < A1 > l1;
  ...
  < An > ln;
};
```

where `cname` is the name of this particular choice.

For example, to represent binary numbers

$$bin = \oplus\{\mathsf{b0} : bin, \mathsf{b1} : bin, \epsilon : 1\}$$

we would start by declaring the choice

```
choice bin {
  < ... > b0;
  < ... > b1;
  < ... > eps;
};
```

How do we fill in the continuation session types inside the angle brackets? The first two are straightforward: They are of type *bin*, which is the *internal choice* over `bin`.

```
choice bin {
  <!choice bin> b0;
  <!choice bin> b1;
  < ... >       eps;
};
```

In the case of epsilon (label `eps`) we close the channel without a continuation, which is written as the empty session type.

```
choice bin {
  <!choice bin> b0;
  <!choice bin> b1;
  < >           eps;
};
```

For good measure, we *define* the type `bin` to stand for the internal choice `!choice bin`:

```
typedef <!choice bin> bin;
```

4 Channels and Process Definitions

In Concurrent C0, names of channels are prefixed by '`$`' so they can be easily distinguished from ordinary variables. A process definition then has the general form

```
<A> $c pname (t1 x1, ..., tn xn) {
  ... process expression ...
}
```

where `A` is the session type of the channel c provided by the process name `pname`. Each of the arguments `xi` can be either a channel or an ordinary variable.

We start by defining the process that send the representation of the number 0.

```
bin $z zero () {
  $z.eps ; close($z);
}
```

We see that sending a label `l` along channel `$c` is written as `$c.l` and closing a channel `$c` is simply `close($c)`.

Next, we implement the successor process that receives a stream of binary digits representing n along a channel x it uses, and sends a stream of digits representing n along a channel y is provides. Recall from the last lecture:

$$x : \mathit{bin} \Vdash \mathit{succ} :: (y : \mathit{bin})$$

$$\begin{aligned}\mathit{succ} = \mathbf{case}\ x\ (\ & \mathsf{b0} \Rightarrow y.\mathsf{b1}\ ;\ y \leftarrow x \\ \mid\ & \mathsf{b1} \Rightarrow y.\mathsf{b0}\ ;\ \mathit{succ} \\ \mid\ & \epsilon \Rightarrow y.\mathsf{b1}\ ;\ y.\epsilon\ ; \\ & \qquad \mathbf{wait}\ x\ ;\ \mathbf{close}\ y\ )\end{aligned}$$

Following the style of C, the **case** construct is written as a `switch` statement whose subject is a channel `$c`. We select the appropriate branch according to the label received along `$c`.

```
bin $y succ(bin $x) {
  switch ($x) {
    case b0: {
      $y.b1; $y = $x;
    }
    case b1: {
      $y.b0; $y = succ($x);
    }
    case eps: {
      $y.b1; wait($x); $y.eps; close($y);
    }
  }
}
```

Forwarding $y \leftarrow x$ is written as an assignment `$y = $x`.

5 Functions Using Channels

In Concurrent C0, functions that return values may also use channels. For example, here is a function to print a number in binary form, with the *most significant bit* first (the way we are used to seeing numbers).

```
void print_bin_rec(bin $x) {
  switch ($x) {
    case b0: {
      print_bin_rec($x);
      print("0");
      return;
    }
    case b1: {
      print_bin_rec($x);
      print("1");
      return;
    }
    case eps: {
      wait($x);
      return;
    }
  }
}

void print_bin(bin $x) {
  print_bin_rec($x);
  print(".\n");
  return;
}
```

Now we can implement a simple main function for testing purposes.

6 Functions Using Channels

In Concurrent C0, functions that return values may also use channels. For example, here is a function to print a number in binary form, with the *most significant bit* first (the way we are used to representing numbers).

```
void print_bin_rec(bin $x) {
  switch ($x) {
    case b0: {
      print_bin_rec($x);
      print("0");
      return;
    }
    case b1: {
      print_bin_rec($x);
      print("1");
      return;
    }
    case eps: {
      wait($x);
      return;
    }
  }
}

void print_bin(bin $x) {
  print_bin_rec($x);
  print(".\n");
  return;
}
```

The following simple main function should print `100.` and finish. Note that the call to `print_bin` is *sequential*, while the calls to `zero` and `succ` spawn new processes. We also see how each channel is created and then used, so that at the end of the functions all channels have been used.

```
int main() {
  bin $z = zero();
  bin $one = succ($z);
  bin $two = succ($one);
  bin $three = succ($two);
  bin $four = succ($three);
  print_bin($four);
  return 0;
}
```

7 Implementing a Counter Process

Recall that a counter has the interface

$\mathit{ctr} = \&\{\mathsf{inc} : \mathit{ctr}, \mathsf{val} : \mathit{bin}\}$

that is, it receives either a inc or val label. There are no new ideas required to represent this type. We just use external choice `?_` instead of internal choice `!_` where appropriate.

```
choice ctr {
  <?choice ctr> inc;
  <!choice bin> val;
};

typedef <?choice ctr> ctr;
```

Recall from the last lecture

$$x : \mathit{bin} \Vdash \mathit{counter} :: (c : \mathit{ctr})$$

$$\begin{array}{lll} \mathit{counter} = \textbf{case}\ c & (\,\mathsf{inc} \Rightarrow & y \leftarrow \mathit{succ} \leftarrow x\ ; \\ & & c \leftarrow \mathit{counter} \leftarrow y \\ & \mid \mathsf{val} \Rightarrow & c \leftarrow x\,) \end{array}$$

This is easy to transliterate:

```
ctr $c counter(bin $x) {
  switch ($c) {
    case inc: {
      bin $y = succ($x);
      $c = counter($y);
    }
    case val: {
      $c = $x;
    }
  }
}
```

We now write a more complicated `main` function, using two loops. For each loop, we have to make sure that the type of any channel is *loop invariant*, since we do not know how many times we go around the loop.

```
int main() {
  bin $z = zero();
  bin $one = succ($z);
  bin $two = succ($one);
  bin $three = succ($two);
  bin $four = succ($three);
  for (int i = 0; i < 1000; i++) {
    $four = succ($four);
  }
  ctr $c = counter($four);      /* counter, initialized with 10
  for (int i = 0; i < 2000; i++) {
    $c.inc;
  }
  $c.val;
  print_bin($c);                /* 3004 */
  return 0;
}
```

8 Lists and Stacks

As a final example, we program lists of binary numbers and stacks, where a stack is like an object holding a list. This example demostrates the passing of channels.

$$\mathit{list} = \oplus\{\mathsf{nil} : 1, \mathsf{cons} : \mathit{bin} \otimes \mathit{list}\}$$

$$\mathit{stack} = \&\,\{\mathsf{push} : \mathit{bin} \multimap \mathit{stack}, \mathsf{pop} : \mathit{response}\}$$
$$\mathit{response} = \oplus\{\mathsf{none} : 1, \mathsf{some} : \mathit{bin} \otimes \mathit{stack}\}$$

We say "list", but it is not represented in memory but a protocol by which individual elements are sent across a channel. Note that type *stack* and *response* are mutually recursive. In Concurrent C0:

```
choice list {
  < >                   nil;
  <!bin ; !choice list> cons;
};

typedef <!choice list> list;

choice stack {
  <?bin ; ?choice stack> push;
  <!choice response>     pop;
};
choice response {
  < >                    none;
  <!bin ; ?choice stack> some;
};
typedef <?choice stack> stack;
```

Then we have processes `Nil` and `Cons`, somewhat similar to `zero` and `succ`.

```
list $n Nil() {
  $n.nil; close($n);
}

list $k Cons(bin $x, list $l) {
  $k.cons; send($k,$x); $k = $l;
}
```

Finally, the process implementing a stack. It is the sole client of the list `$l`, which acts as a "local" storage.

```
stack $s stack_proc(list $l) {
  switch ($s) {
    case push: {
      bin $x = recv($s);
      list $k = Cons($x,$l);
      $s = stack_proc($k);
    }
    case pop: {
      switch ($l) {
        case nil: {
          wait($l);
          $s.none; close($s);
        }
        case cons: {
          bin $x = recv($l);
          $s.some; send($s, $x);
          $s = stack_proc($l);
        }
      }
    }
  }
}
```

In the updated `main` function we just push one element onto the stack, pop it off, and print it. We should now actually call pop again and wait for the stack process to terminate, but we ran out of time during lecture so we just raise an error. With this particular code we cannot reach the end of the `main` function, so we have to comment out the return since Concurrent C0 detects and flags unreachable code.

```
int main() {
  bin $z = zero();
  bin $one = succ($z);
  bin $two = succ($one);
  bin $three = succ($two);
  bin $four = succ($three);
  for (int i = 0; i < 1000; i++) {
    $four = succ($four);
  }
  ctr $c = counter($four);
  for (int i = 0; i < 2000; i++) {
    $c.inc;
  }
  $c.val;
  // print_bin($c);                /* 3004 */
  list $n = Nil();
  stack $s = stack_proc($n);
  $s.push; send($s, $c);
  $s.pop;
  switch ($s) {
    case none: {
      error("impossible");
    }
    case some: {
      bin $d = recv($s);
      print_bin($d);
      error("out of time");
    }
  }
  // return 0;
}
```

References

[BP17] Stephanie Balzer and Frank Pfenning. Manifest sharing with session types. In *International Conference on Functional Programming (ICFP)*, pages 37:1–37:29. ACM, September 2017.

[Hon93] Kohei Honda. Types for dyadic interaction. In *4th International Conference on Concurrency Theory*, CONCUR'93, pages 509–523. Springer LNCS 715, 1993.

[WPP16] Max Willsey, Rokhini Prabhu, and Frank Pfenning. Design and implementation of Concurrent C0. In *Fourth International Workshop on Linearity*, pages 73–82. EPTCS 238, June 2016.