Collected Lecture Notes

15-814: Types and Programming Languages
Frank Pfenning

Lecture 1–26
Fall 2020

Lecture Notes on The Lambda Calculus

15-814: Types and Programming Languages
Frank Pfenning

Lecture 1
Tuesday, September 1, 2020

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 $\lambda x.\, x + 5 = \lambda y.\, y + 5 = \lambda whatever.\, whatever + 5$. 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 Simple Functions and Combinators

The simplest functions are the identity function and the constant function. The identity function, called *I*, just returns its argument x.

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

The constant function returning x could be written as

$$\lambda y.\, x$$

We calculate

$$(\lambda y.\, x)\, e \longrightarrow_\beta x$$

for any expression e since y does not occur in the expression x. This is somewhat incomplete in the sense the expression $\lambda y.\, x$ has a *free variable* which is therefore fixed. What we would like is a *closed expression K* (one without free variables) such $K\, x$ is the constant function, always returning x. But that's easy: we just abstract over x!

$$K = \lambda x.\, \lambda y.\, x$$

Then $K\, x \longrightarrow_\beta \lambda y.\, x$ is the constant function returning x.

A combinator for us is just a closed λ-expression like *I* or *K*. We will see more interesting combinators in the next lecture.

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

We generally apply α-conversion silently, identifying terms that differ only in the names of the bound variables.

Reduction.

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

5 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

$$\begin{array}{rcl} \mathit{not}\ \mathit{true} & =_\beta & \mathit{false} \\ \mathit{not}\ \mathit{false} & =_\beta & \mathit{true} \end{array}$$

We start with the obvious:

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

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 both paths. Let's first try the one where b is applied to some arguments.

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

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

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

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

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

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.

$$\begin{array}{lcl} \mathit{not}\ \mathit{true} & \longrightarrow_\beta^3 & \mathit{false} \\ \mathit{not}\ \mathit{false} & \longrightarrow_\beta^3 & \mathit{true} \end{array}$$

We write $\longrightarrow_\beta^n$ 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 $\longrightarrow_\beta$.

An alternative solution hinted at above is to start with

$$\mathit{not}' = \lambda b.\, \lambda x.\, \lambda y.\ \ldots$$

We pose this because the result of $\mathit{not}\, b$ should be a Boolean, and the two Booleans both start with two λ-abstractions. Now we reuse the previous idea, but apply b not to *false* and *true*, but to y and x.

$$\mathit{not}' = \lambda b.\, \lambda x.\, \lambda y.\, b\, y\, x$$

Again, we calculate

$$\begin{array}{lcl} \mathit{not}'\ \mathit{true} & \longrightarrow_\beta^3 & \mathit{false} \\ \mathit{not}'\ \mathit{false} & \longrightarrow_\beta^3 & \mathit{true} \end{array}$$

An important observation here is that

$$\mathit{not} = \lambda b.\, b\ (\lambda x.\, \lambda y.\, y)\ (\lambda x.\, \lambda y.\, x) \neq \lambda b.\, \lambda x.\, \lambda y.\, b\, y\, x = \mathit{not}'$$

Both of these are *normal forms* (they cannot be reduced) and therefore represent *values* (the results of computation). Both correctly implement negation on Booleans, but they are *different*. This is evidence that when computing with particular data representations in the λ-calculus it is *not extensional*: even though the functions behave the same on all the arguments we care about (here just *true* and *false*), the are not convertible. To actually see that they are not convertible we need the Church-Rosser theorem which says if e_1 and e_2 are $\alpha\beta$-convertible then there is a common reduct e such that $e_1 \longrightarrow_\beta^* e$ and $e_2 \longrightarrow_\beta^* e$.

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

$$\begin{array}{lcl} \mathit{and}\ \mathit{true}\ \mathit{true} & =_\beta & \mathit{true} \\ \mathit{and}\ \mathit{true}\ \mathit{false} & =_\beta & \mathit{false} \\ \mathit{and}\ \mathit{false}\ \mathit{true} & =_\beta & \mathit{false} \\ \mathit{and}\ \mathit{false}\ \mathit{false} & =_\beta & \mathit{false} \end{array}$$

Learning from the negation, we start by guessing

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

where we arbitrarily put b first. Looking at the equations, we see that if b is *true* then the result is always c.

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

If b is *false* the result is always just *false*, no matter what c is.

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

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.

6 The LAMBDA Language

In lecture, we used a toy implementation of a the λ-calculus in a language called LAMBDA. This implementation uses a *concrete syntax* where λ is

written as a backslash '\'. A program consists of a sequence of declarations, of which there are three forms:

defn $x = e$	variable x stands for e
norm $x = e$	variable x stands for the normal form of e
conv $e_1 = e_2$	verify that e_1 and e_2 have the same normal form

Allowing definitions is a convenience, but it does not change the expressive power of the λ-calculus, because we can replace **defn** $x = e$ by $(\lambda x.\ \ldots)\, e$ where '$\ldots$' represents the scope of the definition. The **norm** and **conv** declarations initiate computation and allow the programmer to examine the normal form of an expression (if it exists).

In addition, declarations can be negated with !, for example, to check that two expressions are not convertible.

```
1  % represent booleans as closed expressions in normal form
2
3  defn true = \x. \y. x
4  defn false = \x. \y. y
5
6  defn not = \b. b false true
7  defn not' = \b. \x. \y. b y x
8
9  (* confirm that not and not' are not convertible *)
10 !conv not = not'
11
12 % normalize "not true"
13 norm _ = not true
14
15 % test not and not' against their specification
16 conv not true = false
17 conv not false = true
18
19 conv not' true = false
20 conv not' false = true
```

Listing 1: Booleans in LAMBDA

For more information on LAMBDA, consult the Software page for the course.

Exercises

Exercise 1 Define the following functions on Booleans in at least two distinct ways.

1. Exclusive or "*xor*".
2. The conditional "*if*" such that

$$\begin{array}{lcl} \textit{if true } e_1\ e_2 & =_\beta & e_1 \\ \textit{if false } e_1\ e_2 & =_\beta & e_2 \end{array}$$

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 Primitive Recursion

15-814: Types and Programming Languages
Frank Pfenning

Lecture 2
Thursday, September 3, 2020

1 Introduction

In this lecture we continue our exploration of the λ-calculus and the representation of data and functions on them. We give schematic forms to define functions on natural numbers and give uniform ways to represent them in the λ-calculus. We begin with the *schema of iteration* and then proceed the more complex *schema of primitive recursion*. In the next lecture we will arrive at the fully general scheme of *recursion*.

2 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

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

We call it B because that's its traditional name as a combinator.

The unit of composition should the identity function, as defined by $I = \lambda x.\, x$. Composing any other function f with I should just yield f. In other words, we expect

$$B\ f\ I \stackrel{?}{=} f \stackrel{?}{=} B\, I\, f$$

Let's calculate:

$$\begin{array}{rcl} B\ f\ I & = & (\lambda f.\, \lambda g.\, \lambda x.\, f\,(g\,x))\ f\ I \\ & \longrightarrow_\beta & (\lambda g.\, \lambda x.\, f\,(g\,x))\ I \\ & \longrightarrow_\beta & \lambda x.\, f\,(I\,x) \\ & \longrightarrow_\beta & \lambda x.\, f\,x \\ & \stackrel{?}{=} & f \end{array}$$

We see the result is not exactly f as we expected, but $\lambda x.\, f\,x$. However, these two expressions always behave the same when applied to an arbitrary argument so they are *extensionally equal*. To capture this we add one more rule to the λ-calculus:

$$\eta\text{-conversion} \quad \lambda x.\, e\,x \quad =_\eta \quad e \quad \text{provided } x \notin \mathsf{FV}(e)$$

The proviso that x not be among the *free variables* of e is needed, because $\lambda x.\, x\,x \neq \lambda x.\, y\,x$. The first applies the argument to itself, the second applies y to the given argument.

It is possible to orient this equation and investigate the notion of $\beta\eta$-reduction. However, it turns out this is somewhat artificial because extensionality is a reasoning principle for equality and not a priori a computational principle. Interestingly, in the setting of typed λ-calculi it makes more sense to use the equation from right to left, called η-expansion, but some discipline has to be imposed or expansion does not terminate.

We should remember that this form of extensionality does not extend to functions defined over specific representations. For example, we saw there are (at least) two formulations of negation on Booleans which are not equal, even if we throw in the rule of η-conversion.

3 Nontermination

At this point we pause briefly to ask three natural questions:

1. Does every expression have a normal form?
2. Can we always compute a normal form if one exists?

3. Are normal forms unique?

The answers to these questions are crucial to understanding to what extent we might consider the λ-calculus a universal model of computation.

Does every expression have a normal form?

If the λ-calculus is to be equivalent in computational power to Turing machines in some way, then we would expect the answer to be "no" because computations of Turing machines may not halt. However, it is not immediate to think of some expression that doesn't have a normal form. If you haven't seen something like this already, you may want to play around with some expressions to see if you can come up with one.

The simplest one is

$$\Omega = (\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$$

Indeed, there is only one possible β-reduction and it immediately leads to exactly the same term:

$$\begin{array}{lcl} \Omega & = & (\lambda x.\, x\, x)\, (\lambda x.\, x\, x) \\ & \longrightarrow_\beta & (\lambda x.\, x\, x)\, (\lambda x.\, x\, x) \\ & \longrightarrow_\beta & (\lambda x.\, x\, x)\, (\lambda x.\, x\, x) \\ & \longrightarrow_\beta & \ldots \end{array}$$

So Ω reduces in one step to itself and only to itself.

Can we always compute a normal form if one exists?

The answer here is "yes", although it is not easy to prove that this is the case. Let's consider an example (recall that $K = \lambda x.\, \lambda y.\, x$):

$$K\, I\, \Omega \longrightarrow_\beta (\lambda y.\, I)\, \Omega \longrightarrow_\beta I$$

So the expression $K\, I\, \Omega$ does have a normal form, even though Ω does not. This is because the constant function $K\, I$ ignores its argument. On the other hand we also have

$$K\, I\, \Omega \longrightarrow_\beta K\, I\, \Omega \longrightarrow_\beta K\, I\, \Omega \longrightarrow_\beta \cdots$$

because we have the $\Omega \longrightarrow_\beta \Omega$ and reduction can be applied anywhere in an expression.

Fortunately, there is a strategy which turns out to be complete in the sense that if an expression has a normal form, this strategy will find it. It is called *leftmost-outermost* or *normal-order reduction*. This strategy scans through the expression from left to right and when it find a *redex* (that is, an expression of the form $(\lambda x.\, e)\, e'$) it applies β-reduction and then returns to the beginning of the result expression. In particular, it does not consider any redex in e or e', only the "outermost" one. Also, in an expression $((\lambda x.\, e_1)\, e_2)\, e_3$ it does not consider any potential redex in e_3, only the leftmost one.

This strategy works in our example: the redex in Ω would not be considered, only the redex $K\, I$ and then the redex $(\lambda y.\, I)\, \Omega$.

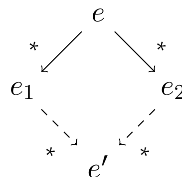
The implementation of LAMBDA uses a straightforward function for leftmost-outermost reduction, complicated very slightly by the fact that

names such as K or I which in the notes are only abbreviations at the mathematical level of discourse, are actual language-level definitions in the implementation. So we have to expand the definition of K, for example, before applying β-reduction, but we do not officially count this as a substitution.

The notion of leftmost-outermost reduction is closely related to the notion of call-by-name evaluation in programming languages (and, with a little more distance, to call-by-need which is employed in Haskell). In contrast, call-by-value would reduce the argument of a function before applying the β-reduction, which is not complete, as our example shows. The analogy is not exact, however, since in programming languages such as ML or Haskell we also do not reduce under λ-abstractions, a fact that represents a sharp dividing line between foundational calculi such as the λ-calculus and actual programming languages. We will justify and understand these decisions in a few lectures.

Are normal forms unique?

The outcome of a computation starting from e is its normal form. At any point during a computation there may be many redices. Ideally, the outcome would be independent of the reduction strategy we choose as long as we reach a normal form. Otherwise, the meaning of an expression (as represented by its normal form) may be ambiguous. Therefore, Church and Rosser [**?**] spend considerable effort in proving the uniqueness of normal forms. The key technical device is a property called *confluence* (also referred to as the *Church-Rosser property*). It is often depicted in the following diagram:



In words: if we can reduce e to e_1 and also e to e_2 then there exists an e' such that e_1 and e_2 both reduce to e'. The solid lines are given reduction sequences while the reduction sequences represented by dashed lines have to be shown to exist. Reduction here is in multiple steps (indicated by the star "$*$"). For the λ-calculus (and the original Church-Rosser Theorem), this reduction would usually be β-reduction. Very roughly, the proof shows how

to simulate the steps from e to e_2 when starting from e_1 and (symmetrically) simulate the steps from e to e_1 when starting from e_2.

Confluence implies the uniqueness of normal forms. Suppose e_1 and e_2 in the diagram are normal forms. Because they cannot be reduced further, the sequence of reductions to e' must consist of zero steps, so $e_1 = e' = e_2$.

Confluence implies that even though we might embark on an unfortunate path (for example, keep reducing Ω in $K\,I\,\Omega$) we can still recover if indeed there is a normal form. In this example, we might eventually decide to reduce $K\,I$ and then the redex $(\lambda y.\,I)\,\Omega$.

4 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 as 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.

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

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

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

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):

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

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)

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

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.

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

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

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

5 The Schema of Iteration

As we saw in the first lecture, a natural number n is represented by a function $\overline{n}$ that iterates its first argument n times applied to the second: $\overline{n}\, g\, c = \underbrace{g\, (\ldots (g}_{n \text{ times}}\ c))$. Another way to specify such a function schematically is

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

If a function satisfies such a *schema of iteration* then it can be defined in the λ-calculus on Church numerals as

$$f = \lambda n.\, n\, g\, c$$

which is easy to verify. The class of function definable this way is *total* (that is, defined on all natural numbers if c and g are), which can easily be proved by induction on n. Returning to examples from the last lecture, let's consider multiplication again.

$$\begin{array}{lcl} \mathit{times}\ 0\ k & = & 0 \\ \mathit{times}\ (n+1)\ k & = & k + \mathit{times}\ n\ k \end{array}$$

This doesn't exactly fit our schema because k is an additional parameter. That's usually allowed for iteration, but to avoid generalizing our schema the *times* function can just return a *function* by abstracting over k.

$$\begin{array}{lcl} \mathit{times}\; 0 & = & \lambda k.\, 0 \\ \mathit{times}\; (n+1) & = & \lambda k.\, k + \mathit{times}\, n\, k \end{array}$$

We can read off the constant c and the function g from this schema

$$\begin{array}{lcl} c & = & \lambda k.\, \mathit{zero} \\ g & = & \lambda r.\, \lambda k.\, \mathit{plus}\, k\, (r\, k) \end{array}$$

and we obtain

$$\mathit{times} = \lambda n.\, n\, (\lambda r.\, \lambda k.\, \mathit{plus}\, k\, (r\, k))\, (\lambda k.\, \mathit{zero})$$

which is more complicated than the solution we constructed by hand

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

The difference in the latter solution is that it takes advantage of the fact that k (the second argument to *times*) never changes during the iteration. We have repeated here the definition of *plus*, for which there is a similar choice between two versions as for *times*.

6 The Schema of Primitive Recursion

It is easy to define very fast-growing functions by iteration, such as the exponential function, or the "stack" function iterating the exponential.

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

Everything appears to be going swimmingly until we think of a very simple function, namely the predecessor function defined by

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

You may try for a while to see if you can define the predecessor function, but it is difficult. The problem is that we have to go from $\lambda s.\, \lambda z.\, s\, (\ldots (s\, z))$

to $\lambda s.\, \lambda z.\, s\,(\ldots z)$, that is, we have to *remove* an s rather than add an s as was required for the successor. One possible way out is to change representation and define $\overline{n}$ differently so that predecessor becomes easy (see Exercise **??**). We run the risk that other functions then become more difficult to define, or that the representation is larger than the already inefficient unary representation already is. We follow a different path, keeping the representation the same and defining the function directly.

We can start by assessing why the schema of iteration does not immediately apply. The problem is that in

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

the function g only has access to the result of the recursive call of f on n, but not to the number n itself. What we would need is the *schema of primitive recursion*:

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

where n is passed to h. For example, for the predecessor function we have $c = 0$ and $h = \lambda x.\, \lambda y.\, x$ (we do not need the result of the recursive call, just n which is the first argument to h).

6.1 Defining the Predecessor Function

Instead of trying to solve the general problem of how to implement primitive recursion, let's define the predecessor directly. Mathematically, we write $n \dot{-} 1$ for the predecessor (that is, $0 \dot{-} 1 = 0$ and $n + 1 \dot{-} 1 = n$). The key idea is to gain access to n in the schema of primitive recursion by *rebuilding it* during the iteration. This requires *pairs*, a representation of which we will construct shortly.

Our specification then is

$$\mathit{pred}_2\, n = \langle n, n \dot{-} 1 \rangle$$

and the key step in its implementation in the λ-calculus is to express the definition by a schema of *iteration* rather than *primitive recursion*. The start is easy:

$$\mathit{pred}_2\, 0 = \langle 0, 0 \rangle$$

For $n + 1$ we need to use the value of $\mathit{pred}_2\, n$. For this purpose we assume we have a function *letpair* where

$$\mathit{letpair}\, \langle e_1, e_2 \rangle\, k = k\, e_1\, e_2$$

In other words, *letpair* passes the elements of the pair to a "continuation" k. Using *letpair* we start as

$$\mathit{pred}_2\,(n+1) = \mathit{letpair}\,(\mathit{pred}_2\,n)\,(\lambda x.\,\lambda y.\,\ldots)$$

If pred_2 satisfies it specification then reduction will substitute n for x and $n \dot{-} 1$ for y. From these we need to construct the pair $\langle n+1, n\rangle$ which we can do, for example, with $\langle x+1, x\rangle$. This gives us

$$\begin{array}{lcl}
\mathit{pred}_2\,0 & = & \langle 0, 0\rangle \\
\mathit{pred}_2\,(n+1) & = & \mathit{letpair}\,(\mathit{pred}_2\,n)\,(\lambda x.\,\lambda y.\,\langle x+1, x\rangle) \\
\\
\mathit{pred}\,n & = & \mathit{letpair}\,(\mathit{pred}_2\,n)\,(\lambda x.\,\lambda y.\,y)
\end{array}$$

6.2 Defining Pairs

The next question is how to define pairs and *letpair*. The idea is to simply abstract over the continuation itself! Then *letpair* isn't really needed because the functional representation of the pair itself will apply its argument to the two components of the pair, but if want to write it out it would be the identity.

$$\begin{array}{lcl}
\overline{\langle x, y\rangle} & = & \lambda k.\,k\,x\,y \\
\mathit{pair} & = & \lambda x.\,\lambda y.\,\lambda k.\,k\,x\,y \\
\mathit{letpair} & = & \lambda p.\,p
\end{array}$$

6.3 Proving the Correctness of the Predecessor Function

Summarizing the above and expanding the definition of *letpair* we obtain

$$\begin{array}{l}
\mathit{pred}_2 = \lambda n.\,n\,(\lambda p.\,p\,(\lambda x.\,\lambda y.\,\mathit{pair}\,(\mathit{succ}\,x)\,x))\,(\mathit{pair}\,\mathit{zero}\,\mathit{zero}) \\
\mathit{pred} = \lambda n.\,\mathit{pred}_2\,n\,(\lambda x.\,\lambda y.\,y)
\end{array}$$

Let's do a rigorous proof of correctness of *pred*, especially since we got it wrong when we worked in a hurry during lecture. For the representation of natural numbers, it is convenient to assume its correctness in the form

$$\begin{array}{lcl}
\overline{0}\,g\,c & =_\beta & c \\
\overline{n+1}\,g\,c & =_\beta & g\,(\overline{n}\,g\,c)
\end{array}$$

Lemma 1 $\mathit{pred}_2\,\overline{n} =_\beta \overline{\langle n, n \dot{-} 1\rangle}$

Proof: By mathematical induction on n.

Base: $n = 0$. Then

$$\begin{array}{lll} \mathit{pred}_2\,\overline{0} & =_\beta \overline{0}\,(\ldots)\,(\mathit{pair}\,\mathit{zero}\,\mathit{zero}) & \\ & =_\beta \mathit{pair}\,\mathit{zero}\,\mathit{zero} & \text{By repn. of } 0 \\ & =_\beta \overline{\langle 0, 0\rangle} = \overline{\langle 0, 0 \dot{-} 1\rangle} & \text{By repn. of } 0 \text{ and pairs} \end{array}$$

Step: $n = m + 1$. Then

$$\begin{array}{ll} \mathit{pred}_2\,\overline{m+1} =_\beta \overline{m+1}\,(\lambda p.\, p\,(\lambda x.\, \lambda y.\, \mathit{pair}\,(\mathit{succ}\,x)\,x))\,(\mathit{pair}\,\mathit{zero}\,\mathit{zero}) & \\ =_\beta (\lambda p.\, p\,(\lambda x.\, \lambda y.\, \mathit{pair}\,(\mathit{succ}\,x)\,x))\,(\overline{m}\,(\lambda p.\, \ldots)\,(\ldots)) & \text{By repn. of } m+1 \\ =_\beta (\lambda p.\, p\,(\lambda x.\, \lambda y.\, \mathit{pair}\,(\mathit{succ}\,x)\,x))\,(\mathit{pred}_2\,\overline{m}) & \text{By defn. of } \mathit{pred}_2 \\ =_\beta (\lambda p.\, p\,(\lambda x.\, \lambda y.\, \mathit{pair}\,(\mathit{succ}\,x)\,x))\,\overline{\langle m, m \dot{-} 1\rangle} & \text{By ind. hyp. on } m \\ =_\beta \overline{\langle m, m \dot{-} 1\rangle}\,(\lambda x.\, \lambda y.\, \mathit{pair}\,(\mathit{succ}\,x)\,x) & \\ =_\beta \mathit{pair}\,(\mathit{succ}\,\overline{m})\,\overline{m} & \text{By repn. of pairs} \\ =_\beta \overline{\langle m+1, m\rangle} & \text{By repn. of successor and pairs} \\ = \overline{\langle m+1, (m+1) \dot{-} 1\rangle} & \text{By defn. of } \dot{-} \end{array}$$

□

Theorem 2 $\mathit{pred}\,\overline{n} =_\beta \overline{n \dot{-} 1}$

Proof: Direct, from Lemma **??**.

$$\begin{array}{lll} \mathit{pred}\,\overline{n} & = (\lambda n.\, \mathit{pred}_2\,n\,(\lambda x.\, \lambda y.\, y))\,\overline{n} & \\ & =_\beta \mathit{pred}_2\overline{n}\,(\lambda x.\, \lambda y.\, y) & \\ & =_\beta \overline{\langle n, n \dot{-} 1\rangle}\,(\lambda x.\, \lambda y.\, y) & \text{By Lemma ??} \\ & =_\beta (\lambda k.\, k\,\overline{n}, \overline{n \dot{-} 1})\,(\lambda x.\, \lambda y.\, y) & \text{By repn. of pairs} \\ & =_\beta \overline{n \dot{-} 1} & \end{array}$$

□

An interesting consequence of the Church-Rosser Theorem is that if $e =_\beta e'$ where e' is in normal form, then $e \longrightarrow_\beta^* e'$.

6.4 General Primitive Recursion

The general case of primitive recursion follows by a similar argument. Recall

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

We begin by defining a function f_2 specified with

$$f_2\, n = \langle n, f\, n\rangle$$

We can define f_2 using the schema of iteration.

$$\begin{array}{lcl} f_2\, 0 & = & \langle 0, c\rangle \\ f_2\,(n+1) & = & \mathit{letpair}\,(f_2\, n)\,(\lambda x.\, \lambda y.\, \langle x+1, h\, x\, y\rangle) \\ f\, n & = & \mathit{letpair}\,(f_2\, n)\,(\lambda x.\, \lambda y.\, x) \end{array}$$

To put this all together, we implement a function specified with

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

with the following definition in terms of c and h:

$$\begin{array}{lcl} \mathit{pair} & = & \lambda x.\, \lambda y.\, \lambda g.\, g\, x\, y \\ f_2 & = & \lambda n.\, n\,(\lambda r.\, r\ (\lambda x.\, \lambda y.\, \mathit{pair}\ (\mathit{succ}\ x)\ (h\ x\ y)))\,(\mathit{pair}\ \mathit{zero}\ c) \\ f & = & \lambda n.\, f_2\, n\ (\lambda x.\, \lambda y.\, y) \end{array}$$

Recall that for the concrete case of the predecessor function we have $c = 0$ and $h = \lambda x.\, \lambda y.\, x$.

7 The Significance of Primitive Recursion

We have used primitive recursion here only as an aid to see how we can define functions in the pure λ-calculus. However, when computing over natural numbers we can restrict the functions that can be formed in schematic ways to obtain a language in which all functions terminate. Primitive recursion plays a central role in this because if c and g are terminating then so is f formed from them by primitive recursion. This is easy to see by induction on n.

In this ways we obtain a very rich set of functions but we couldn't use them to fully simulate Turing machines, for example.

Furthermore, if we give a so-called *constructive* proof of a statement in certain formulations of arithmetic with mathematical induction, we can extract a function that is defined by primitive recursion. We will probably not have an opportunity to discuss this observation further in this course, but it is an important topic in the course 15-317/15-657 *Constructive Logic*.

8 A Few Somewhat More Rigorous Definitions

We write out some definitions for notions from the first two lectures a little more rigorously.

λ-Expressions. First, the abstract syntax.

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

$\lambda x.\, e$ *binds* x with scope e. In the concrete syntax, the scope of a binder λx is as large as possible while remaining consistent with the given parentheses so $y\,(\lambda x.\, x\, x)$ stands for $y\,(\lambda x.\,(x\, x))$. Juxtaposition $e_1\, e_2$ is left-associative so $e_1\, e_2\, e_3$ stands for $(e_1\, e_2)\, e_3$.

We define $\mathsf{FV}(e)$, the *free variables* of e with

$$\begin{array}{lll} \mathsf{FV}(x) & = & \{x\} \\ \mathsf{FV}(\lambda x.\, e) & = & \mathsf{FV}(e) \backslash \{x\} \\ \mathsf{FV}(e_1\, e_2) & = & \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2) \end{array}$$

Renaming. Proper treatment of names in the λ-calculus is notoriously difficult to get right, and even more difficult when one *reasons about* the λ-calculus. A key convention is that "*variable names do not matter*", that is, we actually *identify expressions that differ only in the names of their bound variables*. So, for example, $\lambda x.\, \lambda y.\, x\, z = \lambda y.\, \lambda x.\, y\, z = \lambda u.\, \lambda w.\, u\, z$. The textbook defines *fresh renamings* [**?**, pp. 8–9] as bijections between sequences of variables and then α-conversion based on fresh renamings. Let's take this notion for granted right now and write $e =_\alpha e'$ if e and e' differ only in the choice of names for their bound variables and this observation is important. From now on we identify e and e' if they differ only in the names of their bound variables, which means that other operations such as substitution and β-conversion are defined on α-equivalence classes of expressions.

Substitution. We can now define *substitution of e' for x in e*, written $[e'/x]e$, following the structure of e.

$$\begin{array}{llll} [e'/x]x & = & e' & \\ [e'/x]y & = & y & \text{for } y \neq x \\ [e'/x](\lambda y.\, e) & = & \lambda y.[e'/x]e & \text{provided } y \notin \mathsf{FV}(e') \\ [e'/x](e_1\, e_2) & = & ([e'/x]e_1)\,([e'/x]e_2) & \end{array}$$

This looks like a partial operation, but since we identify terms up to α-conversion we can always rename the bound variable y in $[e'/x](\lambda y.\, e)$ to another variable that is not free in e' or e. Therefore, substitution is a *total function* on α-equivalence classes of expressions.

Now that we have substitution, we also characterize α-conversion as $\lambda x.\, e =_\alpha \lambda y.\, [y/x]e$ provided $y \notin \mathsf{FV}(e)$ but as a definition it would be circular because we already required renaming to define substitution.

Equality. We can now define β- and η-conversion. We understand these conversion rules as defining a *congruence*, that is, we can apply an equation anywhere in an expression that matches the left-hand side of the equality. Moreover, we extend them to be reflexive, symmetric, and transitive so we can write $e =_\beta e'$ if we can go between e and e' by multiple steps of β-conversion.

$$\begin{array}{llcll} \beta\text{-conversion} & (\lambda x.\, e)\, e' & =_\beta & [e'/x]e & \\ \eta\text{-conversion} & \lambda x.\, e\, x & =_\eta & e & \text{provided } x \notin \mathsf{FV}(e) \end{array}$$

Reduction. Computation is based on reduction, which applies β-conversion in the left-to-right direction. In the pure calculus we also treat it as a congruence, that is, it can be applied anywhere in an expression.

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

Sometimes we like to keep track of length of reduction sequences so we write $e \longrightarrow_\beta^n e'$ if we can go from e to e' with n steps of β-reduction, and $e \longrightarrow_\beta^* e'$ for an arbitrary n (including 0).

Confluence. The Church-Rosser property (also called confluence) guarantees that the normal form of a λ-expression is unique, if it exists.

Theorem 3 (Church-Rosser [?]) *If $e \longrightarrow_\beta^* e_1$ and $e \longrightarrow_\beta^* e_2$ then there exists an e' such that $e_1 \longrightarrow_\beta^* e'$ and $e_2 \longrightarrow_\beta^* e'$.*

Exercises

Exercise 1 Analyze whether $B\ I\ f \stackrel{?}{=} f$ and, if so, whether it requires only β-conversion or $\beta\eta$-conversion.

Exercise 2 *Once we can define each individual instance of the schemas of iteration and primitive recursion, we can also define them explicitly as combinators.*

Define combinators iter and primrec such that

(i) The function iter $g\ c$ satisfies the schema of iteration

(ii) The function primrec $h\ c$ satisfies the schema of primitive recursion

You do not need to prove the correctness of your definitions.

Exercise 3 One approach to representing functions defined by the schema of primitive recursion is to change the representation so that $\overline{n}$ is not an iterator but a *primitive recursor*.

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

1. Define the successor function *succ* (if possible) and show its correctness.
2. Define the predecessor function *pred* (if possible) and show its correctness.
3. Explore if it is possible to directly represent any function f specified by a schema of primitive recursion, ideally without constructing and destructing pairs.

References

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

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

Lecture Notes on Recursion

15-814: Types and Programming Languages
Frank Pfenning

Lecture 3
Tuesday, September 8, 2020

1 Introduction

In this lecture we first complete our development of recursion: from iteration through primitive recursion to full recursion. Then we will introduce *simple types* to sort out our data representations.

2 General Recursion

Recall the schemas of iteration and primitive recursion:

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

We have already seen how functions defined by iteration and primitive recursion can be represented in the λ-calculus. We can also see that functions defined in this manner are terminating as long as c and g are.

But there are many functions that do not fit such of schema, for two reasons: (1) their natural presentation differs from the rigid schema (even if there actually is one that fits it), and (2) they simply fall out of the class of functions. An example of (1) is below; an example of (2) would be a function simulating a Turning machine. Since setting up a representation of Turing machines is tedious, we just show simple examples of (1).

Let's consider the subtraction-based specification of a *gcd* function for

the greatest common divisor of strictly positive natural numbers $a, b > 0$.

$$\begin{array}{lcll} gcd\ a\ a & = & a & \\ gcd\ a\ b & = & gcd\ (a-b)\ b & \text{if } a > b \\ gcd\ a\ b & = & gcd\ a\ (b-a) & \text{if } b > a \end{array}$$

Why is this correct? First, the result of *gcd* a b is a divisor of both a and b. This is clearly true in the first clause. For the second clause, assume c is a common divisor of a and b. Then there are n and k such that $a = n \times c$ and $b = k \times c$. Then $a - b = (n - k) \times c$ (defined because $a > b$ and therefore $n > k$) so c still divides both $a - b$ and b. In the last clause the argument is symmetric. It remains to show that the function terminates, but this holds because the sum of the arguments to *gcd* becomes strictly smaller in each recursive call because $a, b > 0$.

While this function looks simple and elegant, it does not fit the schema of iteration or primitive recursion. The problem is that the recursive calls are not just on the immediate predecessor of an argument, but on the results of subtraction. So it might look like

$$f\,n = h\,n\,(f\,(g\,n))$$

but that doesn't fit exactly, either, because the recursive calls to *gcd* are on different functions in the second and third clauses.

So, let's be bold! The most general schema we might think of is

$$f = h\,f$$

which means that in the right-hand side we can make arbitrary recursive calls to f. For the *gcd*, the function h might look something like this:

$$\begin{array}{l} h = \lambda g.\,\lambda a.\,\lambda b.\ \textit{if}\ \ (a = b)\ a \\ \qquad\qquad\qquad\quad (\textit{if}\ \ (a > b)\ (g\,(a-b)\,b) \\ \qquad\qquad\qquad\qquad (g\,(b-a)\,b)) \end{array}$$

Here, we assume functions for testing $x = y$ and $x > y$ on natural numbers, for subtraction $x - y$ (assuming $x > y$) and for conditionals (see Exercise L1.4).

The interesting question now is if we can in fact define an f explicitly when given h so that it satisfies $f = h\,f$. We say that f is a *fixed point* of h, because when we apply h to f we get f back. Since our solution should be in the λ-calculus, it would be $f =_\beta h\,f$. A function f satisfying such an equation may *not* be uniquely determined. For example, the equation $f = f$

(so, $h = \lambda x.x$) is satisfied by every function f. On the other hand, if h is a constant function such as $\lambda x.I$ then $f =_\beta (\lambda x.\, I)\, f =_\beta I$ has a simple unique solution. For the purpose of this lecture, any function that satisfies the given equation is acceptable.

If we believe in the Church-Turing thesis, then any partial recursive function should be representable on Church numerals in the λ-calculus, so there is reason to hope there are explicit representations for such f. The answer is given by the so-called Y combinator.[1] Before we write it out, let's reflect on which laws Y should satisfy? We want that if $f = Y\, h$ and we specified that $f = h\, f$, so we get $Y\, h = h\, (Y\, h)$. We can iterate this reasoning indefinitely:

$$Y\, h = h\, (Y\, h) = h\, (h\, (Y\, h)) = h\, (h\, (h\, (Y\, h))) = \ldots$$

In other words, Y must iterate its argument arbitrarily many times.

The ingenious solution deposits one copy of h and the replicates $Y\, h$.

$$Y = \lambda h.\, (\lambda x.\, h\, (x\, x))\, (\lambda x.\, h\, (x\, x))$$

Here, the application $x\, x$ takes care of replicating $Y\, h$, and the outer application of h in $h\, (x\, x)$ leaves a copy of h behind. Formally, we calculate

$$\begin{array}{lll} Y\, h & =_\beta & (\lambda x.\, h\, (x\, x))\, (\lambda x.\, h\, (x\, x)) \\ & =_\beta & h\, ((\lambda x.\, h\, (x\, x))\, (\lambda x.\, h\, (x\, x))) \\ & =_\beta & h\, (Y\, h) \end{array}$$

In the first step, we just unwrap the definition of Y. In the second step we perform a β-reduction, substituting $[(\lambda x.\, h\, (x\, x))/x]\, h\, (x\, x)$. In the third step we recognize that this substitution recreated a copy of $Y\, h$.

You might wonder how we could ever get an answer since

$$Y\, h =_\beta h\, (Y\, h) =_\beta h\, (h\, (Y\, h)) =_\beta h\, (h\, (h\, (Y\, h))) = \ldots$$

Well, we sometimes don't! Actually, this is important if we are to represent *partial recursive functions* which include functions that are undefined (have no normal form) on some arguments. Reconsider the specification $f = f$ as a recursion schema. Then $h = \lambda g.\, g$ and

$$Y\, h = Y\, (\lambda g.\, g) =_\beta (\lambda x.\, (\lambda g.\, g)\, (x\, x))\, (\lambda x.\, (\lambda g.\, g)\, (x\, x)) =_\beta (\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$$

The term on the right-hand side here (called Ω) has the remarkable property that it only reduces to itself! It therefore does not have a normal form. In

[1] For our purposes, a *combinator* is simply a λ-expression without any free variables.

other words, the function $f = Y\,(\lambda g.\, g) = \Omega$ solves the equation $f = f$ by giving us a result which always diverges.

We do, however, sometimes get an answer. Consider, for example, a case where f does not call itself recursively at all: $f = \lambda n.\, \mathit{succ}\ n$. Then $h_0 = \lambda g.\, \lambda n.\, \mathit{succ}\ n$. And we calculate further

$$\begin{array}{lcl} Y\ h_0 & = & Y\,(\lambda g.\, \lambda n.\, \mathit{succ}\ n) \\ & =_\beta & (\lambda x.\, (\lambda g.\, \lambda n.\, \mathit{succ}\ n)\,(x\,x))\,(\lambda x.\, (\lambda g.\, \lambda n.\, \mathit{succ}\ n)\,(x\,x)) \\ & =_\beta & (\lambda x.\, (\lambda n.\, \mathit{succ}\ n))\,(\lambda x.\, (\lambda n.\, \mathit{succ}\ n)) \\ & =_\beta & \lambda n.\, \mathit{succ}\ n \end{array}$$

So, fortunately, we obtain just the successor function *if we apply β-reduction from the outside in*. It is however also the case that there is an infinite reduction sequence starting at $Y\ h_0$. By the Church-Rosser Theorem (Theorem L2.3) this means that at any point during such an infinite reduction sequence we could still also reduce to $\lambda n.\, \mathit{succ}\ n$. A remarkable and nontrivial theorem about the λ-calculus is that if we always reduce the left-most/outer-most redex (which is the first expression of the form $(\lambda x.\, e_1)\, e_2$ we come to when reading an expression from left to right) then we will definitely arrive at a normal form when one exists. And by the Church-Rosser theorem such a normal form is unique (up to renaming of bound variables, as usual).

3 Defining Functions by Recursion

As a simpler example than *gcd*, consider the factorial function, which we deliberately write using general recursion rather than primitive recursion.

$\mathsf{fact}\ n = \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n * \mathsf{fact}(n-1)$

To write this in the λ-calculus we first define a zero test *if0* satisfying

$$\begin{array}{l} \mathit{if0}\ \overline{0}\ c\ d = c \\ \mathit{if0}\ \overline{n+1}\ c\ d = d \end{array}$$

which is a special case of if iteration and can be written, for example, as

$\mathit{if0} = \lambda n.\, \lambda c.\, \lambda d.\, n\,(K\ d)\, c$

Eliminating the mathematical notation from the recursive definition of fact get the equation

$\mathsf{fact} = \lambda n.\, \mathit{if0}\ n\ (\mathit{succ}\ \mathit{zero})\ (\mathit{times}\ n\ (\mathsf{fact}\ (\mathit{pred}\, n)))$

where we have already defined *succ*, *zero*, *times*, and *pred*. Of course, this is not directly allowed in the λ-calculus since the right-hand side mentions fact which we are just trying to define. The function h_{fact} which will be the argument to the Y combinator is then

$$h_{\mathsf{fact}} = \lambda f.\, \lambda n.\, \mathit{if0}\; n\; (\mathit{succ}\; \mathit{zero})\; (\mathit{times}\; n\; (f\; (\mathit{pred}\; n)))$$

and

$$\mathit{fact} = Y\; h_{\mathsf{fact}}$$

We can write and execute this now in LAMBDA notation (see file nat.lam)

```
1  defn I = \x. x
2  defn K = \x. \y. x
3  defn Y = \h. (\x. h (x x)) (\x. h (x x))
4
5  defn if0 = \n. \c. \d. n (K d) c
6
7  defn h_fact = \f. \n. if0 n (succ zero) (times n (f (pred n)))
8  defn fact = Y h_fact
9
10 norm _120 = fact _5
11 norm _720 = fact (succ _5)
```

Listing 1: Recursive factorial in LAMBDA label

4 Introduction to Types

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 than

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.

5 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.

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

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.

6 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

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

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

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.

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

In this construction we exploit that the rules for typing are *syntax-directed*: for every form of expression there is exactly one rule we can use to infer its type.

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:

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

A final observation here about type variables: if $\cdot \vdash e : \alpha \to (\beta \to \beta)$ then also $\cdot \vdash e : \tau_1 \to (\tau_2 \to \tau_2)$ for any types τ_1 and τ_2. In other words, we can *substitute* arbitrary types for type variables in a typing judgment $\Gamma \vdash e : \tau$ and still get a valid judgment. In particular, the expressions *true* and *false* have *infinitely many types*.

7 Type Inference

An important property of the typing rules we have so far is that they are *syntax-directed*, that is, for every form of expression there is exactly one

typing rule that can be applied. We then perform *type inference* by constructing the skeleton of the typing derivation, filling it with *unknown types*, and reading off a set of equations that have be satisfied between the unknowns. Fortunately, these equations are relatively straightforward to solve with an algorithm called *unification*. This is the core of what is used in the implementation of modern functional languages such as Standard ML, OCaml, or Haskell.

We sketch how this process work, but only for a specific example; we might return to the general algorithm form in a future lecture. Consider the representation of 2:

$$\lambda s.\, \lambda z.\, s\,(s\,z)$$

We know it must have type $?\tau_1 \to (?\tau_2 \to ?\tau_3)$ for some unknown types $?\tau_1$, $?\tau_2$, and $?\tau_3$ where

$$s : ?\tau_1, z : ?\tau_2 \vdash s\,(s\,z) : ?\tau_3$$

Now, $s\,z$ applies s to z, so $?\tau_1 = ?\tau_2 \to ?\tau_4$ for some new $?\tau_4$. Next, the s is applied to the result of $s\,z$, so $?\tau_4 = ?\tau_2$. Also, the right-hand side is the same as the result type of s, so $?\tau_3 = ?\tau_4 = ?\tau_2$. Substituting everything out, we obtain

$$s : ?\tau_2 \to ?\tau_2, z : ?\tau_2 \vdash s\,(s\,z) : ?\tau_2$$

It is straightforward to write down the typing derivation for this judgment. Also, because we did not need to commit to what $?\tau_2$ actually is, we obtain

$$\lambda s.\, \lambda z.\, s\,(s\,z) : (\tau_2 \to \tau_2) \to (\tau_2 \to \tau_2) \qquad \text{for any type } \tau_2$$

We can express this by using a type variable instead, writing

$$\lambda s.\, \lambda z.\, s\,(s\,z) : (\alpha \to \alpha) \to (\alpha \to \alpha) \qquad \text{for any type } \alpha$$

because if the the type of an expression contains type variables we can alway substitute arbitrary types for them and still obtain a valid type.

We find that

$$\vdash \overline{n} : (\alpha \to \alpha) \to (\alpha \to \alpha)$$

even though some of the representations (such as $\overline{0} = \mathit{zero}$) also have other types. So our current hypothesis is that this type is a good candidate as a characterization of Church numerals, just as $\alpha \to (\alpha \to \alpha)$ is a characterization of the Booleans.

Exercises

Exercise 1 The unary representation of natural numbers requires tedious and error-prone counting to check whether your functions (such a factorial, Fibonacci, or greatest common divisor in the exercises below) behave correctly on some inputs with large answers. Fortunately, you can exploit that the LAMBDA implementation counts the number or reduction steps for you and prints it in decimal form!

(i) We have

$$\overline{n}\ \mathit{succ}\ \mathit{zero} \longrightarrow_{\beta}^{*} \overline{n}$$

because $\overline{n}$ iterates the successor function n times on 0. Run some experiments in LAMBDA and conjecture how many leftmost-outermost reduction steps are required as a function of n. Note that only β-reductions are counted, and *not* replacing a definition (for example, *zero* by $\lambda s.\, \lambda z.\, z$). We justify this because we think of the definitions as taking place at the metalevel, in our mathematical domain of discourse.

(ii) Prove your conjecture from part (i), using induction on n. It may be helpful to use the mathematical notation $f^k c$ to describe a λ-expression generated by $f^0\, c = c$ and $f^{k+1}\, c = f\,(f^k\, c)$ where f and c are λ-expressions. For example, $\overline{n} = \lambda s.\, \lambda z.\, s^n\ z$ or $\mathit{succ}^3\ \mathit{zero} = \mathit{succ}\ (\mathit{succ}\ (\mathit{succ}\ \mathit{zero}))$.

Exercise 2 Give an implementation of the factorial function in the λ-calculus as it arises from the schema of primitive recursion. How many β-reduction steps are required for factorial of 0, 1, 2, 3, 4, 5 in each of the two implementations?

Exercise 3 The Fibonacci function is defined by

$$\begin{array}{lcl} \mathsf{fib}\ 0 & = & 0 \\ \mathsf{fib}\ 1 & = & 1 \\ \mathsf{fib}\ (n+2) & = & \mathsf{fib}\ n + \mathsf{fib}\ (n+1) \end{array}$$

Give two implementations of the Fibonacci function in the λ-calculus (using the LAMBDA implementation). You may use the functions in (see file nat.lam).

(i) Exploit the idea behind the encoding of primitive recursion using pairs to give a direct implementation of fib without using the Y combinator.

(ii) Give an implementation of fib using the Y combinator.

Test your implementation on inputs 0, 1, 9, and 11, expecting results 0, 1, 34, and 89. Which of the two is more "efficient" (in the sense of number of β-reductions)?

Exercise 4 Recall the specification of the greatest common divisor (*gcd*) from this lecture for natural numbers $a, b > 0$:

$$
\begin{array}{lcll}
gcd\ a\ a & = & a & \\
gcd\ a\ b & = & gcd\ (a-b)\ b & \text{if } a > b \\
gcd\ a\ b & = & gcd\ a\ (b-a) & \text{if } b > a
\end{array}
$$

We don't care how the function behaves if $a = 0$ or $b = 0$.

Define *gcd* as a closed expression in the λ-calculus over Church numerals. You may use the Y combinator we defined, and any other functions like *succ*, *pred*, and you should define other functions you may need such as subtraction or arithmetic comparisons.

Also analyze how your function behaves when one or both of the arguments a and b are $\overline{0}$.

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 Representation Theorems

15-814: Types and Programming Languages
Frank Pfenning

Lecture 4
September 10, 2020

1 The Limits of Simple Types

We have proposed types as a way to classify functions, fixing their domain and their codomain, and making sure that functions are applied to arguments of the correct type. We also started to observe some patterns, such as *true* : $\alpha \to (\alpha \to \alpha)$ and *false* : $\alpha \to (\alpha \to \alpha)$, possibly using this type to characterize Booleans.

But what do we give up? Are there expressions that cannot be typed? From the historical perspective, this should definitely be the case, because types were introduced exactly to rule out certain "paradoxical" terms such as Ω, which does not have a normal form.

One term that is no longer typeable is self-application $\omega = \lambda x.\, x\, x$. As a result, we also can type neither $\Omega = \omega\, \omega$ nor Y, which can be seen as achieving a goal from the logical perspective, but it does give up computational expressiveness. How do we prove that ω cannot be typed? We begin by creating the skeleton of a typing derivation, which is unique due to the syntax-directed nature of the rules (that is, for each language construct there is exactly one typing rule). We highlight in red rules whose constraints on types have not yet been considered. When all the rules are black, we know that every solution to the accumulated constraints leads to a valid typing

derivation (and therefore a valid type in the conclusion).

$$\dfrac{\dfrac{\dfrac{}{x : \boxed{} \vdash x : \boxed{}}\;\textsf{tp/var} \qquad \dfrac{}{x : \boxed{} \vdash x : \boxed{}}\;\textsf{tp/var}}{x : \boxed{} \vdash x\,x : \boxed{}}\;\textsf{tp/app}}{\cdot \vdash \lambda x.\,x\,x : \boxed{}}\;\textsf{tp/lam}$$

The type in the final judgment must be $?\tau_1 \to ?\tau_2$ for some types $?\tau_1$ and $?\tau_2$.

$$\dfrac{\dfrac{\dfrac{}{x : \boxed{} \vdash x : \boxed{}}\;\textsf{tp/var} \qquad \dfrac{}{x : \boxed{} \vdash x : \boxed{}}\;\textsf{tp/var}}{x : \boxed{?\tau_1} \vdash x\,x : \boxed{?\tau_2}}\;\textsf{tp/app}}{\cdot \vdash \lambda x.\,x\,x : \boxed{?\tau_1 \to ?\tau_2}}\;\textsf{tp/lam}$$

Once the type of a variable is available in the context, this types is propagated upwards unchanged in a derivation, so we can fill in some more of the types.

$$\dfrac{\dfrac{\dfrac{}{x : \boxed{?\tau_1} \vdash x : \boxed{}}\;\textsf{tp/var} \qquad \dfrac{}{x : \boxed{?\tau_1} \vdash x : \boxed{}}\;\textsf{tp/var}}{x : \boxed{?\tau_1} \vdash x\,x : \boxed{?\tau_2}}\;\textsf{tp/app}}{\cdot \vdash \lambda x.\,x\,x : \boxed{?\tau_1 \to ?\tau_2}}\;\textsf{tp/lam}$$

In the tp/var rules the type of variable is just looked up in the context, so we can fill in those two types as well.

$$\dfrac{\dfrac{\dfrac{}{x : \boxed{?\tau_1} \vdash x : \boxed{?\tau_1}}\;\textsf{tp/var} \qquad \dfrac{}{x : \boxed{?\tau_1} \vdash x : \boxed{?\tau_1}}\;\textsf{tp/var}}{x : \boxed{?\tau_1} \vdash x\,x : \boxed{?\tau_2}}\;\textsf{tp/app}}{\cdot \vdash \lambda x.\,x\,x : \boxed{?\tau_1 \to ?\tau_2}}\;\textsf{tp/lam}$$

Finally, for the application of the tp/app rule to be correct, the type of x in the first premise must be a function type, expecting an argument of type $?\tau_1$

(the type of x in the second premise) and returning a result of type $?\tau_2$. That is:

$$\dfrac{\dfrac{\dfrac{}{x : \boxed{?\tau_1} \vdash x : \boxed{?\tau_1}}\,\textsf{tp/var} \qquad \dfrac{}{x : \boxed{?\tau_1} \vdash x : \boxed{?\tau_1}}\,\textsf{tp/var}}{x : \boxed{?\tau_1} \vdash x\,x : \boxed{?\tau_2}}\,\textsf{tp/app}}{\cdot \vdash \lambda x.\, x\,x : \boxed{?\tau_1 \to ?\tau_2}}\,\textsf{tp/lam}$$

provided $?\tau_1 = ?\tau_1 \to ?\tau_2$

Now we observe that there cannot be a solution to the required equation: there are no types τ_1 and τ_2 such that $\tau_1 = \tau_1 \to \tau_2$ since the right-hand side is always bigger (and therefore not equal) to the left-hand side.

To recover from this in full generality we would need so-called *recursive types*. In this example, we see

$$\tau_1 = F\,\tau_1$$

where $F = \lambda\alpha.\, \alpha \to \tau_2$ and we might then have a solution with $\tau_1 = Y\,F$. But such a solution is not immediately available to us. For one thing, we do not have function from types to types such as F. For another, we don't have a Y combinator at the level of types. However, it is perfectly possible to construct recursive types, and we will do so later in the course. In the notation we will introduce later, we would get

$$\tau_1 = \rho\alpha.\, \alpha \to \tau_2$$

where $\rho\alpha.\,\tau$ binds the type variable α with scope τ. Such a type will be equivalent to its unfolding $[\rho\alpha.\,\tau/\alpha]\tau$.

Another way to recover some, but not all of the functions that can be typed in the λ-calculus is to introduce *polymorphism*, which we will also consider.

2 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* $e = \mathit{true}$ *or* $e = \mathit{false}$.

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* $e =_\beta \mathit{true}$ *or* $e =_\beta \mathit{false}$.

By the Church-Rosser Theorem, if $e =_\beta e'$ where e' is a normal form (that is, cannot be reduced), then $e \longrightarrow_\beta^* e'$ so we can replace this by

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

This is actually quite difficult to prove. In particular, it requires that every expression of the given type does have a normal form. We have already seen that the standard divergent term Ω does not have a type, and neither does the Y combinator. In fact, it will turn out (although with a difficult proof) that every simple-typed λ-expression does have a normal form! This is commonly called the *weak normalization* property. *Strong normalization* requires that every reduction sequence terminates, which, incidentally, also holds here.

Fortunately, we can prove simpler theorems that do not directly rely on normalization. The first one concerns only *normal forms*, that is, expressions that cannot be β-reduced. They play the role that *values* play in many programming languages.

Conjecture 4 (Representation of Booleans, v4)
If $\cdot \vdash e : \alpha \to (\alpha \to \alpha)$ *and* e *is a normal form, then* $e = \mathit{true}$ *or* $e = \mathit{false}$.

We will later combine this with the following theorems which yields 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 (Weak Normalization) *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* $\Gamma \vdash e' : \tau$.

3 Reduction Revisited

Our characterization of normal forms so far is quite simple: they are terms that do not reduce. But this is a *negative* condition, and negative conditions can be difficult to work with in proofs. So we would like a *positive* definition normal forms. Just like typing, we tend to give such definitions in the form of inference rules. The property then holds if the judgment of interest (here, that an expression is normal) can be derived using the given rules. This is closely related to the notion of *inductive definition*.

Before we get to defining normal forms by rules, we formally define β-reduction by inference rules. Previously, we just stated informally that a step of β-reduction can be "applied anywhere in an expression". Now we write this out. We refer to the last three rules as *congruence rules* because they allow the reduction of a subterm. The judgment is here $e \longrightarrow e'$ (omitting the β for brevity) expressing that e reduces to e'.

$$\frac{}{(\lambda x.\, e_1)\, e_2 \longrightarrow [e_2/x]e_1}\ \mathsf{red/beta} \qquad \frac{e \longrightarrow e'}{\lambda x.\, e \longrightarrow \lambda x.\, e'}\ \mathsf{red/lam}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2}\ \mathsf{red/app}_1 \qquad \frac{e_2 \longrightarrow e_2'}{e_1\, e_2 \longrightarrow e_1\, e_2'}\ \mathsf{red/app}_2$$

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 so we can easily prove inductive theorems on them. We might start with the following incorrect attempt:

$$\frac{}{x\ \mathit{normal}}\ \mathsf{norm/var} \qquad \frac{e\ \mathit{normal}}{\lambda x.\, e\ \mathit{normal}}\ \mathsf{norm/lam}$$

$$\frac{e_1\ \mathit{normal} \quad e_2\ \mathit{normal}}{e_1\, e_2\ \mathit{normal}}\ \mathsf{norm/app}$$

It is easy to see that under such a definition *every* term would be normal. The culprit here is the rule of application, because, for example, in the application $(\lambda x.\, x)\, (\lambda y.\, y)$ both function and argument are normal, but their term itself is not. So we need a separate judgment for *neutral* terms which

do *not* create a redex when they are applied to an argument. In particular, a λ-abstraction is *not* neutral, but a variable is. Then $e_1\, e_2$ is normal if e_1 is neutral and e_2 is normal.

$$\frac{e\ \mathit{normal}}{\lambda x.\, e\ \mathit{normal}}\ \mathsf{norm/lam} \qquad \frac{e\ \mathit{neutral}}{e\ \mathit{normal}}\ \mathsf{norm/neut}$$

$$\frac{}{x\ \mathit{neutral}}\ \mathsf{neut/var} \qquad \frac{e_1\ \mathit{neutral} \quad e_2\ \mathit{normal}}{e_1\, e_2\ \mathit{neutral}}\ \mathsf{neut/app}$$

This definition captures terms of 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. It is not strictly syntax-directed in the given form because, for a λ-abstraction, both rules norm/lam and norm/neut could be used. However, norm/neut will fail immediately in the next step, so we only need to "look ahead" one rule to make the construction deterministic.

As an example, to show that $\lambda x.\, x\, x$ *normal* we construct the following derivation, starting from the bottom.

$$\cfrac{\cfrac{\cfrac{\cfrac{}{x\ \mathit{neutral}}\ \mathsf{neut/var} \quad \cfrac{\cfrac{}{x\ \mathit{neutral}}\ \mathsf{neut/var}}{x\ \mathit{normal}}\ \mathsf{norm/neut}}{x\, x\ \mathit{neutral}}\ \mathsf{neut/app}}{x\, x\ \mathit{normal}}\ \mathsf{norm/neut}}{\lambda x.\, x\, x\ \mathit{normal}}\ \mathsf{norm/lam}$$

4 Normal Forms and Reduction

The characterization of normal forms via inference rules is compact, but is it really the same as saying that an expression does not reduce? We would like to work as much as possible with positive characterizations, so we break this down into the following two properties

1. For all expressions e, either e reduces or e is normal.

2. For all expressions e, it is not that case that e reduces and e is normal.

The second property just states that the "either/or" in part 1 is an exclusive or. We will prove the first, and leave the second as Exercise **??**.

To make the proof just a bit easier to write, we introduce a new judgment $e \longrightarrow$ expressing that e reduces, but we do not care what to. We obtain it by erasing the right-hand sides of all the reduction rules. It is then immediate (although formally done by induction) that $e \longrightarrow e'$ for some e' iff $e \longrightarrow$.

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

$$\frac{e \longrightarrow}{\lambda x.\, e \longrightarrow}\ \mathsf{rbl/lam} \qquad \frac{e_1 \longrightarrow}{e_1\, e_2 \longrightarrow}\ \mathsf{rbl/app}_1 \qquad \frac{e_2 \longrightarrow}{e_1\, e_2 \longrightarrow}\ \mathsf{rbl/app}_2$$

Theorem 7 (Reduction and normal forms, Part (i))
For every expression e, either $e \longrightarrow$ or e normal.

Proof: We are only given an expression e, so the proof is likely by induction on the structure of e. Such a proof has the following parts:

(i) We have to establish the property outright for $e = x$.

(ii) We have to establish the property for $e = \lambda x.\, e_1$, where the induction hypothesis is the property for e_1.

(iii) We have the establish the property for $e = e_1\, e_2$ where the induction hypotheses are the properties for e_1 and e_2.

If we can cover all three cases we know that the property must hold for all expressions. Let's try!

Case: $e = x$. Then

x *neutral*	By rule neut/var
x *normal*	By rule norm/neut

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

Either $e_1 \longrightarrow$ or e_1 *normal*	By ind.hyp. on e_1
$e_1 \longrightarrow$	First subcase
$e = \lambda x.\, e_1 \longrightarrow$	By rule rbl/lam
e_1 *normal*	Second subcase
$e = \lambda x.\, e_1$ *normal*	By rule norm/lam

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

Either $e_1 \longrightarrow$ or e_1 *normal*	By ind.hyp. on e_1
$e_1 \longrightarrow$	First subcase
$e_1\, e_2 \longrightarrow$	By rule rbl/app$_1$
e_1 *normal*	Second subcase
Either $e_1 = \lambda x.\, e_1'$ and e_1' *normal*	
or e_1 *neutral*	By inversion on e_1 *normal*
$e_1 = \lambda x.\, e_1'$	First sub^{2}case
$e = e_1\, e_2 = (\lambda x.\, e_1')\, e_2 \longrightarrow$	By rule rbl/beta
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 rbl/app$_2$
e_2 *normal*	Second sub^{3}case
$e = e_1\, e_2$ *neutral*	By rule neut/app

□

This proof is slightly shorter from the proof we did in lecture, using an inversion step that is highlighted in red. What are we doing in this step? We know we are in the "second subcase" so we have the knowledge that e_1 *normal*. Now we examine the inference rule and we see there are only two possible rules that could be used to conclude this judgment: *norm/lam* and *norm/neut*. So we can distinguish these to cases. In each case, we also know that the premise must hold to obtain the (known) conclusion.

This step in a proof is called *inversion* because we infer, at the metalevel at which we reason about our judgments, that the premise of a rule must

hold if the conclusion does. This is only valid if we consider all the possible cases, of which there are two in this particular situation. Often, there is only one, and sometimes there is none (which means that the case were are in is actually impossible).

Now that we have characterized normal forms, we will be able to prove a representation theorem for Booleans in the next lecture.

Exercises

Exercise 1 Fill in the blanks in the following typing judgments so the resulting judgment holds, or indicate there is no way to do so. You do not need to justify your answer or supply a typing derivation, and the types do not need to be "most general" in any sense. Remember that the function type constructor associates to the right, so that $\tau \to \sigma \to \rho = \tau \to (\sigma \to \rho)$.

(i) $\boxed{} \vdash y\,x : \alpha$

(ii) $\boxed{} \vdash x\,x : \boxed{}$

(iii) $\cdot \vdash \boxed{} : (\alpha \to \alpha) \to \alpha$

(iv) $\cdot \vdash (\lambda z.\, z)\,(\lambda x.\, \lambda y.\, \lambda p.\, p\,x\,y) : \boxed{}$

(v)

$$\begin{aligned} &\cdot \vdash \lambda f.\, \lambda g.\, \lambda x.\, (f\,x)\,(g\,x) \\ &\quad : (\alpha \to \boxed{}) \to (\alpha \to \boxed{}) \to (\alpha \to \boxed{}) \end{aligned}$$

Since this is the first time we (that is, you) are proving theorems about judgments defined by rules, we ask you to be very explicit, as we were in the lectures and lecture notes. In particular:

- Explicitly state the overall structure of your proof: whether it proceeds by rule induction, and, if so, on the derivation of which judgment, or by structural induction, or by inversion, or just directly. If you need to split out a lemma for your proof, state it clearly and prove it separately. If you need to generalize your induction hypothesis, clearly state the generalized form.

- Explicitly list all cases in an induction proof. If a case is impossible, prove that is is impossible. Often, that's just inversion, but sometimes it is more subtle.

- Explicitly note any appeals to the induction hypothesis.
- Any appeals to inversion should be noted as such, as well as the rules that could have inferred the judgment we already know. This could lead to zero cases (a contradiction—the judgment could not have been derived), one case (there is exactly one rule whose conclusion matches our knowledge), or multiple cases, in which case your proof now splits into multiple cases.
- We recommend that you follow the line-by-line style of presentation where each line is justified by a short phrase. This will help you to check your proof and us to read and verify it.

Exercise 2 Prove that there does not exist an expression e such that $e \longrightarrow$ and e *normal*. In other words, the alternatives stated in Theorem **??** are exclusive.

As a reminder, the way we prove that a proposition A is false is to assume A is true and derive a contradiction. For those who care about such things, this is a perfectly valid intuitionistic (constructive) reasoning principle, as opposed to an *indirect proof*. The rule of indirect proof (which we should avoid at all cost, since all proofs in this course should be constructive) says that we can prove A is true by assuming that A is false and then deriving a contradiction from that.

Typing $\Gamma \vdash e : \tau$

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

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

Reduction $e \longrightarrow e'$ and $e \longrightarrow^* e'$

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

$$\frac{e \longrightarrow e'}{\lambda x.\, e \longrightarrow \lambda x.\, e'}\ \textsf{red/lam} \qquad \frac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2}\ \textsf{red/app}_1 \qquad \frac{e_2 \longrightarrow e_2'}{e_1\, e_2 \longrightarrow e_1\, e_2'}\ \textsf{red/app}_2$$

$$\frac{}{e \longrightarrow^* e}\ \textsf{red}^*\textsf{/refl} \qquad \frac{e \longrightarrow e' \quad e' \longrightarrow^* e''}{e \longrightarrow^* e''}\ \textsf{red}^*\textsf{/step}$$

Reducible $e \longrightarrow$

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

$$\frac{e \longrightarrow}{\lambda x.\, e \longrightarrow}\ \textsf{rbl/lam} \qquad \frac{e_1 \longrightarrow}{e_1\, e_2 \longrightarrow}\ \textsf{rbl/app}_1 \qquad \frac{e_2 \longrightarrow}{e_1\, e_2 \longrightarrow}\ \textsf{rbl/app}_2$$

Normal and Neutral Expressions e *normal* and e *neutral*

$$\frac{e\ \mathit{normal}}{\lambda x.\, e\ \mathit{normal}}\ \textsf{norm/lam} \qquad \frac{e\ \mathit{neutral}}{e\ \mathit{normal}}\ \textsf{norm/neut}$$

$$\frac{}{x\ \mathit{neutral}}\ \textsf{neut/var} \qquad \frac{e_1\ \mathit{neutral} \quad e_2\ \mathit{normal}}{e_1\, e_2\ \mathit{neutral}}\ \textsf{neut/app}$$

Lecture Notes on Subject Reduction

15-814: Types and Programming Languages
Frank Pfenning

Lecture 5
September 15, 2020

1 Introduction

In the last lecture we laid the groundwork for a *representation theorem* on Booleans, which will prove in this lecture. This provides a clear relationship between normal forms and one particular type and is an exemplar of many similar theorems characterizing the normal forms of given types.

In the second part of the this lecture we establish a relationship between *computation* and types, complementing the relation between *normal forms* and types. The essence of this is that if $\Gamma \vdash e : \tau$ and $e \longrightarrow e'$ then $\Gamma \vdash e' : \tau$.

The third part (which we already proved) is that every well-typed expression either reduces or is a normal form so there is no "loophole" in the type system.

Together, these three parts form the basic pillars for interpreting the meaning of types in programming languages, studied here in the setting of the simply-typed λ-calculus which we can think of as a proto-programming-language, and which we will find embedded in richer and more practical languages.

2 A Representation Theorem for Booleans

Theorem 1 (Representation of Booleans, v4) *If* $\cdot \vdash e : \alpha \to (\alpha \to \alpha)$ *and* e *normal then* $e = \mathit{true} = \lambda x.\, \lambda y.\, x$ *or* $e = \mathit{false} = \lambda x.\, \lambda y.\, y$.

We postpone the proof to first show an important lemma about *neutral terms* which will be used in the proof.

Lemma 2 (Neutrality) *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 $1 \leq i \leq n$.*

Proof: The intuition behind this theorem is that a neutral term e has the form $((x\, e_1) \ldots e_k)$ but there is no variable x that has a function type so $k = 0$ and $e = x$. But the only variables x in the context are $x_i : \alpha_i$.

There are essentially three different forms of induction we could apply here (abbreviating $\Gamma_0 = x_1 : \alpha_1, \ldots, x_n : \alpha_n$)

1. Over the structure of the expression e
2. Over the derivation of $\Gamma_0 \vdash e : \tau$
3. Over the derivation of e *neutral*

Generally, when we have additional information about an expression such as e, we rarely perform an induction over the structure of e, but we prefer to directly exploit the knowledge about e. Secondly (and also a heuristic), we can easily apply inversion to syntax-directed judgments such as typing, and less directly so for others. Therefore, we prefer rule induction over judgments *other* than typing.

More formally, we proceed by rule induction on e *neutral*. There are just two cases.

Case:

$$\frac{}{x \textit{ neutral}}\ \mathsf{neut/var}$$

where $e = x$. Then we reason

$x_1 : \alpha_1, \ldots, x_n : \alpha_n \vdash x : \tau$ — Assumption
$x = x_i$ and $\tau = \alpha_i$ for some $1 \leq i \leq n$ — By inversion

"Inversion" here refers to the fact that there is only one typing rule for variables, tp/var, and this rule requires x to be one of the variables in the context and τ to be the corresponding type.

Case:

$$\frac{e_1 \textit{ neutral} \quad e_2 \textit{ normal}}{e_1\, e_2 \textit{ neutral}}\ \mathsf{neut/app}$$

where $e = e_1\, e_2$. Then we reason

$\Gamma_0 \vdash e_1\, e_2 : \tau$ — Assumption
$\Gamma_0 \vdash e_1 : \tau_2 \to \tau$
and $\Gamma_0 \vdash e_2 : \tau_2$ for some τ_2 — By inversion
$e_1 = x_i$ and $\tau_2 \to \tau = \alpha_i$ for some $1 \leq i \leq n$ — By ind. hyp.
Contradiction — Since $\tau_2 \to \tau = \alpha_i$ is impossible

Therefore, the second case is impossible, as we already noted informally at the outset. The appeal to the induction hypothesis relies on the derivations of e_1 *neutral* and $\Gamma_0 \vdash e_1 : \tau_2 \to \tau$ and is correct because e_1 *neutral* is a subderivation (in fact, the immediate premise) of the given derivation for $e = e_1\, e_2$.

□

Now we are ready to tackle the proof of the representation theorem for normal forms.

Proof: (of Theorem 1) Let's remind ourselves:

If $\cdot \vdash e : \alpha \to (\alpha \to \alpha)$ *and* e *normal then* $e = \mathit{true} = \lambda x.\, \lambda y.\, x$ *or* $e = \mathit{false} = \lambda x.\, \lambda y.\, y$.

Again we have a choice: we could try induction over the structure of e (not a good idea), rule induction over the derivation of $\cdot \vdash e : \alpha \to (\alpha \to \alpha)$ (okay), or rule induction over e *normal* (even better). As it turns out, we can do a *proof by cases*, since the induction hypothesis is never needed! This is, of course, a special case of induction but we would like to be precise if a simpler proof principle suffices.

Case:

$$\frac{e\ \mathit{neutral}}{e\ \mathit{normal}}\ \mathsf{norm/neut}$$

We conclude that this case is impossible as follows:

$\cdot \vdash e : \alpha \to (\alpha \to \alpha)$ — Assumption
e *neutral* — Premise in this case
Contradiction — By Lemma 2

Case:

$$\frac{e_1\ \mathit{normal}}{\lambda x.\, e_1\ \mathit{normal}}\ \mathsf{norm/lam}$$

where $e = \lambda x.\, e_1$. We continue:

$\cdot \vdash \lambda x.\, e_1 : \alpha \to (\alpha \to \alpha)$ Assumption
$x : \alpha \vdash e_1 : \alpha \to \alpha$ By inversion
Either e_1 *neutral* or $e_1 = \lambda x.\, e_2$ for some e_2 and e_2 *normal*
By inversion on e_1 *normal*

Here the appeal to inversion yields two cases, because the conclusion e_1 *normal* could be derived by two different rules (norm/neut or norm/lam).

Subcase: e_1 *neutral*. Again, this case is impossible by neutrality.

$x : \alpha \vdash e_1 : \alpha \to \alpha$ From above
e_1 *neutral* This case
Contradiction By Lemma 2

Subcase: $e_1 = \lambda y.\, e_2$ for some e_2 and e_2 *normal*. Then

$x : \alpha \vdash \lambda y.\, e_2 : \alpha \to \alpha$ From above with $e_1 = \lambda y.\, e_2$
$x : \alpha, y : \alpha \vdash e_2 : \alpha$ By inversion
e_2 *normal* This subcase
$e_2 = \lambda z.\, e_3$ for some e_3 *normal*
or e_2 *neutral* By inversion on e_2 *normal*

We now distinguish the reasoning in these two subcases.

Sub^{2}case: $e_2 = \lambda z.\, e_3$ for some e_3 with e_3 *normal*. Now it is this case that is impossible:

$x : \alpha, y : \alpha \vdash \lambda z.\, e_3 : \alpha$ From above with $e_2 = \lambda x\, e_3$
Contradiction By inversion
(no typing rule matches this conclusion)

Sub^{2}case: e_2 *neutral*. Then

$x : \alpha, y : \alpha \vdash e_2 : \alpha$ From above
e_2 *neutral* This case
$e_2 = x$ or $e_2 = y$ By neutrality (Lemma 2)
$e = \lambda x.\, e_1 = \lambda x.\, \lambda y.\, e_2 = \lambda x.\, \lambda y.\, x$
or $e = \lambda x.\, e_1 = \lambda x.\, \lambda y.\, e_2 = \lambda x.\, \lambda y.\, y$
By form of e, e_1, and e_2 in this case

□

3 Subject Reduction

Let's put the representation theorem into the bigger picture. We had previously conjectured:

Conjecture 3 (L3.4, Representation of Booleans, v2)
If $\cdot \vdash e : \alpha \to (\alpha \to \alpha)$ *then* $e =_\beta$ *true or* $e =_\beta$ *false.*

But we want to relate this to computation. Fortunately, by the Church-Rosser Theorem, $e =_\beta e'$ for a normal form e' if and only if $e \longrightarrow^* e'$ (where $\longrightarrow^*$ is the reflexive and transitive closure of single-step reduction we have been mostly working with). So we recast this one more time, relating typing to computation and representation.

Conjecture 4 (Computation of Booleans)
If $\cdot \vdash e : \alpha \to (\alpha \to \alpha)$ *then* $e \longrightarrow^*$ *true or* $e \longrightarrow^*$ *false.*

Since every well-typed expression has a normal form (which we did not prove), the missing link in our reasoning chain is that typing is preserved under reduction: if we start with an expression e of type τ and we reduce it all the way to a normal form e', then 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.

Now we return to the main topic of this lecture, namely subject reduction. Recall our characterization of reduction:

$$\frac{e \longrightarrow e'}{\lambda x.\, e \longrightarrow \lambda x.\, e'}\ \mathsf{red/lam} \qquad \frac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2}\ \mathsf{red/app_1} \qquad \frac{e_2 \longrightarrow e_2'}{e_1\, e_2 \longrightarrow e_1\, e_2'}\ \mathsf{red/app_2}$$

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

And, for reference, here are the typing rules.

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

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

Theorem 5 (Subject Reduction)
If $\Gamma \vdash e : \tau$ *and* $e \longrightarrow e'$ *then* $\Gamma \vdash e' : \tau$.

Proof: In this theorem statement we are given derivations for two judgments: $\Gamma \vdash e : \tau$ and $e \longrightarrow e'$. Most likely, the proof will proceed by rule

induction on one of these and by inversion on the other. The typing judgment is syntax-directed and therefore amenable to reasoning by inversion, so we try rule induction over the reduction judgment.

By rule induction on the derivation of $e \longrightarrow e'$.

Case:

$$\frac{e_1 \longrightarrow e_1'}{\lambda x.\, e_1 \longrightarrow \lambda x.\, e_1'}\ \mathsf{red/lam}$$

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:

$$\frac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2}\ \mathsf{red/app}_1$$

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 e_1' has the type of e_1.

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

Case:

$$\frac{e_2 \longrightarrow e_2'}{e_1\, e_2 \longrightarrow e_1'\, e_2}\ \mathsf{red/app}_2$$

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* (Theorem 6)

□

Theorem 6 (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 $x_1 : \tau_1$ 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 $x_1 : \tau_1$ 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.

4 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

(i) For all e, either $e \longrightarrow$ or e *normal*.

(ii) There is no e such that $e \longrightarrow$ and e *normal*

Representation of Booleans in Normal Form (L5.1)
If $\cdot \vdash e : \alpha \to (\alpha \to \alpha)$ and e *normal* then either $e = \mathit{true} = \lambda x.\, \lambda y.\, x$ or $e = \mathit{false} = \lambda x.\, \lambda y.\, y$.

Subject Reduction (L5.5)
If $\Gamma \vdash e : \tau$ and $e \longrightarrow e'$ we have $\Gamma \vdash e' : \tau$.

We did not prove normalization (also called *termination*) or confluence (also called the Church-Rosser property).

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

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

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

Exercises

Exercise 1 Define multi-step reduction $e \longrightarrow^* e'$ by the following rules:

$$\frac{}{e \longrightarrow^* e}\ \mathsf{red^*/refl} \qquad \frac{e \longrightarrow e' \quad e' \longrightarrow^* e''}{e \longrightarrow^* e''}\ \mathsf{red^*/step}$$

Prove by rule induction that if $\Gamma \vdash e : \tau$ and $e \longrightarrow^* e'$ then $\Gamma \vdash e' : \tau$

Exercise 2 Define a new single-step relation $e \mapsto e'$ which means that e reduces to e' by *leftmost-outermost reduction*, using a collection of inference rules. Recall that I claimed this strategy is *sound* (it only performs β-reductions) and *complete for normalization* (if e has a normal form, we can reach it by performing only leftmost-outermost reductions). Prove the following statements about your reduction judgment:

(i) If $e \mapsto e'$ then $e \longrightarrow e'$.

(ii) $\mapsto$ is *small-step deterministic*, that is, if $e \mapsto e_1$ and $e \mapsto e_2$ then $e_1 = e_2$.

You should interpret $=$ as α-equality, that is, the two terms differ only in the names of their bound variables (which we always take for granted). For each of the following statements, either indicate that they are true (without proof) or provide a counterexample.

(iii) For all e, either $e \mapsto e'$ for some e' or e *normal*.

(iv) There does not exist an e such that $e \mapsto e'$ for some e' and e *normal*.

(v) If $e \longrightarrow e'$ then $e \mapsto e'$.

(vi) $\longrightarrow$ is *small-step deterministic*.

(vii) $\longrightarrow$ is *big-step deterministic*, that is, if $e \longrightarrow^* e_1$ and $e \longrightarrow^* e_2$ where e_1 *normal* and e_2 *normal*, then $e_1 = e_2$.

(viii) For arbitrary e and *normal* e', $e \longrightarrow^* e'$ iff $e \mapsto^* e'$.

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 Parametric Polymorphism

15-814: Types and Programming Languages
Frank Pfenning

Lecture 6
Thursday, September 17, 2020

1 Introduction

Polymorphism refers to the possibility of an expression to have multiple types. In that sense, the simply-typed λ-calculus is *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, we might want to state

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

to express all the types above, but now in a single form. This means we could 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* τ, ...". Our system will be slightly different from this, for reasons that will become apparent later.

Christopher Strachey [Str00] 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 *int* $\to$ *int* $\to$ *int* but also *float* $\to$ *float* $\to$ *float* with different implementations at these two types. Similarly, a function *show* : $\forall \alpha.\, \alpha \to$ *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 until the beautiful analysis by John Reynolds [Rey83], which we will sketch in a later lecture on parametricity. In this lecture we will present typing rules and some examples.

Slightly different systems for parametric polymorphism were discovered independently by Jean-Yves Girard [Gir71] and John Reynolds [Rey74]. Girard worked in the context of *logic* and developed *System F*, while Reynolds worked directly on *type systems* for programming language and designed the polymorphic λ-calculus. With minor syntactic changes, we will follow Reynolds's presentation.

2 Universally Quantified Types

We would like to add types of the form $\forall\alpha.\,\tau$ to express parametric polymorphism. The fundamental idea is that an expression of type $\forall\alpha.\,\tau$ is a *function* that takes a *type* as an argument.

This is a rather radical change of attitude. So far, our expressions contained no types at all, and suddenly types become embedded in expressions and are actually passed to functions! Let's see where it leads us. Now we *could* write

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

but abstraction over a type seems so different from abstraction over a expressions that we make up a new notation and instead write

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

using a capital lambda (Λ). In order to express the typing rules, our contexts carry two different forms of declarations: $x : \tau$ (as we had so far) and now also α *type*, expressing that α is a type variable. The typing judgment then is still $\Gamma \vdash e : \tau$, without repeated variables or type variables in Γ. There will be some further presuppositions mentioned later. For type abstractions, we have the rule

$$\frac{\Gamma, \alpha\ \mathit{type} \vdash e : \tau}{\Gamma \vdash \Lambda\alpha.\,e : \forall\alpha.\,\tau}\ \mathsf{tp/tplam}$$

Here, α is a bound variable in $\Lambda\alpha.\, e$ and $\forall\alpha.\, \tau$ so we allow it to be silently renamed if it conflicts with any variable already declared in Γ.

We haven't yet seen how α can actually appear in e, but we can already verify:

$$\dfrac{\dfrac{\dfrac{}{\alpha\ \mathit{type}, x : \alpha \vdash x : \alpha}\ \mathsf{var}}{\alpha\ \mathit{type} \vdash \lambda x.\, x : \alpha \to \alpha}\ \mathsf{lam}}{\cdot \vdash \Lambda\alpha.\, \lambda x.\, x : \forall\alpha.\, \alpha \to \alpha}\ \mathsf{tp/tplam}$$

The next question is how do we *apply* such a polymorphic function to a type? Again, we *could* just write $e\ \tau$ for the application of a polymorphic function e to a type τ, but we would like it to be more syntactically apparent so we write $e\,[\tau]$.

Let's return to Church's representation of natural numbers. With the quantifier, we now have

$$\mathit{nat} = \forall\alpha.\, (\alpha \to \alpha) \to \alpha \to \alpha$$

Then we can verify with typing derivations as above:

$$\begin{array}{lcl} \mathit{zero} & : & \mathit{nat} \\ \mathit{zero} & = & \Lambda\alpha.\, \lambda s.\, \lambda z.\, z \end{array}$$

We also expect the successor function to have type $\mathit{nat} \to \mathit{nat}$, but there is one slightly tricky spot. We start:

$$\begin{array}{lcl} \mathit{succ} & : & \mathit{nat} \to \mathit{nat} \\ \mathit{succ} & = & \lambda n.\, \Lambda\alpha.\, \lambda s.\, \lambda z.\, s\,(n\ \boxed{}\,) \end{array}$$

Before, we just applied n to s and z, but now $n : \mathit{nat}$, which means that it expects a *type* as its first argument! At this point (in a hypothetical typing derivation we did not write out) we have the context

$$n : \mathit{nat}, \alpha\ \mathit{type}, s : \alpha \to \alpha, z : \alpha$$

so we need to instantiate the quantifier with α, which next requires arguments of type $\alpha \to \alpha$ and α (which we have at hand with s and z).

$$\begin{array}{lcl} \mathit{succ} & : & \mathit{nat} \to \mathit{nat} \\ \mathit{succ} & = & \lambda n.\, \Lambda\alpha.\, \lambda s.\, \lambda z.\, s\,(n\,[\alpha]\,s\,z) \end{array}$$

It becomes more interesting with the addition function. Recall that in the untyped setting we had

$$plus = \lambda n.\, \lambda k.\, n\ succ\ k$$

iterating the successor function n times on argument k. The start of the typed version is again relatively straightforward: the only difference is that we need to apply n first to a type.

$$\begin{array}{lcl} plus & : & nat \to nat \to nat \\ plus & = & \lambda n.\, \lambda k.\, n\, [\boxed{}]\ succ\ k \end{array}$$

But what type do we need? We have that the next argument has type $nat \to nat$ and the following one nat, so that we need to instantiate α with *nat*!

$$\begin{array}{lcl} plus & : & nat \to nat \to nat \\ plus & = & \lambda n.\, \lambda k.\, n\, [nat]\ succ\ k \end{array}$$

So we need that

$$n : \forall \alpha.\, (\alpha \to \alpha) \to \alpha \to \alpha$$

and then

$$n\, [nat] : (nat \to nat) \to nat \to nat$$

We should point out that this definition of addition cannot be typed in the simply-typed λ-calculus. In that setting, n can only be applied to functions s of type $\alpha \to \alpha$ to iterate starting from $z : \alpha$. This means that very few functions are actually definable—essentially only functions like successor and addition, but not exponentiation, or predecessor (see Exercise 1).

A significant aspect of this is that we instantiate the quantifier in $nat = \forall \alpha.\, (\alpha \to \alpha) \to \alpha \to \alpha$ with *nat* itself.

These considerations lead us to a rule where we *substitute into the type*:

$$\frac{\Gamma \vdash e : \forall \alpha.\, \tau \quad \Gamma \vdash \sigma\ type}{\Gamma \vdash e\, [\sigma] : [\sigma/\alpha]\tau}\ \mathsf{tp/tpapp}$$

The second premise is there to check that the type σ which is part of the expression $e\, [\sigma]$ is *valid*. At this point in the course, this just means that all the type variables occurring in σ are declared in Γ (just like all the expression variables in e must be declared in Γ).

Here is a small sample derivation, assuming we have defined

$$\begin{array}{lcl} id & : & \forall \alpha.\, \alpha \to \alpha \\ id & = & \Lambda \alpha.\, \lambda x.\, x \end{array}$$

Then we can typecheck:

$$\dfrac{\dfrac{\dfrac{\vdots}{\cdot \vdash id : \forall \alpha.\, \alpha \to \alpha} \quad \dfrac{\vdots}{\cdot \vdash nat\ type}}{\cdot \vdash id\,[nat] : nat \to nat}\ \textsf{tpapp} \qquad \dfrac{\vdots}{\cdot \vdash \overline{3} : nat}}{\cdot \vdash id\,[nat]\,\overline{3} : nat}\ \textsf{app}$$

where we need some rules to verify that *nat* is a closed type (that is, has no free type variables). Fortunately, that's easy: we just check all the components of a type.

$$\dfrac{\Gamma \vdash \tau_1\ type \quad \Gamma \vdash \tau_2\ type}{\Gamma \vdash \tau_1 \to \tau_2\ type}\ \textsf{tp/arrow} \qquad \dfrac{\alpha\ type \in \Gamma}{\Gamma \vdash \alpha\ type}\ \textsf{tp/tpvar}$$

$$\dfrac{\Gamma, \alpha\ type \vdash \tau\ type}{\Gamma \vdash \forall \alpha.\, \tau\ type}\ \textsf{tp/forall}$$

3 Summary: Typing Rules

For the "official" typing rules it is convenient to assume that λ-abstractions are annotated with the type of the bound variable. In practice, we use a more flexible set of rules where λ-abstractions do not necessarily have to be annotated.

Here is the summary of the language of the polymorphic λ-calculus:

Types	τ	::=	$\alpha \mid \tau_1 \to \tau_2 \mid \forall \alpha.\, \tau$
Expressions	e	::=	$x \mid \lambda x{:}\tau.\, e \mid e_1\, e_2 \mid \Lambda \alpha.\, e \mid e\,[\tau]$
Contexts	Γ	::=	$\cdot \mid \Gamma, x : \tau \mid \Gamma, \alpha\ type$

We assume that all variables and type variables in a context are distinct, and rename bound variable or type variables to maintain this invariant.

In order to avoid any "loopholes" in typing derivations we would like to *presuppose* that the context is well-formed, which comes down to ensuring that all the types occurring in them are valid. We did not discuss this somewhat technical point in lecture, but for completness' sake we provide the rules. The judgment Γ *ctx* means that Γ is a valid (or well-formed) context.

$$\frac{}{(\cdot)\ \mathit{ctx}}\ \text{ctx/emp} \qquad \frac{\Gamma\ \mathit{ctx}}{(\Gamma, \alpha\ \mathit{type})\ \mathit{ctx}}\ \text{ctx/tpvar} \qquad \frac{\Gamma\ \mathit{ctx} \quad \Gamma \vdash \tau\ \mathit{type}}{(\Gamma, x : \tau)\ \mathit{ctx}}\ \text{ctx/var}$$

We now assume that whenever we write $\Gamma \vdash e : \tau$ or $\Gamma \vdash \tau\ \mathit{type}$ then $\Gamma\ \mathit{ctx}$. In type theory we call this a *presupposition* and we are always careful to maintain this presupposition.

$$\frac{\Gamma \vdash \tau_1\ \mathit{type} \quad \Gamma, x_1 : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \lambda x_1{:}\tau_1.\, e_2 : \tau_1 \to \tau_2}\ \text{tp/lam} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \text{tp/var}$$

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

$$\frac{\Gamma, \alpha\ \mathit{type} \vdash e : \tau}{\Gamma \vdash \Lambda\alpha.\, e : \forall\alpha.\, \tau}\ \text{tp/tplam} \qquad \frac{\Gamma \vdash e : \forall\alpha.\, \tau \quad \Gamma \vdash \sigma\ \mathit{type}}{\Gamma \vdash e\,[\sigma] : [\sigma/\alpha]\tau}\ \text{tp/tpapp}$$

We then have the property (called *regularity* in the textbook) that if $\Gamma \vdash e : \tau$ under the presupposition that $\Gamma\ \mathit{ctx}$ then $\Gamma \vdash \tau\ \mathit{type}$. This is easy to verify by rule induction (see Exercise 2).

4 Typing Self-Application Polymorphically

As an exercise in building a typing derivation, we provide a polymorphic type for self-application $\lambda x.\, x\, x$. We accomplish this by allowing x to have a polymorphic types $\forall\alpha.\, \alpha \to \alpha$. We call this type u because there is exactly one normal term of this type: the polymorphic identity function. Applying the identity to itself seems plausible in any case. So we claim:

$$\begin{array}{lcl} u & = & \forall\alpha.\, \alpha \to \alpha \\ \omega & : & u \to u \\ \omega & = & \lambda x.\, x\,[u]\, x \end{array}$$

This is established by the following typing derivation. When you want to build such a derivation yourself, you should always built it "bottom-up", starting with the final conclusion. The fact that the rules are syntax-directed

means you have no choice which rule to choose, but some parts of the type may be unknown and may need to be filled in later.

$$\dfrac{\dfrac{x : u \vdash x\,[u] : \boxed{} \qquad x : u \vdash x : \boxed{}}{x : u \vdash x\,[u]\; x : u}\;\mathsf{tp/app}}{\cdot \vdash \lambda x.\, x\,[u]\; x : u \to u}\;\mathsf{tp/lam}$$

As a rule of thumb, it seems to work best to first fill in the first premise of an application (rule tp/app) and then the second. Continuing in the left branch of the derivation (and remembering that $u = \forall \alpha.\, \alpha \to \alpha$:

$$\dfrac{\dfrac{\dfrac{}{x : u \vdash x : u}\;\mathsf{tp/var} \qquad \dfrac{\begin{matrix}\vdots \\ x : u, \alpha\ \mathit{type} \vdash \alpha \to \alpha\ \mathit{type}\end{matrix}}{x : u \vdash u\ \mathit{type}}\;\mathsf{tp/forall}}{x : u \vdash x\,[u] : {\color{red} u \to u}}\;\mathsf{tp/tpapp} \qquad x : u \vdash x : \boxed{}}{\dfrac{x : u \vdash x\,[u]\; x : u}{\cdot \vdash \lambda x.\, x\,[u]\; x : u \to u}\;\mathsf{tp/lam}}\;\mathsf{tp/app}$$

The type emphasized in red arises as $[u/\alpha](\alpha \to \alpha) = u \to u$. The second premise of the application is immediate by the typing rule for variables and we obtain

$$\dfrac{\dfrac{\dfrac{}{x : u \vdash x : u}\;\mathsf{tp/var} \qquad \dfrac{\begin{matrix}\vdots \\ x : u, \alpha\ \mathit{type} \vdash \alpha \to \alpha\ \mathit{type}\end{matrix}}{x : u \vdash u\ \mathit{type}}\;\mathsf{tp/forall}}{x : u \vdash x\,[u] : u \to u}\;\mathsf{tp/tpapp} \qquad \dfrac{}{x : u \vdash x : u}\;\mathsf{tp/var}}{\dfrac{x : u \vdash x\,[u]\; x : u}{\cdot \vdash \lambda x.\, x\,[u]\; x : u \to u}\;\mathsf{tp/lam}}\;\mathsf{tp/app}$$

The fact that $\alpha \to \alpha$ is a valid type follows quickly by the tp/arrow and tp/tpvar rules. There are more types that work for self-application (see Exercise 3).

Crucial in this example is that we can instantiate the quantifier in $u = \forall \alpha.\, \alpha \to \alpha$ with u itself. This "self-referential" nature of the type quantifier is called *impredicativity* because it quantifies not only over types already defined, but also itself. Some systems of type theory reject impredicative quantification because the meaning of the quantified type is not constructed from the meaning of types we previously understand. Impredicativity was

also seen as a source of paradoxes, although Girard did give a syntactic argument for the consistency of System F [Gir71] with impredicative quantification.

5 Church Numerals Revisited

We can now revisit the representation of Church numerals and express them and functions on them in the polymorphic λ-calculus. We present the definitions in the language LAMBDA, which uses polymorphic types when files have extension `.poly` or the command line argument `-l poly`. We use `!a` as concrete syntax for $\forall\alpha$, and `/\a` for $\Lambda\alpha$. Type definitions are preceded by the keyword `type`, and type declarations for variable definitions are preceded by the keyword `decl`.

```
type nat = !a. (a -> a) -> a -> a

decl zero : nat
decl succ : nat -> nat

defn zero = /\a. \s. \z. z
defn succ = \n. /\a. \s. \z. s (n [a] s z)

decl plus : nat -> nat -> nat
defn plus = \n. \k. n [nat] succ k

decl times : nat -> nat -> nat
defn times = \n. \k. n [nat] (plus k) zero

norm _0 = zero
norm _1 = succ _0
norm _2 = succ _1
norm _3 = succ _2

norm _6 = times _2 _3
```

Listing 1: Polymorphic natural numbers in LAMBDA

So far, this straightforwardly follows the structure of the motivating examples. In order to represent the predecessor function, we require pairs of natural numbers. But what are their types? Recall:

$$\mathit{pair} = \lambda x.\, \lambda y.\, \lambda k.\, k\, x\, y$$

from which conjecture something like

$$pair : nat \to nat \to (nat \to nat \to \tau) \to \tau$$

where τ is arbitrary. So we realize that this function is *polymorphic* and we abstract over the result type of the continuation. We call the type of pairs of natural numbers *nat2*. In the type of the *pair* function it is then convenient to place the type abstraction *after* the two natural numbers have been received.

$$\begin{array}{lcl} nat2 & = & \forall\alpha.\,(nat \to nat \to \alpha) \to \alpha \\ \\ pair & : & nat \to nat \to nat2 \\ pair & = & \lambda x.\,\lambda y.\,\Lambda\alpha.\,\lambda k.\,k\,x\,y \end{array}$$

Now we can define the *pred2*, with the specification that $pred2\;\overline{n} = pair\;\overline{n}\;\overline{n \dot{-} 1}$. We leave open the two places we have to provide a type.

$$\begin{array}{lcl} pred2 & : & nat \to nat \to nat \\ pred2 & = & \lambda n.\,n\,[\boxed{}]\,(\lambda p.\,p\,[\boxed{}]\,(\lambda x.\,\lambda y.\,pair\,(succ\;x)\;x))\,(pair\;zero\;zero) \end{array}$$

In the first box, we need to fill in the result type of the iteration (which is the type argument to $\overline{n}$), and this is *nat2*. In the second box we need to fill in the result type for the decomposition into a pair, and that is also *nat2*. Then, for the final definition of *pred* we only extract the second component of the pair, so the continuation only returns a natural number rather than a pair.

$$\begin{array}{lcl} pred & : & nat \to nat \\ pred & = & \lambda n.\,n\,[nat]\,(\lambda x.\,\lambda y.\,y) \end{array}$$

Below is a summary of this code in LAMBDA.

```
1  type nat2 = !a. (nat -> nat -> a) -> a
2
3  decl pair : nat -> nat -> nat2
4  defn pair = \x. \y. /\a. \k. k x y
5
6  decl pred2 : nat -> nat2
7  defn pred2 = \n. n [nat2] (\p. p [nat2] (\x. \y. pair (succ x) x))
8                            (pair zero zero)
9
10 decl pred : nat -> nat
11 defn pred = \n. pred2 n [nat] (\x. \y. y)
12
13 norm _6_5 = pred2 _6
14 norm _5 = pred _6
```

Listing 2: Predecessor on natural numbers in LAMBDA

6 Theory

We did not discuss this in lectures, but of course we should expect the properties for the simply-typed λ-calculus to carry over, once suitable reduction rules have been defined. We will talk about these at the beginning of the next lecture.

One remarkable fact about the polymorphic λ-calculus (which is quite difficult to prove) is that every expression still has a normal form.

Exercises

Exercise 1

(i) Find a definition of *plus* : *nat* → *nat* → *nat* that works in the simply-typed λ-calculus in the sense that we need to instantiate the type $\forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$ only with a type variable.

(ii) Give a simply-typed definition (in the sense of part (i)) for *times* or conjecture that none exists.

Exercise 2 Prove that if $\Gamma \vdash e : \tau$ under the presupposition that Γ *ctx* then $\Gamma \vdash \tau$ *type*.

Exercise 3 We write F for a (mathematical) function from types to types (which is not expressible in the polymorphic λ-calculus but requires system F^ω). A more general family of types (one for each F) for self-application is given by

$$\begin{array}{rcl} w_F & = & \forall\alpha.\,\alpha \to F(\alpha) \\ \omega_F & : & w_F \to F(w_F) \\ \omega_F & = & \lambda x.\, x\,[w_F]\,x \end{array}$$

We recover the type from this lecture with $F = \Lambda\alpha.\,\alpha$. You may want to verify the general typing derivation in preparation for the following questions, but you do not need to show it.

(i) Consider $F = \Lambda\alpha.\,\alpha \to \alpha$. In this case $w_F = \mathit{bool}$. Calculate the type and characterize the behavior of ω_F as a function on Booleans.

(ii) Consider $F = \Lambda\alpha.\,(\alpha \to \alpha) \to \alpha$. Calculate w_F, the type of ω_F, and characterize the the behavior of ω_F. Can you relate w_F and ω_F to the types and functions we have considered in the course so far?

References

[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.

[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.

Lecture Notes on
From λ-Calculus to Programming Languages

15-814: Types and Programming Languages
Frank Pfenning

Lecture 7
Tuesday, September 22, 2020

1 Introduction

First, we will briefly talk about the dynamic of polymorphism (which abstracts over types and applies functions to types), and then exercise polymorphism a little to generalizing iteration from natural numbers to richer types, using trees as an example.

Then we take the a big step from a pure λ-calculus to real programming languages by changing our attitude on data: we would like to represent them directly instead of indirectly as functions, for several reasons explained in **??**.

2 Dynamics of Polymorphism

We already gave the typing rules for parametric polymorphism in the previous lecture, but we did not yet update the rules for computation or normal and neutral terms. A key observation is that the structure of the types in our little language is such that we should be able to just add new rules without touching the old ones in any way. This form of modularity also carries over to the proofs of the key properties we would like the system to have: they decompose into cases along the lines of the type constructs we have.

First, reduction:

$$\frac{}{(\Lambda\alpha.\,e)\,[\tau] \longrightarrow [\tau/\alpha]e}\ \textsf{red/tpbeta}$$

$$\frac{e \longrightarrow e'}{\Lambda\alpha.\,e \longrightarrow \Lambda\alpha.\,e'}\ \textsf{red/tplam} \qquad \frac{e \longrightarrow e'}{e\,[\tau] \longrightarrow e'\,[\tau]}\ \textsf{red/tpapp}_1$$

There is no $\textsf{red/tpapp}_2$ rule since we do not reduce types themselves.

In this definition we use substitution $[\tau/\alpha]e$, which is defined in the expected way, possibly renaming type variables bound by $\Lambda\beta.\,\sigma$ or $\forall\beta.\,sigma$ that may occur in e so as to avoid capturing any type variables free in τ.

There are also two new rules for normal and neutral terms, retaining all the others.

$$\frac{e\ \mathit{normal}}{\Lambda\alpha.\,e\ \mathit{normal}}\ \textsf{norm/lam} \qquad \frac{e\ \mathit{neutral}}{e\,[\tau]\ \mathit{neutral}}\ \textsf{neut/app}$$

The key theorems are *preservation* and *progress*, establishing a connection between types, reduction, and normal forms.

Preservation. If $\Gamma \vdash e : \tau$ and $e \longrightarrow e'$ then $\Gamma \vdash e' : \tau$

Progress. If $\Gamma \vdash e : \tau$ then either $e \longrightarrow e'$ for some e' or e *normal*.

Finality of Normal Forms. There is no $\Gamma \vdash e : \tau$ such that $e \longrightarrow e'$ for some e' and e *normal*.

3 Generalizing Iteration

It may be helpful to think of iteration on natural numbers to arise from they way they are constructed

$$\begin{array}{lcl} \textsf{zero} & : & \mathit{nat} \\ \textsf{succ} & : & \mathit{nat} \to \mathit{nat} \end{array}$$

Namely, if we imaging a term

$$\textsf{succ}\,(\textsf{succ}\ \ldots\ (\textsf{succ}\,\textsf{zero})) : \mathit{nat}$$

then we *replace* the constructor by appropriate functions and constants (using g for succ and c for zero

$$g\,(g\ \ldots, (g\,c))$$

Now we should work out the types of g and c. Clearly, $g : \tau \to \tau$ for any type τ and $c : \tau$. We can obtain these types from the type of zero and succ by replacing *nat* with α. So, if we want to see $n : \mathit{nat}$ as an *iterator* then

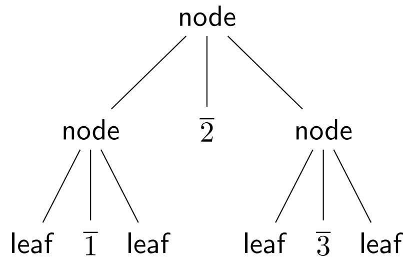
$$\mathit{nat} = \forall\alpha.\,(\alpha \to \alpha) \to \alpha \to \alpha$$

where the first argument is the result type τ following by a function $g : \tau \to \tau$ and a constant $c : \tau$.

Let's follow the same recipe for trees of natural numbers. They are generated from

$$\begin{array}{lcl} \mathsf{node} & : & \mathit{tree} \to \mathit{nat} \to \mathit{tree} \to \mathit{tree} \\ \mathsf{leaf} & : & \mathit{tree} \end{array}$$

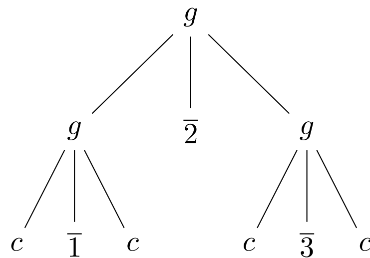
In this representation, leaves carry no information and every interior node has a left subtree, a natural number, and right subtree. For example, the tree



would be constructed with

$$\mathsf{node}\ (\mathsf{node}\ \mathsf{leaf}\ \overline{1}\ \mathsf{leaf})\ \overline{2}\ (\mathsf{node}\ \mathsf{leaf}\ \overline{3}\ \mathsf{leaf})$$

To see the form of an iterator we replace the constructors node and leaf with functions g and a constant c, respectively, which would give use the tree



This time, we see that we should have

$$\begin{array}{lcl} g & : & \tau \to \mathit{nat} \to \tau \to \tau \\ c & : & \tau \end{array}$$

for an arbitrary type τ. Once again, this was obtained from replacing the type *tree* in the types of node and leaf with an arbitrary type. We can express this as a polymorphic type as:

$$\mathit{tree} = \forall \alpha.\, (\alpha \to \mathit{nat} \to \alpha \to \alpha) \to \alpha \to \alpha$$

As an example, to sum up the elements of the tree we would define

$$\begin{array}{lcl} \mathit{sum} & : & \mathit{tree} \to \mathit{nat} \\ \mathit{sum} & = & \lambda t.\, t\ [\mathit{nat}]\ (\lambda x.\, \lambda n.\, \lambda y.\, \mathit{plus}\ x\ (\mathit{plus}\ n\ y))\ \mathit{zero} \end{array}$$

First, we pass to t the result type *nat*, then a function g expecting the sum of the left subtree as x, then n as the value stored in the node, and then the sum of the right subtree as y. The function g then just has to add these three numbers to obtain the sum of a tree. Since the leaf does not contain any number, its value is 0 (the neutral element of addition).

The definition of the tree constructors themselves follow the structure of the type. The easy case first:

$$\begin{array}{lcl} \mathsf{leaf} & : & \mathit{tree} \\ \mathsf{leaf} & = & \Lambda \alpha.\, \lambda n.\, \lambda l.\, l \end{array}$$

For the node constructor we have the parameter n at the head of the term, and we just have to remember to match the types by applying the representations of the left and right subtrees to all parameters (including the type α).

$$\begin{array}{lcl} \mathsf{node} & : & \mathit{tree} \to \mathit{nat} \to \mathit{tree} \to \mathit{nat} \\ \mathsf{node} & = & \lambda t_1.\, \lambda x.\, \lambda t_2.\, n\ (t_1\ [\alpha]\ n\ l)\ x\ (t_2\ [\alpha]\ n\ l) \end{array}$$

We did not live-code this in lecture, but below is the code for trees in LAMBDA, which should come after the code for natural number from the last lecture. You can find this code online at tree.poly.

```
1 type tree = !a. (a -> nat -> a -> a) -> a -> a
2
3 decl leaf : tree
4 decl node : tree -> nat -> tree -> tree
5
6 defn leaf = /\a. \n. \l. l
```

```
7  defn node = \t1. \x. \t2. /\a. \n. \l. n (t1 [a] n l) x (t2 [a] n l)
8
9  decl sum : tree -> nat
10 defn sum = \t. t [nat] (\s1. \x. \s2. plus s1 (plus x s2)) zero
11
12 norm t123 = node (node leaf _1 leaf) _2 (node leaf _3 leaf)
13 norm s6 = sum t123
14 conv s6 = _6
```

Listing 1: Trees of natural numbers in LAMBDA

4 Evaluation versus Reduction

The λ-calculus is exceedingly elegant and minimal, a study of functions in the purest possible form. We find versions of it in most, if not all modern programming languages because the abstractions provided by functions are a central structuring mechanism for software. On the other hand, there are some problem with the functions-as-data representation technique of which we have seen Booleans, natural numbers, and trees. Here are a few notes:

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 or even the polymorphic λ-calculus where all well-typed expressions have a normal form. Types, however, are needed to understand and classify data representations and the functions defined over them. Fortunately, this can be fixed by introducing *recursive types*, so this is not a deeper obstacle to representing data as functions.

Expressiveness. While all *computable functions* on the natural numbers can be represented in the sense of correctly modeling their input/output behavior, some natural *algorithms* are difficult or impossible to express. For example, under some reasonable assumptions the minimum function on numbers n and k has complexity $O(\max(n, k))$ **[?]**, which is surprisingly slow, and our predecessor function took $O(n)$ steps. Other representations are possible, but they either complicate typing or inflate the size of the representations.

Observability of functions. Since reduction results in normal form, to interpret the outcome 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.

In the remainder of this lecture we focus on the last point: 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.

5 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 *value*. 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. We'll call this property *sequentiality*, since it means execution is sequential rather than parallel or concurrent. Furthermore, since we do not reduce underneath λ-abstractions, we only evaluate expressions that are *closed*, that is, have *no free variables*.

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 $e \mapsto e'$ for some e' or e *value*.

Finality of Values. There is no $\cdot \vdash e : \tau$ such that $e \mapsto e'$ for some e' and e *value*.

Sequentiality. 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.

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

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

$$\frac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2}\ \mathsf{step/app_1} \qquad \frac{}{(\lambda x.\, e_1)\, e_2 \mapsto [e_2/x]e_1}\ \mathsf{beta}$$

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.

$$\frac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2}\ \mathsf{step/app_1} \qquad \frac{e_1\ \mathit{value} \quad e_2 \mapsto e_2'}{e_1\, e_2 \mapsto e_1\, e_2'}\ \mathsf{step/app_2}$$

$$\frac{e_2\ \mathit{value}}{(\lambda x.\, e_1)\, e_2 \mapsto [e_2/x]e_1}\ \mathsf{step/beta/val}$$

We achieve sequentiality 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.

There are a lot of spurious arguments about whether a language should support call-by-value or call-by-name. This turns out to be a false dichotomy and only historically in opposition.

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

6 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:

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

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.

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

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

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

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.

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

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.

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

Exercises

Exercise 1 Show the *new cases* in the proof of preservation and progress arising from parametric polymorphism.

(i) (Preservation) If $\Gamma \vdash e : \tau$ and $e \longrightarrow e'$ then $\Gamma \vdash e' : \tau$

(ii) (Progress) If $\Gamma \vdash e : \tau$ then either $e \longrightarrow e'$ for some e' or e *normal*

(iii) (Finality of Normal Forms) There is no $\Gamma \vdash e : \tau$ such that $e \longrightarrow e'$ for some e' and e *normal*.

Explicitly state any additional substitution properties you need (in addition to Theorem L5.6), but you do not need to prove them.

Exercise 2 An alternative form of binary tree given in **??** is one where all information is stored in the leaves and none in the nodes. Let's call such a tree a *shrub*.

(i) Give the types for shrub constructors.

(ii) Give the construction of a shrub containing the numbers 1, 2, and 3.

(iii) Give the polymorphic definition of the type *shrub*, assuming it is represented by its own iterator.

(iv) Write a function $sumup$ to sum the elements of a shrub.

(v) Write a function $mirror$ that returns the mirror image of a given tree, reflected about a vertical line down from the root.

Exercise 3 We say two types τ and σ are *isomorphic* (written $\tau \cong \sigma$) if there are two functions $\mathit{forth} : \tau \to \sigma$ and $\mathit{back} : \sigma \to \tau$ such that they compose to the identity in both directions, that is, $\lambda x.\, \mathit{back}\ (\mathit{forth}\ x))$ is equal to $\lambda x.\, x$ and $\lambda y.\, \mathit{forth}\ (\mathit{back}\ y)$ is equal to $\lambda y.\, y$.

Consider the two types

$$\begin{array}{rcl} \mathit{nat} & = & \forall \alpha.\, (\alpha \to \alpha) \to \alpha \to \alpha \\ \mathit{tan} & = & \forall \alpha.\, \alpha \to (\alpha \to \alpha) \to \alpha \end{array}$$

(i) Provide functions $\mathit{forth} : \mathit{nat} \to \mathit{tan}$ and $\mathit{back} : \mathit{tan} \to \mathit{nat}$ that, intuitively, should witness the isomorphism between *nat* and *tan*.

(ii) Compute the normal forms of the two function compositions. You may recruit the help of the LAMBDA implementation for this purpose.

(iii) Are the two function compositions β-equal to the identity? If yes, you are done. If not, can you see a sense under which they would be considered equal, either by changing your two functions or be defining a suitably justified notion of equality?

Exercise 4 Prove sequentiality: If $\cdot \vdash e : \tau$, $e \mapsto e_1$ and $e \mapsto e_2$ then $e_1 = e_2$.

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 Progress

15-814: Types and Programming Languages
Frank Pfenning

Lecture 8
Thursday, September 24, 2020

1 Introduction

We start by short exploration of the consequences of making the structure of functions opaque and then focus on proving *progress*, one of the key properties connecting typing and evaluation. This in turn requires the *canonical forms theorem*, which is a new form of representation theorem (such as we have proved for Booleans, represented in the typed λ-calculus).

Let's reiterate the critical properties we care about for now:

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

Progress. For every expression $\cdot \vdash e : \tau$ either $e \mapsto e'$ for some e' or e *value*.

Finality of Values. There is no $\cdot \vdash e : \tau$ such that $e \mapsto e'$ for some e' and e *value*.

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

Since we already proved preservation for ordinary reduction in some detail for the simply-typed λ-calculus, in this lecture we focus on the progress theorem so we can understand the structure of its proof.

2 Observing Functional Values

As we have emphasized, we assume we cannot directly observe the structure of functions when they are outcome of computation. Instead, we can probe

such functions by applying them to argument and observing the results. As an example, consider our language with parametric polymorphism and Booleans, and our usual representation of natural numbers as their iterators:

$$\mathit{nat} : \forall\alpha.\, (\alpha \to \alpha) \to \alpha \to \alpha$$

If we have an expression $\cdot \vdash e : \mathit{nat}$ such that e *value* we know it will have the form $\Lambda\alpha.\, e'$ for some e', but we cannot observe e'. Moreover, e' may not even be a value, even though e is. Nevertheless, we can test, for example, if the value e is zero or positive. Consider

$$\cdot \vdash e\ [\mathsf{bool}] : (\mathsf{bool} \to \mathsf{bool}) \to \mathsf{bool} \to \mathsf{bool}$$

and

$$\cdot \vdash e\ [\mathsf{bool}]\ (\lambda b.\, \mathsf{false})\ \mathsf{true} : \mathsf{bool}$$

If this expression evaluates to true then e "represents" zero, and if it evaluates to false then e "represents" some positive number. We put "represents" in quotes here because, for example, e may not be equal to $\Lambda\alpha.\, \lambda s.\, \lambda z.\, z$. Instead, it *behaves* like this function when applied to a type τ, and two arguments of type $\tau \to \tau$ and τ in this order. We just have to keep in mind that this computation takes place when we observe e, and not when e is originally evaluated.

A small item of notation: we write $e \hookrightarrow v$ to express that e *evaluates* to the value v. This presupposes that $\cdot \vdash e : \tau$ for some τ and ensures that v *value*. Formally, it is defined by

$$\frac{v\ \mathit{value}}{v \hookrightarrow v}\ \mathsf{eval/val} \qquad \frac{e \mapsto e' \quad e' \hookrightarrow v}{e \hookrightarrow v}\ \mathsf{eval/step}$$

It is also possible to define evaluation directly as a so-called *big-step evaluation* judgment as compared to the *small-step evaluation* we have defined so far (see Exercise 1).

From now on we will often write v for an expression we know to be a value, but at least for the moment we will not automatically imply this from the notation, that is, we will still write v *value* where we are not already assured that v is indeed a value.

3 Progress

The progress property is intended to rule out intuitively meaningless expressions that neither reduce nor constitute a value. For example, the ill-typed expression $\mathsf{if}\ (\lambda x.\, x)\ \mathsf{false}\ \mathsf{true}$ cannot take a step since the subject $(\lambda x.\, x)$ is a value but the whole expression is not a value and cannot take a step. Similarly, the expression $\mathsf{if}\ b\ \mathsf{false}\ \mathsf{true}$ is well-typed in the context with $b : \mathsf{bool}$, but it cannot take a step nor is it a value. Therefore, it is clear that the assumptions that e is closed that that e has a valid type are both needed for this theorem. It may be helpful to refer to the summary of the judgments inference rules while reading this proof.

Theorem 1 (Progress)
If $\cdot \vdash e : \tau$ then either $e \mapsto e'$ for some e' or e value.

Proof: There are not many candidates for the structure of 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:

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

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

$\lambda x_1.\, e_2$ *value* 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:

$$\frac{x : \tau \in (\cdot)}{\cdot \vdash x : \tau}$$

This case is impossible because there is not declaration for x in the empty context.

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 redex. In more detail:

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

$e_1 \mapsto e_1'$ — Subcase
$e = e_1\, e_2 \mapsto e_1'\, e_2$ — by rule step/app$_1$

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

$e_2 \mapsto e_2'$ — Sub^{2}case
$e_1\, e_2 \mapsto e_1\, e_2'$ — By rule step/app$_2$ since e_1 *value*

e_2 *value* — 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 *value*. 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 theorem* which we state after this proof. Finishing this sub^{2}case:

$e = (\lambda x.\, e_1')\, e_2 \mapsto [e_2/x]e_1'$ — By rule step/app/lam since e_2 *value*

Case:

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

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

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 $e = \mathsf{if}\ e_1\ e_2\ e_3$.

Either $e_1 \mapsto e_1'$ for some e_1' or e_1 *value* — 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 step/if

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

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

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

Cases: For rules tp/tplam and tp/tpapp see Exercise 2.

□

This completes the proof. The complex inversion steps can be summarized in the *canonical forms theorem* 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.

Theorem 2 (Canonical Forms)

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

(ii) If $\cdot \vdash v : \forall \alpha.\, \tau$ then $v = \Lambda \alpha.\, e$.

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

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

4 Type Preservation*

This proof was not done in lecture, but is presented here for completeness. In a future lecture we will reexamine the proof of this theorem.

We already know that the rules should satisfy the substitution property (Theorem L5.6). We can easily check the new cases in the proof because substitution remains compositional. For example, $[e'/x](\mathsf{if}\ e_1\ e_2\ e_3) = \mathsf{if}\ ([e'/x]e_1)\ ([e'/x]e_2)\ ([e'/x]e_3)$. However, some new properties are needed for parametric polymorphism, so we make them explicit here and generalize the previous theorem.

Theorem 3 (Substitution Property)

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

(ii) *If* $\Gamma \vdash \tau$ *type and* $(\Gamma, \alpha\ \mathit{type}, \Gamma')$ *ctx then* $(\Gamma, [\tau/\alpha]\Gamma')$ *ctx*.

(iii) *If* $\Gamma \vdash \tau$ *type and* $\Gamma, \alpha\ \mathit{type}, \Gamma' \vdash \sigma$ *type then* $\Gamma, [\tau/\alpha]\Gamma' \vdash [\tau/\alpha]\sigma$ *type*.

(iv) *If* $\Gamma \vdash \tau$ *type and* $\Gamma, \alpha\ \mathit{type}, \Gamma' : \tau \vdash e : \sigma$ *then* $\Gamma, [\tau/\alpha]\Gamma' \vdash [\tau/\alpha]e : [\tau/\alpha]\sigma$.

Proof: Each part by rule induction on the second given derivation. We have to exploit the fact that term variables x do not occur in types, and we need to remember our presuppositions and (silent) renaming of bound variables (both for terms and types). □

On to preservation.

Theorem 4 (Type Preservation)
If $\cdot \vdash e : \tau$ *and* $e \mapsto e'$ *then* $\cdot \vdash e' : \tau$.

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

In each case we apply inversion on the typing derivation to obtain typing derivations for the components of e. From these derivations we assemble a typing derivation for e'. In cases of a step involving substitution, we have to appeal to the substitution property to obtain the resulting derivation.

Case:

$$\frac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2}\ \mathsf{step/app}_1$$

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:

$$\frac{v_1\ \mathit{value} \quad e_2 \mapsto e_2'}{v_1\, e_2 \mapsto v_1\, e_2'}\ \mathsf{step/app_2}$$

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:

$$\frac{v_2\ \mathit{value}}{(\lambda x.\, e_1)\, v_2 \mapsto [v_2/x]e_1}\ \mathsf{step/app/lam}$$

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 3).

$\cdot \vdash (\lambda x.\, e_1)\, v_2 : \tau$	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 3)

Case:

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

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 tp/if

Case:

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

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 step/if/false is analogous to the previous case.

Case:

$$\frac{e_1 \mapsto e_1'}{e_1\ [\sigma] \mapsto e_1'\ [\sigma]}\ \mathsf{step/tpapp}$$

where $e = e_1\ [\sigma]$ and $e' = e_1'\ [\sigma]$.

$\cdot \vdash e_1\ [\sigma] : \tau$	Assumption
$\cdot \vdash e_1 : \forall\alpha.\,\tau_2$ where $\tau = [\sigma/\alpha]\tau_2$	By inversion
$\cdot \vdash e_1' : \forall\alpha.\,\tau_2$	By ind. hyp
$\cdot \vdash e_1'\ [\sigma] : [\sigma/\alpha]\tau_2$	By rule tp/tpapp
$\cdot \vdash e' : \tau$	Since $e' = e_1'\ [\sigma]$ and $\tau = [\sigma/\alpha]\tau_2$

Case:

$$\frac{}{(\Lambda\alpha.\,e_2)\ [\sigma] \mapsto [\sigma/\alpha]e_2}\ \mathsf{step/tpapp/tplam}$$

where $e = (\Lambda\alpha.\,e_2)\ [\sigma]$ and $e' = [\sigma/\alpha]e_2$.

$\cdot \vdash (\Lambda\alpha.\,e_2)\ [\sigma] : \tau$	Assumption
$\cdot \vdash (\Lambda\alpha.\,e_2) : \forall\alpha.\,\tau_2$ and $\cdot \vdash \sigma$ *type*	
with $\tau = [\sigma/\alpha]\tau_2$ for some τ_2	By inversion
α *type* $\vdash e_2 : \tau_2$	By inversion
$\cdot \vdash [\sigma/\alpha]e_2 : [\sigma/\alpha]\tau_2$	By the substitution property (Theorem 3)

□

5 Pairs

Types capture fundamental programming abstractions. If a type system and its underlying programming language is well-designed, we can then build complex data representations and computational mechanisms from a few primitives. The most fundamental is that of a function, captured in the type $\tau_1 \to \tau_2$. As a next step we look for ways to *aggregate* data. The simplest is *pairs*, which are captured by the type $\tau_1 \times \tau_2$. By iterating pairs we can then assemble tuples with elements of arbitrary types.

5.1 Constructing Pairs

Fundamentally, for each new type we introduce we must be able to *construct* element of the type. For example, $\lambda x.\, e$ constructs element of the function type $\tau_1 \to \tau_2$. To construct new elements of the type $\tau_1 \times \tau_2$ we use the almost universal notation $\langle e_1, e_2 \rangle$. The typing rule is straightforward

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}\ \mathsf{tp/pair}$$

This is the only rule for pairs, so we maintain the property that the rules are syntax-directed.

Next we should consider the *dynamics*, that is, which are the new values of type $\tau_1 \times \tau_2$ and how do we evaluate pairs. In this lecture we consider *eager pairs*, that is, a pair is only a value if both components are. *Lazy pairs* are the subject of Exercise 6.

$$\frac{e_1\ \mathit{value} \quad e_2\ \mathit{value}}{\langle e_1, e_2 \rangle\ \mathit{value}}\ \mathsf{val/pair}$$

We then assume that we can *observe* the components of a pair. So, at the current extent of our language we can observe the Booleans and, inductively, pairs of observable type.

Types	τ	$::=$	$\alpha \mid \tau_1 \to \tau_2 \mid \forall \alpha.\, \tau \mid \mathsf{bool} \mid \tau_1 \times \tau_2$
Observable Types	o	$::=$	$\mathsf{bool} \mid o_1 \times o_2$

To *evaluate* a pair we decided on evaluating from left to right: it preserves sequentiality and is consistent with other constructs like function applications that are also evaluated from left to right.

$$\frac{e_1 \mapsto e_1'}{\langle e_1, e_2 \rangle \mapsto \langle e_1', e_2 \rangle}\ \mathsf{step/pair}_1 \qquad \frac{v_1\ \mathit{value} \quad e_2 \mapsto e_2'}{\langle v_1, e_2 \rangle \mapsto \langle v_1, e_2' \rangle}\ \mathsf{step/pair}_2$$

In writing this rule we are starting a convention where expressions known to be values are denoted by v instead of e.

5.2 Destructing Pairs

Constructing pairs is only one side of the coin. We also need to be able to access the components of a pair. There seem to be two natural choices: (1) to have a first and second projection function, and (2) decompose a pair with a *letpair*-like construct (from the pure λ-calculus) that gives access to both components. It turns out, projections as a primitive are more suitable for lazy pairs, while a *letpair* construct matches eager pairs. We formulate it here as a *case* expression, because it turns out that several other destructors can also be written in this way, leading to a more uniform language.

$$\mathsf{case}\ e\ (\langle x_1, x_2\rangle \Rightarrow e')$$

The crucial operational rule just deconstructs a pair of values.

$$\frac{v_1\ \mathit{value} \quad v_2\ \mathit{value}}{\mathsf{case}\ \langle v_1, v_2\rangle\ (\langle x_1, x_2\rangle \Rightarrow e_3) \mapsto [v_1/x_2][v_2/x_2]e_3}\ \mathsf{step/casep/pair}$$

We also need a second rule to reduce the subject of the case-expression until it becomes a value.

$$\frac{e_0 \mapsto e_0'}{\mathsf{case}\ e_0\ (\langle x_1, x_2\rangle \Rightarrow e_3) \mapsto \mathsf{case}\ e_0'\ (\langle x_1, x_2\rangle \Rightarrow e_3)}\ \mathsf{step/casep}_0$$

In the typing rule, we know the subject of the case-expression should be a pair and the body should be the same type as the whole expression.

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2 \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e' : \tau'}{\Gamma \vdash \mathsf{case}\ e\ (\langle x_1, x_2\rangle \Rightarrow e') : \tau'}\ \mathsf{tp/casep}$$

Note how x_1 and x_2 are added to the context in the second premise because they may appear in e'.

We are of course obligated to check that our language properties are preserved under this extension, which we will do shortly. Meanwhile, let's write two small programs, verifying that the projections can indeed be defined.

$$\begin{array}{lcl} \mathit{fst} & : & \forall\alpha.\,\forall\beta.\,(\alpha \times \beta) \to \alpha \\ \mathit{fst} & = & \Lambda\alpha.\,\Lambda\beta.\,\lambda p.\,\mathsf{case}\ p\ (\langle x, y\rangle \Rightarrow x) \\ \mathit{snd} & : & \forall\alpha.\,\forall\beta.\,(\alpha \times \beta) \to \beta \\ \mathit{snd} & = & \Lambda\alpha.\,\Lambda\beta.\,\lambda p.\,\mathsf{case}\ p\ (\langle x, y\rangle \Rightarrow y) \end{array}$$

6 Preservation and Progress, Revisited*

This section was also not covered in lecture, but given here for completeness.

Design of the new types and expressions are always carefully rigged so that the preservation and progress theorems continue to hold. Among other things, we make sure that each definition is self-contained. For example, we might have postulated a *primitive function* $\mathsf{pair} : \tau_1 \to (\tau_2 \to (\tau_1 \times \tau_2))$ but then the canonical forms theorem would have to be altered: not every value of function type is actually a λ-expression. Instead, we have a new *expression constructor* $\langle -, - \rangle$ and we can *define pair* as a regular function from that.

Theorem 5 (Type Preservation, new cases for $\tau_1 \times \tau_2$)
If $\cdot \vdash e : \tau$ *and* $e \mapsto e'$ *then* $\cdot \vdash e' : \tau$

Proof: Recall the structure of the proof of type preservation. We use rule induction on the derivation of $e \mapsto e'$ and apply inversion on $\cdot \vdash e : \tau$ in order to gain enough information to assemble a derivation of e'. We exploit here that the typing rules are syntax-directed. Technically, we rely on the substitution property and so that needs to be extended as well. But since we continue to use a standard hypothetical judgment and we do not touch our notion of variable, the new cases don't require any particular attention.

The congruence cases of reduction, where we reduce a subexpression, are straightforward because we can follow this pattern mechanically. For example:

Case:

$$\frac{e_1 \mapsto e_1'}{\langle e_1, e_2 \rangle \mapsto \langle e_1', e_2 \rangle}\ \mathsf{step/pair}_1$$

where $e = \langle e_1, e_2 \rangle$, $e' = \langle e_1', e_2 \rangle$.

$\cdot \vdash \langle e_1, e_2 \rangle : \tau$	Assumption
$\cdot \vdash e_1 : \tau_1$ and $\cdot \vdash e_2 : \tau_2$ where $\tau = \tau_1 \times \tau_2$.	By inversion
$\cdot \vdash e_1' : \tau_1$	By ind. hyp.
$\cdot \vdash \langle e_1', e_2 \rangle : \tau_1 \times \tau_2$	By rule tp/pair

The main case to check then is one where some "real" reduction takes place. This is when a destructor for values of a type meets a constructor.

Case:

$$\frac{v_1 \ \mathit{value} \quad v_2 \ \mathit{value}}{\mathsf{case}\ \langle v_1, v_2\rangle\ (\langle x_1, x_2\rangle \Rightarrow e_3) \mapsto [v_1/x_1][v_2/x_2]e_3}\ \mathsf{step/casep/pair}$$

where $e = \mathsf{case}\ \langle v_1, v_2\rangle\ (\langle x_1, x_2\rangle \Rightarrow e_3)$ and $e' = [v_1/x_2][v_2/x_2]e_3$. In this case, we cannot apply the induction hypothesis (the premises are of a different form), but we can nevertheless apply inversion and then use the substitution property.

$\cdot \vdash \mathsf{case}\ \langle v_1, v_2\rangle\ (\langle x_1, x_2\rangle \Rightarrow e_3) : \tau$ — Assumption
$\cdot \vdash \langle v_1, v_2\rangle : \tau_1 \times \tau_2$
and $x_1 : \tau_1, x_2 : \tau_2 \vdash e_3 : \tau$ for some τ_1 and τ_2 — By inversion
$\cdot \vdash v_1 : \tau_1$ and $\cdot \vdash v_2 : \tau_2$ — By inversion
$x_1 : \tau_1 \vdash [v_2/x_2]e_3 : \tau$ — By substitution (Theorem 3)
$\cdot \vdash [v_1/x_1][v_2/x_2]e_3 : \tau$ — By substitution (Theorem 3)

□

In preparation for the progress theorem, we extend the canonical forms theorem. The latter is a bit different than the other theorems in that for every extension of our language by a new form of type, we need to add a case that characterizes the values of the new type.

Theorem 6 (Canonical Forms)
Assume v value. Then

(i) If $\cdot \vdash v : \tau_1 \to \tau_2$ then $v = \lambda x.\, e'$ for some x and e'.

(ii) If $\cdot \vdash v : \forall \alpha.\, \tau$ then $v = \Lambda \alpha.\, e$.

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

(iv) If $\cdot \vdash v : \tau_1 \times \tau_2$ then $v = \langle v_1, v_2\rangle$ for some v_1 value and v_2 value.

Proof: We consider each case for v *value* and then invert on the typing derivation in each case. □

Theorem 7 (Progress, new cases for $\tau_1 \times \tau_2$)
If $\cdot \vdash e : \tau$ then either $e \mapsto e'$ for some e' or e value.

Proof: By rule induction on $\cdot \vdash e : \tau$. The rules where we reduce pairs are straightforward, as before, so we only write out the case construct.

Case:

$$\frac{\cdot \vdash e_0 : \tau_1 \times \tau_2 \quad x_1 : \tau_1, x_2 : \tau_2 \vdash e_2 : \tau}{\cdot \vdash \mathsf{case}\ e_0\ (\langle x_1, x_2 \rangle \Rightarrow e_3) : \tau}\ \mathsf{tp/casep}$$

where $e = \mathsf{case}\ e_0\ (\langle x_1, x_2 \rangle \Rightarrow e_3)$.

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

$e_0 \mapsto e_0'$ — First subcase
$\mathsf{case}\ e_0\ (\langle x_1, x_2 \rangle \Rightarrow e_3) \mapsto \mathsf{case}\ e_0'\ (\langle x_1, x_2 \rangle \Rightarrow e_3)$ — By rule $\mathsf{step/casep}_0$

e_0 *value* — Second subcase
$e_0 = \langle v_1, v_2 \rangle$ for some v_1 *value* and v_2 *value*
By the canonical forms (Theorem 6)
$\mathsf{case}\ e_0\ (\langle x_1, x_2 \rangle \Rightarrow e_3) \mapsto [v_1/x_1][v_2/x_2]e_3$ — By rule step/casep/pair

□

Exercises

Exercise 1 Design rules for the big-step evaluation judgment $e \hookrightarrow v$ which do not use any auxiliary judgment. In particular, you cannot refer to e *value* or $e \mapsto e'$, nor may design your own auxiliary judgments. You may restrict yourself to functions and Booleans, and you should presuppose that $\cdot \vdash e : \tau$.

(i) Show the rules.

(ii) Prove that if $e \hookrightarrow v$ with $\cdot \vdash e : \tau$ then v *value*.

(iii) Prove that if $e \hookrightarrow v$ (with $\cdot \vdash e : \tau$) then $e \mapsto^* v$.

Your rules should also be complete in the sense that if $e \mapsto^* v$ with v *value* then $e \hookrightarrow v$, but you do not need to prove this.

Exercise 2 Show cases for type abstraction and type application in the proof of progress (Theorem 1).

Exercise 3 Consider adding a new expression $\bot$ to our call-by-value language (with functions and Booleans) with the following evaluation and typing rules:

$$\frac{}{\bot \mapsto \bot}\ \mathsf{step/bot} \qquad \frac{}{\Gamma \vdash \bot : \tau}\ \mathsf{bot}$$

We do not change our notion of value, that is, $\bot$ is not a value.

1. Does preservation (Theorem L6.2) still hold? If not, provide a counterexample. If yes, show how the proof has to be modified to account for the new form of expression.

2. Does the canonical forms theorem (L6.4) still hold? If not, provide a counterexample. If yes, show how the proof has to be modified to account for the new form of expression.

3. Does progress (Theorem L6.3) still hold? If not, provide a counterexample. If yes, show how the proof has to be modified to account for the new form of expression.

Once we have nonterminating computation, we sometimes compare expressions using *Kleene equality*: e_1 and e_2 are Kleene equal ($e_1 \simeq e_2$) if they evaluate to the same value, or they both diverge (do not compute to a value). Since we assume we cannot observe functions, we can further restrict this definition: For $\cdot \vdash e_1 : \mathsf{bool}$ and $\cdot \vdash e_2 : \mathsf{bool}$ we write $e_1 \simeq e_2$ iff for all values v, $e_1 \mapsto^* v$ iff $e_2 \mapsto^* v$.

4. Give an example of two closed terms e_1 and e_2 of type bool such that $e_1 \simeq e_2$ but not $e_1 =_\beta e_2$, or indicate that no such example exists (no proof needed in either case).

Exercise 4 In our call-by-value language with functions, Booleans, and $\bot$ (see Exercise 3) consider the following specification of *or*, sometimes called "short-circuit or":

$$\begin{array}{lcl} \mathit{or}\ \mathsf{true}\ e & \simeq & \mathsf{true} \\ \mathit{or}\ \mathsf{false}\ e & \simeq & e \end{array}$$

where $e_1 \simeq e_2$ is Kleene equality from Exercise 3.

- We cannot define a *function or* : $\mathsf{bool} \to (\mathsf{bool} \to \mathsf{bool})$ with this behavior. Prove that it is indeed impossible.

- Show how to translate an expression *or* $e_1\ e_2$ into our language so that it satisfies the specification, and verify the given equalities by calculation.

Exercise 5 In our call-by-value language with functions, Booleans, and $\bot$ (see Exercise 3) consider the following specification of *por*, sometimes called "parallel or":

$$\begin{array}{lcl} \mathit{por}\ \mathsf{true}\ e & \simeq & \mathsf{true} \\ \mathit{por}\ e\ \mathsf{true} & \simeq & \mathsf{true} \\ \mathit{por}\ \mathsf{false}\ \mathsf{false} & \simeq & \mathsf{false} \end{array}$$

where $e_1 \simeq e_2$ is Kleene equality as in Exercises 3 and 4.

1. We cannot define a *function por* : $\mathsf{bool} \to (\mathsf{bool} \to \mathsf{bool})$ in our language with this behavior. Prove that it is indeed impossible.

2. We also cannot translate expressions *por* e_1 e_2 into our language so that the result satisfies the given properties (which you do not need to prove). Instead consider adding a new primitive form of expression $\mathsf{por}\ e_1\ e_2$ to our language.

   (a) Give one or more typing rules for $\mathsf{por}\ e_1\ e_2$.

   (b) Provide one or more evaluation rules for $\mathsf{por}\ e_1\ e_2$ so that it satisfies the given specification and, furthermore, such that preservation, canonical forms, and progress continue to hold.

   (c) Show the new case(s) in the preservation theorem.

   (d) Show the new case(s) in the progress theorem.

   (e) Do your rules satisfy sequentiality? If not, provide a counterexample. If yes, just indicate that it is the case (you do not need to prove it).

Exercise 6 *Lazy pairs*, constructed as $\langle\!| e_1, e_2 |\!\rangle$, are an alternative to the eager pairs $\langle e_1, e_2 \rangle$. Lazy pairs are typically available in "lazy" languages such as Haskell. The key differences are that a lazy pair $\langle\!| e_1, e_2 |\!\rangle$ is always a value, whether its components are or not. In that way, it is like a λ-expression, since $\lambda x.\, e$ is always a value. The second difference is that its destructors are $\mathsf{fst}\ e$ and $\mathsf{snd}\ e$ rather than a new form of case expression.

We write the type of lazy pairs as $\tau_1 \,\&\, \tau_2$. In this exercise you are asked to design the rules for lazy pairs and check their correctness.

1. Write out the new rule(s) for e *val*.

2. State the typing rules for new expressions $\langle\!| e_1, e_2 |\!\rangle$, $\mathsf{fst}\ e$, and $\mathsf{snd}\ e$.

3. Give evaluation rules for the new forms of expressions.

Instead of giving the complete set of new proof cases for the additional constructs, we only ask you to explicate a few items. Nevertheless, you need to make sure that the progress and preservation continue to hold.

4. State the new clause in the canonical forms theorem.

5. Show one case in the proof of the preservation theorem where a destructor is applied to a constructor.

6. Show the case in the proof of the progress theorem analyzing the typing rule for $\mathsf{fst}\, e$.

Exercise 7 Design the *lazy unit* $\langle\!| \, |\!\rangle$ as the nullary version of the lazy pairs of Exercise 6. We write this type as $\top$. Give the rules for values, typing, and evaluation, being careful to preserve their origins as "*lazy pairs with zero components*". Prove or refute that $1 \cong \top$.

Exercise 8 It is often stated that lazy pairs are not necessary in an eager language, because we can already define $\tau_1 \,\&\, \tau_2$ and the corresponding constructors and destructors. Fill in this table.

$\tau_1 \,\&\, \tau_2$	$\triangleq$	$(1 \to \tau_1) \times (1 \to \tau_2)$
$\langle\!| e_1, e_2 |\!\rangle$	$\triangleq$	
$\mathsf{fst}\ e$	$\triangleq$	
$\mathsf{snd}\ e$	$\triangleq$	

Abstract Syntax

Types	τ	$::=$	$\tau_1 \to \tau_2 \mid \forall \alpha.\, \tau \mid \alpha \mid \mathsf{bool}$	
Terms	e	$::=$	$x \mid \lambda x.\, e \mid e_1\, e_2$	$(\to)$
		$\mid$	$\Lambda \alpha.\, e \mid e\, [\tau]$	$(\forall)$
		$\mid$	$\mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e_1\ e_2\ e_3$	(bool)
Contexts	Γ	$::=$	$\cdot \mid \Gamma, \alpha\ \mathit{type} \mid \Gamma, x : \tau$	(all variables distinct)

Judgments

$\Gamma\ \mathit{ctx}$	Γ is a valid context	
$\Gamma \vdash \tau\ \mathit{type}$	τ is a valid type	presupposes $\Gamma\ \mathit{ctx}$
$\Gamma \vdash e : \tau$	expression e has type τ	presupposes $\Gamma\ \mathit{ctx}$, ensures $\Gamma \vdash \tau\ \mathit{type}$
$e\ \mathit{value}$	expression e is a value	presupposes $\cdot \vdash e : \tau$ for some τ
$e \mapsto e'$	expression e steps to e'	presupposes $\cdot \vdash e : \tau$ for some τ

Contexts Γ

$$\frac{}{(\cdot)\ \mathit{ctx}}\ \mathsf{ctx/emp} \qquad \frac{\Gamma\ \mathit{ctx}}{(\Gamma, \alpha\ \mathit{type})\ \mathit{ctx}}\ \mathsf{ctx/tpvar} \qquad \frac{\Gamma\ \mathit{ctx} \quad \Gamma \vdash \tau\ \mathit{type}}{(\Gamma, x : \tau)\ \mathit{ctx}}\ \mathsf{ctx/var}$$

Functions $\tau_1 \to \tau_2$

$$\frac{\Gamma \vdash \tau_1\ \mathit{type} \quad \Gamma \vdash \tau_2\ \mathit{type}}{\Gamma \vdash \tau_1 \to \tau_2\ \mathit{type}}\ \mathsf{tp/arrow}$$

$$\frac{\Gamma, x_1 : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \lambda x_1{:}\tau_1.\, e_2 : \tau_1 \to \tau_2}\ \mathsf{tp/lam} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \mathsf{tp/var} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_1}\ \mathsf{tp/app}$$

$$\frac{}{\lambda x.\, e\ \mathit{value}}\ \mathsf{val/lam} \qquad \frac{e_2\ \mathit{value}}{(\lambda x.\, e_1)\, e_2 \mapsto [e_2/x]e_1}\ \mathsf{step/app/lam}$$

$$\frac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2}\ \mathsf{step/app}_1 \qquad \frac{e_1\ \mathit{value} \quad e_2 \mapsto e_2'}{e_1\, e_2 \mapsto e_1\, e_2'}\ \mathsf{step/app}_2$$

Polymorphic Types $\forall\alpha.\,\tau$

$$\frac{\alpha\ \mathit{type} \in \Gamma}{\Gamma \vdash \alpha\ \mathit{type}}\ \mathsf{tp/tpvar} \qquad \frac{\Gamma, \alpha\ \mathit{type} \vdash \tau\ \mathit{type}}{\Gamma \vdash \forall\alpha.\,\tau\ \mathit{type}}\ \mathsf{tp/forall}$$

$$\frac{\Gamma, \alpha\ \mathit{type} \vdash e : \tau}{\Gamma \vdash \Lambda\alpha.\,e : \forall\alpha.\,\tau}\ \mathsf{tp/tplam} \qquad \frac{\Gamma \vdash e : \forall\alpha.\,\tau \quad \Gamma \vdash \sigma\ \mathit{type}}{\Gamma \vdash e\,[\sigma] : [\sigma/\alpha]\tau}\ \mathsf{tp/tpapp}$$

$$\frac{}{\Lambda\alpha.\,e\ \mathit{value}}\ \mathsf{val/tplam} \qquad \frac{}{(\Lambda\alpha.\,e)\,[\tau] \mapsto [\tau/\alpha]e}\ \mathsf{step/tpapp/tplam} \qquad \frac{e \mapsto e'}{e\,[\tau] \mapsto e'\,[\tau]}\ \mathsf{step/tpapp}$$

Booleans bool

$$\frac{}{\Gamma \vdash \mathsf{bool}\ \mathit{type}}\ \mathsf{tp/bool}$$

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}}\ \mathsf{tp/true} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}\ \mathsf{tp/false} \qquad \frac{\Gamma \vdash e_1 : \mathsf{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathsf{if}\ e_1\ e_2\ e_3 : \tau}\ \mathsf{tp/if}$$

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

$$\frac{}{\mathsf{if}\ \mathsf{true}\ e_2\ e_3 \mapsto e_2}\ \mathsf{step/if/true} \qquad \frac{}{\mathsf{if}\ \mathsf{false}\ e_2\ e_3 \mapsto e_3}\ \mathsf{step/if/false} \qquad \frac{e_1 \mapsto e_1'}{\mathsf{if}\ e_1\ e_2\ e_3 \mapsto \mathsf{if}\ e_1'\ e_2\ e_3}\ \mathsf{step/if}$$

Theorems

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

Progress. For every expression $\cdot \vdash e : \tau$ either $e \mapsto e'$ for some e' or e *value*.

Finality of Values. There is no $\cdot \vdash e : \tau$ such that $e \mapsto e'$ for some e' and e *value*.

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

Lecture Notes on Sums

15-814: Types and Programming Languages
Frank Pfenning

Lecture 9
Tuesday, September 30, 2020

1 Introduction

In this lecture we continue to build up our small functional language, isolating fundamental building blocks for constructing data and functions. We begin with the unit type 1 with just a single value, the unit element. After investigating some elementary properties of the unit we introduce *disjoint sums* which is the second form of data aggregation after products (whose values are pairs). With those type constructors in hand, we can represent a variety of interesting types with a finite number of elements, but not yet types with infinitely elements except opaquely through functional representations. This gap will be addressed in the next lecture by introducing *recursive types*.

2 The Unit Type

Even though it may not look particularly useful initially, we now introduce the unit type 1, inhabited by exactly one value $\langle\,\rangle$. It is also the nullary version of (eager) pairs (think: a pair $\langle v_1, v_2 \rangle$ has two components while $\langle\,\rangle$ has zero).

$$\frac{}{\Gamma \vdash \langle\,\rangle : 1}\ \mathsf{tp/unit} \qquad \frac{}{\langle\,\rangle\ \mathit{val}}\ \mathsf{val/unit}$$

With pairs, there is a single destructor thats extracts two components, so for

the unit type there is also a single destructor that extracts zero components.

$$\frac{\Gamma \vdash e : 1 \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash \mathsf{case}\ e\ (\langle\,\rangle \Rightarrow e') : \tau'}\ \mathsf{tp/caseu}$$

In the dynamics, we only reduce the new version of the case construct, since the unit element is already a value.

$$\frac{e_0 \mapsto e_0'}{\mathsf{case}\ e_0\ (\langle\,\rangle \Rightarrow e_1) \mapsto \mathsf{case}\ e_0'\ (\langle\,\rangle \Rightarrow e_1)}\ \mathsf{step/caseu}_0$$

$$\frac{}{\mathsf{case}\ \langle\,\rangle\ (\langle\,\rangle \Rightarrow e_1) \mapsto e_1}\ \mathsf{step/caseu/unit}$$

It is easy to verify that our theorems continue to hold, and that $\cdot \vdash e : 1$ and e *val* imply that $e = \langle\,\rangle$ (as an extension of the canonical forms theorem).

The unit type is not as useless as it might appear. In C, the unit type is called `void` and indicates that a function does not return a value. In a functional language with effects, you will often see code such as

```
let val () = print(v)
```

to execute an effect and return the only value of type 1 (called `unit` in Standard ML). We will also see that it is actually quite important in concert with disjoint sums.

3 Type Isomorphisms

Intuitively, 1 should be the nullary product, which we might hope to express with something like $\tau \times 1 = \tau$. But "=" does not make any sense here: these type are different because the are inhabited by different terms. Instead, what we want to say is the the type are *isomorphic*, written as $\tau \times 1 \cong \tau$. Again, intuitively speaking, two types are isomorphic if they have the same information contents. In the general case, we say that $\tau \cong \sigma$ if there are two functions,

Forth : $\tau \to \sigma$ and *Back* : $\sigma \to \tau$ such that they compose to the identity in both directions. Writing it out explicitly:

$$\begin{array}{lcllcl} \mathit{Forth} & : & \tau \to \sigma \\ \mathit{Back} & : & \sigma \to \tau \\ \mathit{Back} \circ \mathit{Forth} & = & \lambda x.\, x & : & \tau \to \tau \\ \mathit{Forth} \circ \mathit{Back} & = & \lambda y.\, y & : & \sigma \to \sigma \end{array}$$

When comparing functions (or expressions in general) we have to decide which form of equality to use. The simple β- or even $\beta\eta$-equivalence we used before does not apply here for two reasons: (1) we have many other types besides functions, and (2) we have decided that functions are opaque, so we should not try to analyze their structure. The latter observation pushes us in the direction of an *extensional equality*: two functions are equal if they return equal results when applied to the same argument. This is based on the idea that the structure of functions cannot be observed, but their behavior on arguments can. Because we are in a call-by-value language, this means we have to verify their behavior when applied to arbitrary values of the correct type. On the other hand, types like (eager) products are observable, so we can just compare their components directly. We write $v \sim v' : \tau$ if two expressions are extensionally equal, presupposing that v and v' are closed values of type τ. For general expressions, we write $e \approx e' : \tau$ which is defined by evaluating e and e' to a value and comparing the results.

Expressions: $e \approx e' : \tau$ iff $e \mapsto^* v$, $e' \mapsto^* v'$ with v, v' values, and $v \sim v' : \tau$, or neither e nor e' evaluate to value.

As a side remark, in our current language all well-typed expressions have a value, so the final condition is vacuous but will become relevant during the next language. This definition means we now have to compare *values*. For functions, this will refer back to the definition on expressions, but only at a smaller type.

Functions: $v \sim v' : \tau_1 \to \tau_2$ iff for all $v_1 : \tau_1$ we have $v\ v_1 \approx v'\ v_1 : \tau_2$.

Pairs: $v \sim v' : \tau_1 \times \tau_2$ iff $v = \langle v_1, v_2 \rangle$, $v' = \langle v_1', v_2' \rangle$ and $v_1 \sim v_1' : \tau_1$ and $v_2 \sim v_2' : \tau_2$.

Units: $v \sim v' : 1$ iff $v = \langle\,\rangle$ and $v' = \langle\,\rangle$ (which is always the case, by the canonical forms theorem).

We will later have occasion to revisit and slightly revise this definition, but it is adequate for now.

Returning to the specific example of $\tau \cong \tau \times 1$, let's verify this isomorphism. We define

$$
\begin{array}{lcl}
\mathit{Forth} & : & \tau \to (\tau \times 1) \\
\mathit{Forth} & = & \lambda x.\, \langle x, \langle\,\rangle \rangle \\[1ex]
\mathit{Back} & : & (\tau \times 1) \to \tau \\
\mathit{Back} & = & \lambda p.\, \mathsf{case}\ p\ (\langle x, y \rangle \Rightarrow x)
\end{array}
$$

To check that $\mathit{Back} \circ \mathit{Forth} \approx \lambda x.\, x : \tau \to \tau$ we apply both sides to an arbitrary value $v : \tau$ and calculate

$$
\begin{array}{rcl}
\mathsf{LHS} & = & (\mathit{Back} \circ \mathit{Forth})\, v \\
& \mapsto^* & \mathit{Back}\, ((\lambda x.\, \langle x, \langle\,\rangle\rangle)\, v) \\
& \mapsto & \mathit{Back}\, \langle v, \langle\,\rangle\rangle \\
& = & (\lambda p.\, \mathsf{case}\ p\ (\langle x, y\rangle \Rightarrow x))\, \langle v, \langle\,\rangle\rangle \\
& \mapsto & \mathsf{case}\ \langle v, \langle\,\rangle\rangle\ (\langle x, y\rangle \Rightarrow x) \\
& \mapsto & v \\
\\
\mathsf{RHS} & = & (\lambda x.\, x)\ v \\
& \mapsto & v
\end{array}
$$

So the two functions are extensionally equal.

For the other direction, we exploit that, by the canonical forms theorem, a value of type $v : \tau \times 1$ must have the form $v = \langle v', \langle\,\rangle\rangle$:

$$
\begin{array}{rcl}
\mathsf{LHS} & = & (\mathit{Forth} \circ \mathit{Back})\, v \\
& \mapsto^* & \mathit{Forth}\, (\mathit{Back}\, \langle v', \langle\,\rangle\rangle) \\
& = & \mathit{Forth}\, ((\lambda p.\, \mathsf{case}\ p\ (\langle x, y\rangle \Rightarrow x))\, \langle v', \langle\,\rangle\rangle) \\
& \mapsto & \mathit{Forth}\, (\mathsf{case}\ \langle v', \langle\,\rangle\rangle\ (\langle x, y\rangle \Rightarrow x)) \\
& \mapsto & \mathit{Forth}\, v' \\
& = & (\lambda x.\, \langle x, \langle\,\rangle\rangle)\, v' \\
& \mapsto & \langle v', \langle\,\rangle\rangle \\
& = & v \\
\\
\mathsf{RHS} & = & (\lambda y.\, y)\, v \\
& \mapsto & v
\end{array}
$$

Again both sides are equal, so both compositions are equal to the identity, witnessing the isomorphism between τ and $\tau \times 1$.

4 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 is 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 $\mathbf{l}$ and $\mathbf{r}$ as *tags* or *labels* and write $\mathbf{l} \cdot e_1$ for the expression of type $\tau_1 + \tau_2$ if $e_1 : \tau_1$ and, analogously, $\mathbf{r} \cdot e_2$ if $e_2 : \tau_2$.

$$\frac{\Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \mathbf{l} \cdot e_1 : \tau_1 + \tau_2}\ \mathsf{tp/left} \qquad \frac{\Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{r} \cdot e_2 : \tau_1 + \tau_2}\ \mathsf{tp/right}$$

These two forms of expressions allow us to form elements 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.

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \sigma \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \sigma}{\Gamma \vdash \mathsf{case}\ e\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \mathbf{r} \cdot x_2 \Rightarrow e_2) : \sigma}\ \mathsf{tp/cases}$$

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 $\mathbf{l} \cdot v_1$ 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 it 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 σ, which is also the type of the whole case.

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

$$\frac{e\ \mathit{value}}{\mathbf{l} \cdot e\ \mathit{value}}\ \mathsf{val/left} \qquad \frac{e\ \mathit{value}}{\mathbf{r} \cdot e\ \mathit{value}}\ \mathsf{val/right}$$

$$\frac{e \mapsto e'}{\mathbf{l} \cdot e \mapsto l \cdot e'}\ \mathsf{step/left} \qquad \frac{e \mapsto e'}{\mathbf{r} \cdot e \mapsto \mathbf{r} \cdot e'}\ \mathsf{step/right}$$

$$\frac{e_0 \mapsto e_0'}{\mathsf{case}\ e_0\ (\ldots \mid \ldots) \mapsto \mathsf{case}\ e_0'\ (\ldots \mid \ldots)}\ \mathsf{step/cases_0}$$

$$\frac{v_1\ \mathit{value}}{\mathsf{case}\ (\mathbf{l} \cdot v_1)\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \ldots) \mapsto [v_1/x_1]e_1}\ \mathsf{step/cases/left}$$

$$\frac{v_2\ \mathit{value}}{\mathsf{case}\ (\mathbf{r} \cdot v_2)\ (\ldots \mid \mathbf{r} \cdot x_2 \Rightarrow e_2) \mapsto [v_2/x_2]e_2}\ \mathsf{step/cases/right}$$

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. We can see that the type of an expression $\mathbf{l} \cdot e_1$ 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. In the "official" rule, therefore, we should check that τ_2 is a valid type (see Section 8).

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:

$$\frac{e_1 \mapsto e_1'}{\mathbf{l} \cdot e_1 \mapsto \mathbf{l} \cdot e_1'}\ \mathsf{step/left}$$

where $e = \mathbf{l} \cdot e_1$ and $e' = \mathbf{l} \cdot e_1'$

$\cdot \vdash \mathbf{l} \cdot e_1 : \tau_1 + \tau_2$	Assumption
$\cdot \vdash e_1 : \tau_1$	By inversion
$\cdot \vdash e_1' : \tau_1$	By ind.hyp.
$\cdot \vdash \mathbf{l} \cdot e_1' : \tau_1 + \tau_2$	By rule step/left

Case: Rule step/right: analogous to step/left.

Case: Rule $\mathsf{step/cases}_0$: similar to the previous two cases.

Case:

$$\frac{v_1\ \mathit{value}}{\mathsf{case}\ (\mathbf{l} \cdot v_1)\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \ldots) \mapsto [v_1/x_1]e_1}\ \mathsf{step/cases/left}$$

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

[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.

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

Case: Rule step/case/sum/r: analogous to the previous case.

□

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 (iv).

Theorem 2 (Canonical Forms) *Assume v value.*

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

(ii) If $\cdot \vdash v : \tau_1 \times \tau_2$ then $v = \langle v_1, v_2 \rangle$ for some v_1 value and v_2 value.

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

(iv) If $\cdot \vdash v : \tau_1 + \tau_2$ then $v = \mathbf{l} \cdot v_1$ for some v_1 value or $v = \mathbf{r} \cdot v_2$ for some v_2 value.

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 value.

Proof: By rule induction on the given typing derivation.

Cases: For constructs pertaining to types $\tau_1 \to \tau_2$, bool, $\tau_1 \times \tau_2$, and 1 just as before since we did not change their rules.

Case:

$$\frac{\cdot \vdash e_1 : \tau_1}{\cdot \vdash \mathbf{l} \cdot e_1 : \tau_1 + \tau_2}\ \mathsf{tp/left}$$

where $e = \mathbf{l} \cdot e_1$.

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

$e_1 \mapsto e_1'$ — Subcase
$\mathbf{l} \cdot e_1 \mapsto \mathbf{l} \cdot e_1'$ — By rule step/left

e_1 *value* — Subcase
$\mathbf{l} \cdot e_1$ *value* — By rule val/l

Case: Rule tp/right is symmetric to previous case.

Case:

$$\frac{\cdot \vdash e_0 : \tau_1 + \tau_2 \quad x_1 : \tau_1 \vdash e_1 : \tau \quad x_2 : \tau_2 \vdash e_2 : \tau}{\cdot \vdash \mathsf{case}\ e_0\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \mathbf{r} \cdot x_2 \Rightarrow e_2) : \tau}\ \mathsf{tp/cases}$$

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

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

$e_0 \mapsto e_0'$ — Subcase
$e = \mathsf{case}\ e_0\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \mathbf{r} \cdot x_2 \Rightarrow e_2)$
$\quad \mapsto \mathsf{case}\ e_0'\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \mathbf{r} \cdot x_2 \Rightarrow e_2)$ — By rule step/cases$_0$

e_0 *value* — Subcase
$e_0 = \mathbf{l} \cdot e_0'$ for some e_0' *value*
or $e_0 = \mathbf{r} \cdot e_0'$ for some e_0' *value* — By canonical forms (Theorem 2)

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

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

□

5 Examples of Sums

Once we have sums and the unit type from the previous lecture, we can now *define* the Boolean type.

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

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.

Using 1 we can define other types. For example

$$\textit{option}\ \tau \quad = \quad \tau + 1$$

represents an optional value of type τ. Its values are $\mathbf{l} \cdot v$ for $v : \tau$ (we have a value) or $\mathbf{r} \cdot \langle\,\rangle$ (we have not value of type τ).

A more interesting example would be the natural numbers:

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

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

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

but supporting this style of recursive type definition is not straightforward. 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.

6 The Empty Type

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

that $0 + \tau \cong \tau$. There are no constructors and no values of this type, so the e *value* judgment is not extended.

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

$$\frac{\Gamma \vdash e_0 : 0 \quad \Gamma \vdash \tau \; \textit{type}}{\Gamma \vdash \mathsf{case}\; e_0 \; (\,) : \tau}\; \mathsf{tp/casez}$$

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

$$\frac{e_0 \mapsto e_0'}{\mathsf{case}\; e_0 \; (\,) \mapsto \mathsf{case}\; e_0' \; (\,)}\; \mathsf{step/casez}_0$$

Progress and preservation extend somewhat easily, and the canonical forms property is extended with

(v) If $\cdot \vdash v : 0$ 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:

$$\begin{array}{lcl} \lambda x.\, x & : & 0 \to 0 \\ \lambda x.\, \mathsf{case}\; x \; (\,) & : & 0 \to 0 \\ \lambda x.\, \bot & : & 0 \to 0 \end{array}$$

where $\bot$ is introduced in Exercise L8.3.

7 More Isomorphisms

The next example illustrates and important technique and therefore has a name: *Currying*, after the logician Haskell Curry. Instead of a function taking a pair as an argument we can take the two arguments in succession. And vice versa! We express this with the following type isomorphism:[2]

$$(\tau \times \sigma) \to \rho \cong \tau \to (\sigma \to \rho)$$

[2] In lecture, we only discussed the existence of this isomorphism without providing or checking the function witnessing it.

We program the *Forth* and *Back* functions in a type-directed manner. We show the process only once, but we recommend thinking about coding in this general style. We have

$$\mathit{Forth} : ((\tau \times \sigma) \to \rho) \to (\tau \to (\sigma \to \rho))$$

We see this function takes three arguments in succession: first a function of type $(\tau \times \sigma) \to \rho$, then a value of type τ followed by a value of type σ. So we start the code with three λ-abstractions, followed by an as yet unknown body.

$$\mathit{Forth} = \lambda f.\, \lambda x.\, \lambda y.\, \boxed{}$$

where

$$\begin{array}{rcl} f & : & (\tau \times \sigma) \to \rho \\ x & : & \tau \\ y & : & \sigma \\ \boxed{} & : & \rho \end{array}$$

We can see that only f produces a result of type ρ, and it requires a pair of type $\tau \times \sigma$ as an argument. Fortunately, we have x and y available to form the two components of the pair. Filling everything in:

$$\begin{array}{rcl} \mathit{Forth} & : & ((\tau \times \sigma) \to \rho) \to (\tau \to (\sigma \to \rho)) \\ \mathit{Forth} & = & \lambda f.\, \lambda x.\, \lambda y.\, f\,\langle x, y\rangle \end{array}$$

Programming the other direction in a similar manner yields

$$\begin{array}{rcl} \mathit{Back} & : & (\tau \to (\sigma \to \rho)) \to ((\tau \times \sigma) \to \rho) \\ \mathit{Back} & = & \lambda g.\, \lambda p.\, \mathsf{case}\ p\ (\langle x, y\rangle \Rightarrow g\,x\,y) \end{array}$$

Let's see if we can verify that *Forth* and *Back* compose to the identity, picking an arbitrary direction first.

$$\mathit{Back} \circ \mathit{Forth} = \lambda f.\, \mathit{Back}\ (\mathit{Forth}\ f) \stackrel{?}{=} \lambda f.\, f : ((\tau \times \sigma) \to \rho) \to ((\tau \times \sigma) \to \rho)$$

To compare these two functions we apply them to an arbitrary *value* $v : (\tau \times \sigma) \to \rho$ and compare the result. We reason:

$$\begin{array}{rl} & (\lambda f.\, \mathit{Back}\ (\mathit{Forth}\ f))\ v \\ \mapsto & \mathit{Back}\ (\mathit{Forth}\ v) \\ = & \mathit{Back}\ ((\lambda f.\, \lambda x.\, \lambda y.\, f\,\langle x, y\rangle)\ v) \\ \mapsto & \mathit{Back}\ (\lambda x.\, \lambda y. v\,\langle x, y\rangle) \\ = & (\lambda g.\, \lambda p.\, \mathsf{case}\ p\ (\langle x, y\rangle \Rightarrow g\,x\,y))\ (\lambda x.\, \lambda y.\, v\,\langle x, y\rangle) \\ \mapsto & \lambda p.\, \mathsf{case}\ p\ (\langle x, y\rangle \Rightarrow (\lambda x'.\, \lambda y'.\, v\,\langle x', y'\rangle)\,x\,y) \\ \stackrel{?}{=} & v : (\tau \times \sigma) \to \rho \end{array}$$

In the last step we renamed some variable to avoid confusion.

Again, we are comparing two functions, this time on an argument of type $\tau \times \sigma$. These two functions are the same if the return the same result if we apply them to the pair $\langle v_1, v_2\rangle$ of two values $v_1 : \tau$ and $v_2 : \tau_2$. We use values here because the type $\tau \times \sigma$ is observable, and a value of this type is a pair of two values. Then we find:

$$\begin{array}{ll} & (\lambda p.\, \mathsf{case}\; p\; (\langle x, y\rangle \Rightarrow (\lambda x'.\, \lambda y'.\, v\, \langle x', y'\rangle)\, x\, y))\; \langle v_1, v_2\rangle \\ \mapsto & \mathsf{case}\; \langle v_1, v_2\rangle\; (\langle x, y\rangle \Rightarrow (\lambda x'.\, \lambda y'.\, v\, \langle x', y'\rangle)\, x\, y) \\ \mapsto & (\lambda x'.\, \lambda y'.\, v\, \langle x', y'\rangle)\, v_1\, v_2 \\ \mapsto^2 & v\, \langle v_1, v_2\rangle \\ = & v\, \langle v_1, v_2\rangle \end{array}$$

The final equality is the one we wanted to check. Checking the other direction is left to Exercise 3.

For the purpose of reasoning about type isomorphisms we extend our notion of extensional equality by adding a case for sums.

Sums: $v \sim v' : \tau_1 + \tau_2$ iff either $v = \mathbf{l} \cdot v_1$, $v' = \mathbf{l} \cdot v_1'$ and $v_1 \sim v_1' : \tau_1$ or $v = \mathbf{r} \cdot v_2$, $v' = \mathbf{r} \cdot v_2'$ and $v_2 \sim v_2' : \tau_2$

One of the properties that is easy to check is that $\tau + \sigma \cong \sigma + \tau$. We can speculate some other isomorphism, based on an kind of arithmetic interpretation of the types. For example, $\times$ might distribute over $+$:

$$\tau \times (\sigma + \rho) \;\stackrel{?}{\cong}\; (\tau \times \sigma) + (\tau \times \rho)$$

Some strange ones pop up if we think of $\sigma \to \tau$ as τ^σ. The reason to even conjecture this is because we have already checked that $\rho \to (\sigma \to \tau) \cong (\rho \times \sigma) \to \tau$ which could be written as $(\tau^\sigma)^\rho \cong \tau^{\sigma \times \rho}$.

$$\begin{array}{rcl} 2 \to \tau & \stackrel{?}{\cong} & \tau \times \tau \\ 1 \to \tau & \stackrel{?}{\cong} & \tau \\ 0 \to \tau & \stackrel{?}{\cong} & 1 \end{array}$$

While odd, these are not ridiculous. Consider the first one, and recall that $1 + 1 \cong \mathsf{bool}$. In one direction, we can apply the given function to true and false to obtain two values, in other direction we can set the given values as result of the function on true and false, respectively. Do these functions constitute an ismorphism?

An example of types that are *not* isomorphic in general would be

$$\tau \quad \ncong \quad \tau \times \tau$$

In order to show, that they are not always isomorphic it suffices to provide a counterexample where the cardinality of the set of values do not match. (Recall that isomorphism implies equal cardinality, but also that the functions *Forth* and *Back* are expressible in our language.) In this example, if we pick $\tau = 2$ then $v : \tau$ for two values ($\mathbf{l} \cdot \langle\,\rangle$ and $\mathbf{r} \cdot \langle\,\rangle$, to be precise) while the right-hand side contains four values. On the other hand, the isomorphism does hold for $\tau = 1$ since both sides have exactly one value ($\langle\,\rangle$ for the left-hand side and $\langle\,\langle\,\rangle, \langle\,\rangle\,\rangle$ for the right-hand side).

8 Summary

See 09-sums-rules.pdf for a summary of the rules.

Exercises

Exercise 1 Exhibit the functions *Forth* and *Back* witnessing the following isomorphisms. You do not need to prove that they constitute an ismorphism, just show the functions. We remain here in the pure language of Section 8 where every function is terminating.

(i) $\tau \times (\sigma + \rho) \cong (\tau \times \sigma) + (\tau \times \rho)$

(ii) $2 \to \tau \cong \tau \times \tau$

(iii) $1 \to \tau \cong \tau$

(iv) $0 \to \tau \cong 1$

(v) $(\sigma + \rho) \to \tau \cong (\sigma \to \tau) \times (\rho \to \tau)$

Exercise 2 Many of the type isomorphisms follow arithmetic equalities, interpreting $\tau + \sigma$ as addition, $\tau \times \sigma$ as multiplication, and $\tau \to \sigma$ as exponentiation σ^τ (see Exercise 1).

But there are also differences. In arithmetic, we have an additive inverse $-a$ such that $a + (-a) = 0$. Prove that there can be no general type constructor $-\tau$ such that $\tau + (-\tau) \cong 0$.

Exercise 3 Verify that the composition *Forth* $\circ$ *Back* $\approx \lambda g.\, g$ where *Forth* and *Back* coerce from a curried function to its tupled counterpart.

$$\begin{array}{lcl} \mathit{Forth} & : & ((\tau \times \sigma) \to \rho) \to (\tau \to (\sigma \to \rho)) \\ \mathit{Forth} & = & \lambda f.\, \lambda x.\, \lambda y.\, f\, \langle x, y \rangle \\ \\ \mathit{Back} & : & (\tau \to (\sigma \to \rho)) \to ((\tau \times \sigma) \to \rho) \\ \mathit{Back} & = & \lambda g.\, \lambda p.\, \mathsf{case}\ p\ (\langle x, y \rangle \Rightarrow g\, x\, y) \end{array}$$

Abstract Syntax

$$
\begin{array}{llll}
\text{Types} & \tau & ::= & \tau_1 \to \tau_2 \mid \forall \alpha.\, \tau \mid \alpha \mid \tau_1 \times \tau_2 \mid 1 \mid \tau_1 + \tau_2 \mid 0 & \\
\text{Terms} & e & ::= & x \mid \lambda x.\, e \mid e_1\, e_2 & (\to) \\
 & & \mid & \Lambda \alpha.\, e \mid e\,[\tau] & (\forall) \\
 & & \mid & \langle e_1, e_2 \rangle \mid \mathsf{case}\ e\ (\langle x_1, x_2 \rangle \Rightarrow e') & (\times) \\
 & & \mid & \langle\,\rangle \mid \mathsf{case}\ e\ (\langle\,\rangle \Rightarrow e') & (1) \\
 & & \mid & \mathbf{l} \cdot e \mid \mathbf{r} \cdot e \mid \mathsf{case}\ e\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \mathbf{r} \cdot x_2 \Rightarrow e_2) & (+) \\
 & & \mid & \mathsf{case}\ e\ (\,) & (0) \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, \alpha\ \mathit{type} \mid \Gamma, x : \tau & \text{(all variables distinct)}
\end{array}
$$

Judgments

Γ *ctx*	Γ is a valid context	
$\Gamma \vdash \tau$ *type*	τ is a valid type	presupposes Γ *ctx*
$\Gamma \vdash e : \tau$	expression e has type τ	presupposes Γ *ctx*, ensures $\Gamma \vdash \tau$ *type*
e *value*	expression e is a value	presupposes $\cdot \vdash e : \tau$ for some τ
$e \mapsto e'$	expression e steps to e'	presupposes $\cdot \vdash e : \tau$ for some τ

Contexts Γ

$$
\frac{}{(\cdot)\ \mathit{ctx}}\ \mathsf{ctx/emp}
\qquad
\frac{\Gamma\ \mathit{ctx}}{(\Gamma, \alpha\ \mathit{type})\ \mathit{ctx}}\ \mathsf{ctx/tpvar}
\qquad
\frac{\Gamma\ \mathit{ctx} \quad \Gamma \vdash \tau\ \mathit{type}}{(\Gamma, x : \tau)\ \mathit{ctx}}\ \mathsf{ctx/var}
$$

Functions $\tau_1 \to \tau_2$

$$
\frac{\Gamma \vdash \tau_1\ \mathit{type} \quad \Gamma \vdash \tau_2\ \mathit{type}}{\Gamma \vdash \tau_1 \to \tau_2\ \mathit{type}}\ \mathsf{tp/arrow}
$$

$$
\frac{\Gamma \vdash \tau_1\ \mathit{type} \quad \Gamma, x_1 : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \lambda x_1{:}\tau_1.\, e_2 : \tau_1 \to \tau_2}\ \mathsf{tp/lam}
\qquad
\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \mathsf{tp/var}
\qquad
\frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_1}\ \mathsf{tp/app}
$$

$$
\frac{}{\lambda x.\, e\ \mathit{value}}\ \mathsf{val/lam}
\qquad
\frac{e_2\ \mathit{value}}{(\lambda x.\, e_1)\, e_2 \mapsto [e_2/x]e_1}\ \mathsf{step/app/lam}
$$

$$
\frac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2}\ \mathsf{step/app}_1
\qquad
\frac{e_1\ \mathit{value} \quad e_2 \mapsto e_2'}{e_1\, e_2 \mapsto e_1\, e_2'}\ \mathsf{step/app}_2
$$

Polymorphic Types $\forall\alpha.\,\tau$

$$\frac{\alpha\ \mathit{type} \in \Gamma}{\Gamma \vdash \alpha\ \mathit{type}}\ \mathsf{tp/tpvar} \qquad \frac{\Gamma, \alpha\ \mathit{type} \vdash \tau\ \mathit{type}}{\Gamma \vdash \forall\alpha.\,\tau\ \mathit{type}}\ \mathsf{tp/forall}$$

$$\frac{\Gamma, \alpha\ \mathit{type} \vdash e : \tau}{\Gamma \vdash \Lambda\alpha.\,e : \forall\alpha.\,\tau}\ \mathsf{tp/tplam} \qquad \frac{\Gamma \vdash e : \forall\alpha.\,\tau \quad \Gamma \vdash \sigma\ \mathit{type}}{\Gamma \vdash e\,[\sigma] : [\sigma/\alpha]\tau}\ \mathsf{tp/tpapp}$$

$$\frac{}{\Lambda\alpha.\,e\ \mathit{value}}\ \mathsf{val/tplam} \qquad \frac{}{(\Lambda\alpha.\,e)\,[\tau] \mapsto [\tau/\alpha]e}\ \mathsf{step/tpapp/tplam} \qquad \frac{e \mapsto e'}{e\,[\tau] \mapsto e'\,[\tau]}\ \mathsf{step/tpapp}$$

Pairs $\tau_1 \times \tau_2$

$$\frac{\Gamma \vdash \tau_1\ \mathit{type} \quad \Gamma \vdash \tau_2\ \mathit{type}}{\Gamma \vdash \tau_1 \times \tau_2\ \mathit{type}}\ \mathsf{tp/prod}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 \times \tau_2}\ \mathsf{tp/pair} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2 \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e' : \tau'}{\Gamma \vdash \mathsf{case}\ e\ (\langle x_1, x_2\rangle \Rightarrow e') : \tau'}\ \mathsf{tp/casep}$$

$$\frac{e_1\ \mathit{value} \quad e_2\ \mathit{value}}{\langle e_1, e_2\rangle\ \mathit{value}}\ \mathsf{val/pair} \qquad \frac{v_1\ \mathit{value} \quad v_2\ \mathit{value}}{\mathsf{case}\ \langle v_1, v_2\rangle\ (\langle x_1, x_2\rangle \Rightarrow e_3) \mapsto [v_1/x_1][v_2/x_2]e_3}\ \mathsf{step/casep/pair}$$

$$\frac{e_1 \mapsto e_1'}{\langle e_1, e_2\rangle \mapsto \langle e_1', e_2\rangle}\ \mathsf{step/pair_1} \qquad \frac{v_1\ \mathit{value} \quad e_2 \mapsto e_2'}{\langle v_1, e_2\rangle \mapsto \langle v_1, e_2'\rangle}\ \mathsf{step/pair_2}$$

$$\frac{e_0 \mapsto e_0'}{\mathsf{case}\ e_0\ (\langle x_1, x_2\rangle \Rightarrow e_3) \mapsto \mathsf{case}\ e_0'\ (\langle x_1, x_2\rangle \Rightarrow e_3)}\ \mathsf{step/casep_0}$$

Unit 1

$$\frac{}{\Gamma \vdash 1\ \mathit{type}}\ \textsf{tp/one} \qquad \frac{}{\Gamma \vdash \langle\,\rangle : 1}\ \textsf{tp/unit} \qquad \frac{\Gamma \vdash e : 1 \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash \textsf{case}\ e\ (\langle\,\rangle \Rightarrow e') : \tau'}\ \textsf{tp/caseu}$$

$$\frac{}{\langle\,\rangle\ \mathit{value}}\ \textsf{val/unit} \qquad \frac{}{\textsf{case}\ \langle\,\rangle\ (\langle\,\rangle \Rightarrow e) \mapsto e}\ \textsf{step/caseu/unit}$$

$$\frac{e_0 \mapsto e_0'}{\textsf{case}\ e_0\ (\langle\,\rangle \Rightarrow e_1) \mapsto \textsf{case}\ e_0'\ (\langle\,\rangle \Rightarrow e_1)}\ \textsf{step/caseu}_0$$

Sums $\tau_1 + \tau_2$

$$\frac{\Gamma \vdash \tau_1\ \mathit{type} \quad \Gamma \vdash \tau_2\ \mathit{type}}{\Gamma \vdash \tau_1 + \tau_2\ \mathit{type}}\ \textsf{tp/sum}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash \tau_2\ \mathit{type}}{\Gamma \vdash \mathbf{l} \cdot e_1 : \tau_1 + \tau_2}\ \textsf{tp/left} \qquad \frac{\Gamma \vdash \tau_1\ \mathit{type} \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{r} \cdot e_2 : \tau_1 + \tau_2}\ \textsf{tp/right}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \sigma \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \sigma}{\Gamma \vdash \textsf{case}\ e\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \mathbf{r} \cdot x_2 \Rightarrow e_2) : \sigma}\ \textsf{tp/cases}$$

$$\frac{e_1\ \mathit{value}}{\mathbf{l} \cdot e_1\ \mathit{value}}\ \textsf{val/left} \qquad \frac{e_2\ \mathit{value}}{\mathbf{r} \cdot e_2\ \mathit{value}}\ \textsf{val/right}$$

$$\frac{v_1\ \mathit{value}}{\textsf{case}\ \mathbf{l} \cdot v_1\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \mathbf{r} \cdot x_2 \Rightarrow e_2) \mapsto [v_1/x_1]e_1}\ \textsf{step/cases/left}$$

$$\frac{v_2\ \mathit{value}}{\textsf{case}\ \mathbf{r} \cdot v_2\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \mathbf{r} \cdot x_2 \Rightarrow e_2) \mapsto [v_2/x_2]e_2}\ \textsf{step/cases/right}$$

$$\frac{e_1 \mapsto e_1'}{\mathbf{l} \cdot e_1 \mapsto \mathbf{l} \cdot e_1'}\ \textsf{step/left} \qquad \frac{e_2 \mapsto e_2'}{\mathbf{r} \cdot e_2 \mapsto \mathbf{r} \cdot e_2'}\ \textsf{step/right}$$

$$\frac{e_0 \mapsto e_0'}{\textsf{case}\ e_0\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \mathbf{r} \cdot x_2 \Rightarrow e_2) \mapsto \textsf{case}\ e_0\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \mathbf{r} \cdot x_2 \Rightarrow e_2)}\ \textsf{step/cases}_0$$

Empty Type 0

$$\frac{}{\Gamma \vdash 0 \ \mathit{type}} \ \mathsf{tp/zero} \qquad \text{(no constructor)} \qquad \frac{\Gamma \vdash e_0 : 0 \quad \Gamma \vdash \tau \ \mathit{type}}{\Gamma \vdash \mathsf{case}\ e_0\ (\,) : \tau} \ \mathsf{tp/casez}$$

$$\text{(no values)} \qquad \frac{e_0 \mapsto e_0'}{\mathsf{case}\ e_0\ (\,) \mapsto \mathsf{case}\ e_0'\ (\,)} \ \mathsf{step/casez}_0$$

Theorems

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

Progress. For every expression $\cdot \vdash e : \tau$ either $e \mapsto e'$ for some e' or e *value*.

Finality of Values. There is no $\cdot \vdash e : \tau$ such that $e \mapsto e'$ for some e' and e *value*.

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

Canonical Forms. Assume $\cdot \vdash v : \tau$ and v *value*.

(i) If $\tau = \tau_1 \to \tau_2$ then $v = \lambda x.\, e_2$ for some e_2

(ii) If $\tau = \forall \alpha.\, \tau'$ then $v = \Lambda \alpha.\, e'$ for some e'

(iii) If $\tau = \tau_1 \times \tau_2$ then $v = \langle v_1, v_2 \rangle$ for some v_1 *value* and v_2 *value*

(iv) If $\tau = 1$ then $v = \langle\,\rangle$

(v) If $\tau = \tau_1 + \tau_2$ then $v = \mathbf{l} \cdot v_1$ for some v_1 *value* or $v = \mathbf{r} \cdot v_2$ for some v_2 *value*

(vi) If $\tau = 0$ then we have a contradiction

Lecture Notes on Recursive Types

15-814: Types and Programming Languages
Frank Pfenning

Lecture 10
Thursday, October 1, 2020

1 Introduction

Using type structure to capture common constructions available in programming languages, we have built a rich set of primitives in our programming language (see 09-sums-rules.pdf for a summary of the rules). Booleans turned out be representable using generic constructions, since $bool = 1 + 1$. However, natural numbers would be

$$nat = 1 + (1 + (1 + \cdots))$$

which cannot be expressed already. However, we can observe that the tail of the sum is equal to the whole sum. That is,

$$nat = 1 + nat$$

We won't be able to achieve such an equality, but we *can* achieve an *isomorphism*

$$nat \cong 1 + nat$$

with two functions to witness the isomorphism.

$$nat \ \underset{\overleftarrow{\mathsf{fold}}}{\overset{\overrightarrow{\mathsf{unfold}}}{\cong}} \ 1 + nat$$

Actually, unfold and fold will not be functions but language primitives because we want them to apply to a large class of recursively defined types.

2 Recursive Types

The more general type constructor that solves recursive type equations is written as $\rho\alpha.\,\tau$. Rho (ρ) here stands for "recursive", α is a type variable with scope τ. The general picture to keep in mind is that a recursive type $\rho\alpha.\,\tau$ should be isomorphic to its *unfolding* $[\rho\alpha.\,\tau/\alpha]\tau$.

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

Once we have defined the fold and unfold expressions with their statics and dynamics, we will have to check that these two types are indeed isomorphic.

As an example, consider

$$nat = \rho\alpha.\,1 + \alpha$$

Does this give us the desired isomorphism? Let's check:

$$\begin{array}{rcl} nat & = & \rho\alpha.\,1 + \alpha \\ & \cong & [\rho\alpha.\,1 + \alpha/\alpha](1 + \alpha) \\ & = & 1 + (\rho\alpha.\,1 + \alpha) \\ & = & 1 + nat \end{array}$$

So, yes, we get the desired isomorphism. Here are some other examples of types with recursive definitions we'd like to represent in a similar manner.

$$\begin{array}{llcl} \text{Lists} & list\ \tau & \cong & 1 + (\tau \times list\ \tau) \\ \text{Binary Trees} & tree & \cong & 1 + (tree \times nat \times tree) \\ \text{Binary Numbers} & bin & \cong & list\ (1 + 1) \end{array}$$

For example, binary trees of natural numbers would then be explicitly defined as

$$\begin{array}{rcl} tree & = & \rho\alpha.\,1 + (\alpha \times nat \times \alpha) \\ & \cong & 1 + (tree \times nat \times tree) \end{array}$$

and satisfy the desired isomorphism.

3 Fold and Unfold

Let's recall the principal isomorphism we would like to have:

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

Each new type we have comes with some constructors for values of the new type and some destructors. Computation arises when a destructor meets a constructor. According to the display above, fold should be the *constructor* (because it results in something of type $\rho\alpha.\,\tau$), while unfold is a destructor. Reading the types off the above desired isomorphism:

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

We decide that fold e is a value only if e is a value. This is so that, for example, when we write $v : \mathit{nat}$, the value v will actually directly represent a natural number instead of some expression that might result in a natural number (see Exercise 1)

$$\frac{e\ \mathit{value}}{\mathsf{fold}\ e\ \mathit{value}}\ \mathsf{val/fold}$$

The interesting rule for stepping (usually the first one to write) is the one where a destructor meets a constructor.

$$\frac{v\ \mathit{value}}{\mathsf{unfold}\ (\mathsf{fold}\ v) \mapsto v}\ \mathsf{step/unfold/fold}$$

Does this rule preserve types? Let's say we have

$$\cdot \vdash \mathsf{unfold}\ (\mathsf{fold}\ v) : \sigma$$

By inversion (only the unfold rule could have this conclusion), we obtain

$$\cdot \vdash \mathsf{fold}\ v : \rho\alpha.\,\tau$$

where $\sigma = [\rho\alpha.\,\tau/\alpha]\tau$. Applying inversion again, we get

$$\cdot \vdash v : [\rho\alpha.\,\tau/\alpha]\tau$$

which is also the type of unfold (fold v). Therefore, the rule step/unfold satisfies type preservation.

We now only need to add rules to reach values and redices, so-called *congruence rules*.

$$\frac{e \mapsto e'}{\mathsf{fold}\ e \mapsto \mathsf{fold}\ e'}\ \mathsf{step/fold} \qquad \frac{e \mapsto e'}{\mathsf{unfold}\ e \mapsto \mathsf{unfold}\ e'}\ \mathsf{step/unfold}_0$$

It is a matter of checking the progress theorem and also verifying the desired isomorphism to ensure that we now have enough rules. A student suggested

$$\frac{}{\mathsf{fold}\ (\mathsf{unfold}\ e) \mapsto e}\ ?$$

which is eminently reasonable, but turned out to be unnecessary. Instead, we find that fold (unfold e) is extensionally equivalent to e at type $\rho\alpha.\, tau$.

4 Examples

Before we check our desired properties, let's write some examples on natural numbers (in our unary representation).

$$\begin{array}{lcl}
\mathit{nat} & = & \rho\alpha.\, 1 + \alpha \\
 & \cong & 1 + \mathit{nat} \\
\\
\mathit{zero} & : & \mathit{nat} \\
\mathit{zero} & = & \mathsf{fold}\ (\mathbf{l} \cdot \langle\,\rangle) \\
\\
\mathit{one} & : & \mathit{nat} \\
\mathit{one} & = & \mathsf{fold}\ (\mathbf{r} \cdot \mathit{zero}) \\
 & = & \mathsf{fold}\ (\mathbf{r} \cdot \mathsf{fold}\ (\mathbf{l} \cdot \langle\,\rangle)) \\
\\
\mathit{succ} & : & \mathit{nat} \to \mathit{nat} \\
\mathit{succ} & = & \lambda n.\, \mathsf{fold}\ (\mathbf{r} \cdot n) \\
\\
\mathit{pred} & : & \mathit{nat} \to \mathit{nat} \\
\mathit{pred} & = & \lambda n.\, \mathsf{case}\ (\mathsf{unfold}\ n)\ (\mathbf{l} \cdot x_1 \Rightarrow \mathit{zero} \mid r \cdot x_2 \Rightarrow x_2)
\end{array}$$

At this point we realize that we cannot write any function that recurses over a natural number. Unlike the λ-calculus, the representation here as a sum and a recursive types only allows us to implement a case construct. This is not a significant obstacle, since we will shortly add general recursion to our language and then functions like addition, multiplication, exponentiation, and greatest common divisor can be implemented simply and uniformly.

5 Preservation and Progress

We have already seen the key idea in the preservation theorem; all other cases are simple and follow familiar patterns.

For progress, we first need a canonical form theorem. We get the new case

(vi) If $\cdot \vdash v : \rho\alpha.\,\tau$ and v *value* then $v = \mathsf{fold}\ v'$ for a value v'.

This follows, as before, by analyzing the cases for typing and values.

The critical case in the proof of progress (by rule induction on the given typing derivation) is

$$\frac{\cdot \vdash e_1 : \rho\alpha.\,\tau}{\cdot \vdash \mathsf{unfold}\ e_1 : [\rho\alpha.\,\tau/\alpha]\tau}\ \mathsf{tp/unfold}$$

If $e_1 \mapsto e_1'$ then, by rule, $\mathsf{unfold}\ e_1 \mapsto \mathsf{unfold}\ e_1'$. If e_1 is a value, then the canonical forms theorem tells us that $e_1 = \mathsf{fold}\ v_2$ for some value v_2. Therefore, the step/unfold applies and $\mathsf{unfold}\ (\mathsf{fold}\ v_2) \mapsto v_2$.

6 Isorecursive Types

The new type constructor $\rho\alpha.\,\tau$ we have defined is called an *isorecursive type*, because we have and isomorphism

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

rather than an equality between the two types (which would be *equirecursive*). But is it really an isomorphism? Let's check the two directions.

First, we need to check that $\mathsf{unfold}\ (\mathsf{fold}\ v) = v$ for any value $v : [\rho\alpha.\,\tau/\alpha]\tau$. But immediately (by rule step/unfold) we have

$$\mathsf{unfold}\ (\mathsf{fold}\ v) \mapsto v$$

so the two are certainly equal.

In the other direction, we need to verify that

$$\mathsf{fold}\ (\mathsf{unfold}\ v) \stackrel{?}{=} v \qquad \text{for any value } v : \rho\alpha.\,\tau$$

By the canonical forms theorem, $v = \mathsf{fold}\ v'$ for some value v'. Then we reason

$$\begin{array}{ll} & \mathsf{fold}\ (\mathsf{unfold}\ v) \\ = & \mathsf{fold}\ (\mathsf{unfold}\ (\mathsf{fold}\ v')) \\ \mapsto & \mathsf{fold}\ v' \\ = & v \end{array}$$

So, an isorecursive type is indeed isomorphic to its unfolding.

7 Excursion: Embedding the Untyped λ-Calculus

As one of you suspected during lecture, now that we have recursive types, perhaps we can type $\lambda x.\, x\, x$, which we previously proved to have no type. And if that works, why stop there? Why not type the Y combinator itself? In an earlier lecture we convinced ourselves that $\lambda x.\, x\, x : \tau \to \sigma$ for any types τ and τ satisfying $\tau = \tau \to \sigma$. That's because x needs to take itself as an argument.

This does not seem promising, since we still cannot solve this equation! But we may be able to approximate it by an *isomorphism*. Can we find a type U such that $U \cong U \to \tau_2$. The unspecified type τ_2 gets in the way, so let's try it with $\tau_2 = U$. So, we have to solve

$$U \quad \underset{\overset{\longleftarrow}{\mathsf{fold}}}{\overset{\overset{\mathsf{unfold}}{\longrightarrow}}{\cong}} \quad U \to U$$

In our language, any recursive type equation has a solution (perhaps degenerate), so we just set

$$U = \rho\alpha.\, \alpha \to \alpha \cong U \to U$$

Let's try to type self-application at type $U \to U$.

$$\dfrac{\dfrac{?}{x : U \vdash x\, x : U}}{\cdot \vdash \lambda x.\, x\, x : U \to U}\ \mathsf{tp/lam}$$

This still does not work, but we can *unfold* the type of the first occurrence of x so it matches the type of its argument!

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : U \vdash x : U}\ \mathsf{tp/var}}{x : U \vdash \mathsf{unfold}\ x : U \to U}\ \mathsf{tp/unfold} \qquad \dfrac{}{x : U \vdash x : U}\ \mathsf{tp/var}}{x : U \vdash (\mathsf{unfold}\ x)\ x : U}\ \mathsf{tp/app}}{\cdot \vdash \lambda x.\, (\mathsf{unfold}\ x)\ x : U \to U}\ \mathsf{tp/lam}$$

So, lo and behold, if we are willing to insert an unfold we can now type-check self-application.

Curious: can we do the same with the Y combinator? The answer is yes, but let's be even more ambitious: let's translate the whole untyped

λ-calculus into our language! We write M for untyped expressions to distinguish them from the target language expressions e.

$$\text{Untyped Exps} \quad M \quad ::= \quad x \mid \lambda x.\, M \mid M_1\, M_2$$

We try to devise a translation $\ulcorner - \urcorner$ such that

$$\ulcorner M \urcorner : U$$

for any untyped expression M. To be more precise, assume the untyped expression has free variables $x_1, \ldots, x_n$, then we aim for

$$x_1 : U, \ldots, x_n : U \vdash \ulcorner M \urcorner : U$$

The reason all variables have type U because in the source they stand for an arbitrary untyped expression. We define

$$\begin{array}{lcl} \ulcorner x \urcorner & = & x \\ \ulcorner \lambda x.\, M \urcorner & = & \mathsf{fold}\ (\lambda x.\, \ulcorner M \urcorner) \\ \ulcorner M_1\, M_2 \urcorner & = & (\mathsf{unfold}\ \ulcorner M_1 \urcorner)\ \ulcorner M_2 \urcorner \end{array}$$

We suggest you go through these definitions and type-check them, keeping in mind the all-important

$$U \quad \underset{\underset{\mathsf{fold}}{\longleftarrow}}{\overset{\overset{\mathsf{unfold}}{\longrightarrow}}{\cong}} \quad U \to U$$

The type-correctness of this translation means we have a very direct representation of the whole *untyped λ-calculus* in our language, using only a single type U (but exploiting recursive types). Therefore, the untyped λ-calculus is sometimes referred to as the *unityped λ-calculus* because it can be represented with a single universal type U.

Since the Y combinator is only a particular untyped λ-expression, we can also translate it into the target.

However, there is still a fly in the ointment: even though we know the target is well-typed, we don't know if it behaves correctly, operationally. Under some definitions it does not. For example, $\lambda x.\, \Omega$ has no normal form, but $\ulcorner \lambda x.\, \Omega \urcorner = \mathsf{fold}\ (\lambda x.\, \ulcorner \Omega \urcorner)$ is a value and does not take a step. We will discuss at a later point how to bridge this gap, which is not straightforward.

8 Fixed Point Expressions

We have added recursive types that solve recursive type equations. But in order to write all the programs we want (for example, on natural numbers all the recursive functions) we also need recursively defined expressions. The Y combinator is not directly available to us in the needed generality, even though it can be defined at type U. Instead we add a primitive, $\mathsf{fix}\ f.\, e$, where f is a variable. It is not a value, and it steps by *unrolling* the fixed point:

$$\frac{}{\mathsf{fix}\ f.\, e \mapsto [\mathsf{fix}\ f.\, e/f]e}\ \mathsf{step/fix}$$

This "unrolling" is quite similar to unfolding a recursive type, but at the level of expressions. However, it is independent of recursive types and can be applied in full generality. One particular example is $\mathsf{fix}\ f.\, f \mapsto \mathsf{fix}\ f.\, f$ so in this language we can define $\bot = \mathsf{fix}\ f.\, f$ (see Exercise L8.3). Emboldened by this property, we imagine we might have in general

$$\frac{\Gamma, f : \boxed{} \vdash e : \boxed{}}{\Gamma \vdash \mathsf{fix}\ f.\, e : \tau}\ \mathsf{tp/fix}$$

but there are still some holes in this typing rule.

We want preservation to hold (progress is trivial to extend, because a fixed point always steps) so we need that

$$\cdot \vdash \mathsf{fix}\ f.\, e : \tau \text{ implies } \cdot \vdash [\mathsf{fix}\ f.\, e/f]e : \tau$$

From this we can deduce two things: first, $e : \tau$ because that is the result of substitution. And, second, for the substitution property to hold we need that $f : \tau$ so we can substitute $[\mathsf{fix}\ f.\, e/f]e$. Filling in this information:

$$\frac{\Gamma, f : \tau \vdash e : \tau}{\Gamma \vdash \mathsf{fix}\ f.\, e : \tau}\ \mathsf{tp/fix}$$

Now we have settled both statics and dynamic and have fixed point expressions available to us. For example

$$\begin{array}{lcl} \mathit{plus} & : & \mathit{nat} \to (\mathit{nat} \to \mathit{nat}) \\ \mathit{plus} & = & \mathsf{fix}\ p.\, \lambda n.\, \lambda k.\, \mathsf{case}\ (\mathsf{unfold}\ n)\ (\mathbf{l} \cdot _ \Rightarrow k \mid \mathbf{r} \cdot m \Rightarrow \mathit{succ}\ (p\ m\ k)) \end{array}$$

There are a few unpleasant things about fixed point expressions. One is that it is neither a constructor nor a destructor of any particular type, but

is applicable at any type τ. It thus violates one of the design principles of our language that we have followed so far. We may interpret this as an indication that recursion is a fundamental computational principle separate from any particular typing construct, but this is not a universally held view.

The second one is that in $\mathsf{fix}\ f.\, e$ the variable f does not stand for a value (like all other variables x we have used so far) but a expression (we substitute $\mathsf{fix}\ f.\, e$ for f, and that's not a value). To avoid this latter issue, in call-by-value languages sometimes the fixed point expression is limited to functions, as in $\mathsf{fun}\ f(x) = e$ where e can depend on both x and f.

Exercises

Exercise 1 Prove adequacy of natural number encodings in type *nat*.

1. Define a (mathematical) function $\ulcorner n \urcorner$ on natural numbers n such that $\cdot \vdash \ulcorner n \urcorner : \mathit{nat}$ and $\ulcorner n \urcorner\ \mathit{value}$.
2. Define a (mathematical) function $\llcorner v \lrcorner$ on values v with $\cdot \vdash v : \mathit{nat}$ returning the number represented by v.
3. Prove that the pair of functions $\ulcorner - \urcorner$ and $\llcorner - \lrcorner$ witness an isomorphism between the usual (mathematical) natural numbers and closed values of type *nat*.

Exercise 2 Consider the combinators Y and Z. Here Z, the call-by-value fixed point combinator, is defined as

$$Z = \lambda f.\, (\lambda x.\, f\, (\lambda v.\, x\, x\, v))\, (\lambda x.\, f\, (\lambda v.\, x\, x\, v))$$

1. Exhibit a difference between Y and Z under that assumption that the pure untyped λ-calculus follows a call-by-value evaluation strategy.
2. Give the translation $\ulcorner Z \urcorner : U$ into the universal type.

Exercise 3 Consider the type of list of natural numbers

$$\mathit{list} = \rho\alpha.\, (\mathit{nat} \times \alpha) + 1 \cong (\mathit{nat} \times \mathit{list}) + 1$$

Define the following functions

(i) $\mathit{nil} : \mathit{list}$, the empty list.

(ii) $\mathit{cons} : \mathit{nat} \times \mathit{list} \to \mathit{list}$, adding an element to a list.

(iii) *append* : *list* $\rightarrow$ *list* $\rightarrow$ *list*, appending two lists.

(iv) *reverse* : *list* $\rightarrow$ *list*, reversing a list.

(v) *ilist* : *list* $\rightarrow \forall\beta.\,(\mathit{nat} \times \beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta$ satisfying

$$\begin{array}{lcl} \mathit{ilist}\ \mathit{nil}\ [\tau]\ f\ c & = & c \\ \mathit{ilist}\ (\mathit{cons}\ \langle n, l\rangle)\ [\tau]\ f\ c & = & f\ \langle n, \mathit{iter}\ l\ [\tau]\ f\ c\rangle \end{array}$$

where you may take equality to be extensional. This captures *iteration* over lists, for the special case where the elements are all natural numbers. You do not need to prove the correctness of your representation.

(vi) Design a type and implementation for *primitive recursion* over lists, defining a function *plist*.

Abstract Syntax

Types	τ	$::=$	$\tau_1 \to \tau_2 \mid \forall \alpha.\, \tau \mid \alpha \mid \tau_1 \times \tau_2 \mid 1 \mid \tau_1 + \tau_2 \mid 0 \mid \rho\alpha.\, \tau$	
Terms	e	$::=$	$x \mid \lambda x.\, e \mid e_1\, e_2$	$(\to)$
		$\mid$	$\Lambda\alpha.\, e \mid e\, [\tau]$	$(\forall)$
		$\mid$	$\langle e_1, e_2 \rangle \mid \mathsf{case}\ e\ (\langle x_1, x_2 \rangle \Rightarrow e')$	$(\times)$
		$\mid$	$\langle\, \rangle \mid \mathsf{case}\ e\ (\langle\, \rangle \Rightarrow e')$	(1)
		$\mid$	$\mathbf{l} \cdot e \mid \mathbf{r} \cdot e \mid \mathsf{case}\ e\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \mathbf{r} \cdot x_2 \Rightarrow e_2)$	$(+)$
		$\mid$	$\mathsf{case}\ e\ (\,)$	(0)
		$\mid$	$\mathsf{fold}\ e \mid \mathsf{unfold}\ e$	(ρ)
		$\mid$	$\mathsf{fix}\, f.\, e$	
Contexts	Γ	$::=$	$\cdot \mid \Gamma, \alpha\ \mathit{type} \mid \Gamma, x : \tau$	(all variables distinct)

Judgments

$\Gamma\ \mathit{ctx}$	Γ is a valid context	
$\Gamma \vdash \tau\ \mathit{type}$	τ is a valid type	presupposes $\Gamma\ \mathit{ctx}$
$\Gamma \vdash e : \tau$	expression e has type τ	presupposes $\Gamma\ \mathit{ctx}$, ensures $\Gamma \vdash \tau\ \mathit{type}$
$e\ \mathit{value}$	expression e is a value	presupposes $\cdot \vdash e : \tau$ for some τ
$e \mapsto e'$	expression e steps to e'	presupposes $\cdot \vdash e : \tau$ for some τ

Theorems

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

Progress. For every expression $\cdot \vdash e : \tau$ either $e \mapsto e'$ for some e' or e *value*.

Finality of Values. There is no $\cdot \vdash e : \tau$ such that $e \mapsto e'$ for some e' and e *value*.

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

Canonical Forms. Assume $\cdot \vdash v : \tau$ and v *value*.

(i) If $\tau = \tau_1 \to \tau_2$ then $v = \lambda x.\, e_2$ for some e_2

(ii) If $\tau = \forall \alpha.\, \tau'$ then $v = \Lambda\alpha.\, e'$ for some e'

(iii) If $\tau = \tau_1 \times \tau_2$ then $v = \langle v_1, v_2 \rangle$ for some v_1 *value* and v_2 *value*

(iv) If $\tau = 1$ then $v = \langle\, \rangle$

(v) If $\tau = \tau_1 + \tau_2$ then $v = \mathbf{l} \cdot v_1$ for some v_1 *value* or $v = \mathbf{r} \cdot v_2$ for some v_2 *value*

(vi) If $\tau = 0$ then we have a contradiction

(vii) If $\tau = \rho\alpha.\, \tau'$ then $v = \mathsf{fold}\ v'$ for some v' *value*

Contexts Γ

$$\frac{}{(\cdot)\ \mathit{ctx}}\ \mathsf{ctx/emp} \qquad \frac{\Gamma\ \mathit{ctx}}{(\Gamma, \alpha\ \mathit{type})\ \mathit{ctx}}\ \mathsf{ctx/tpvar} \qquad \frac{\Gamma\ \mathit{ctx} \quad \Gamma \vdash \tau\ \mathit{type}}{(\Gamma, x : \tau)\ \mathit{ctx}}\ \mathsf{ctx/var}$$

Functions $\tau_1 \to \tau_2$

$$\frac{\Gamma \vdash \tau_1\ \mathit{type} \quad \Gamma \vdash \tau_2\ \mathit{type}}{\Gamma \vdash \tau_1 \to \tau_2\ \mathit{type}}\ \mathsf{tp/arrow}$$

$$\frac{\Gamma \vdash \tau_1\ \mathit{type} \quad \Gamma, x_1 : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \lambda x_1{:}\tau_1.\, e_2 : \tau_1 \to \tau_2}\ \mathsf{tp/lam} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \mathsf{tp/var} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_1}\ \mathsf{tp/app}$$

$$\frac{}{\lambda x.\, e\ \mathit{value}}\ \mathsf{val/lam} \qquad \frac{e_2\ \mathit{value}}{(\lambda x.\, e_1)\, e_2 \mapsto [e_2/x]e_1}\ \mathsf{step/app/lam}$$

$$\frac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2}\ \mathsf{step/app_1} \qquad \frac{e_1\ \mathit{value} \quad e_2 \mapsto e_2'}{e_1\, e_2 \mapsto e_1\, e_2'}\ \mathsf{step/app_2}$$

Polymorphic Types $\forall \alpha.\, \tau$

$$\frac{\alpha\ \mathit{type} \in \Gamma}{\Gamma \vdash \alpha\ \mathit{type}}\ \mathsf{tp/tpvar} \qquad \frac{\Gamma, \alpha\ \mathit{type} \vdash \tau\ \mathit{type}}{\Gamma \vdash \forall \alpha.\, \tau\ \mathit{type}}\ \mathsf{tp/forall}$$

$$\frac{\Gamma, \alpha\ \mathit{type} \vdash e : \tau}{\Gamma \vdash \Lambda \alpha.\, e : \forall \alpha.\, \tau}\ \mathsf{tp/tplam} \qquad \frac{\Gamma \vdash e : \forall \alpha.\, \tau \quad \Gamma \vdash \sigma\ \mathit{type}}{\Gamma \vdash e\,[\sigma] : [\sigma/\alpha]\tau}\ \mathsf{tp/tpapp}$$

$$\frac{}{\Lambda \alpha.\, e\ \mathit{value}}\ \mathsf{val/tplam} \qquad \frac{}{(\Lambda \alpha.\, e)\,[\tau] \mapsto [\tau/\alpha]e}\ \mathsf{step/tpapp/tplam} \qquad \frac{e \mapsto e'}{e\,[\tau] \mapsto e'\,[\tau]}\ \mathsf{step/tpapp}$$

Pairs $\tau_1 \times \tau_2$

$$\frac{\Gamma \vdash \tau_1 \; \mathit{type} \quad \Gamma \vdash \tau_2 \; \mathit{type}}{\Gamma \vdash \tau_1 \times \tau_2 \; \mathit{type}} \; \mathsf{tp/prod}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \; \mathsf{tp/pair} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2 \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e' : \tau'}{\Gamma \vdash \mathsf{case}\; e \; (\langle x_1, x_2 \rangle \Rightarrow e') : \tau'} \; \mathsf{tp/casep}$$

$$\frac{e_1 \; \mathit{value} \quad e_2 \; \mathit{value}}{\langle e_1, e_2 \rangle \; \mathit{value}} \; \mathsf{val/pair} \qquad \frac{v_1 \; \mathit{value} \quad v_2 \; \mathit{value}}{\mathsf{case}\; \langle v_1, v_2 \rangle \; (\langle x_1, x_2 \rangle \Rightarrow e_3) \mapsto [v_1/x_1][v_2/x_2]e_3} \; \mathsf{step/casep/pair}$$

$$\frac{e_1 \mapsto e_1'}{\langle e_1, e_2 \rangle \mapsto \langle e_1', e_2 \rangle} \; \mathsf{step/pair}_1 \qquad \frac{v_1 \; \mathit{value} \quad e_2 \mapsto e_2'}{\langle v_1, e_2 \rangle \mapsto \langle v_1, e_2' \rangle} \; \mathsf{step/pair}_2$$

$$\frac{e_0 \mapsto e_0'}{\mathsf{case}\; e_0 \; (\langle x_1, x_2 \rangle \Rightarrow e_3) \mapsto \mathsf{case}\; e_0' \; (\langle x_1, x_2 \rangle \Rightarrow e_3)} \; \mathsf{step/casep}_0$$

Unit 1

$$\frac{}{\Gamma \vdash 1 \; \mathit{type}} \; \mathsf{tp/one} \qquad \frac{}{\Gamma \vdash \langle \, \rangle : 1} \; \mathsf{tp/unit} \qquad \frac{\Gamma \vdash e : 1 \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash \mathsf{case}\; e \; (\langle \, \rangle \Rightarrow e') : \tau'} \; \mathsf{tp/caseu}$$

$$\frac{}{\langle \, \rangle \; \mathit{value}} \; \mathsf{val/unit} \qquad \frac{}{\mathsf{case}\; \langle \, \rangle \; (\langle \, \rangle \Rightarrow e) \mapsto e} \; \mathsf{step/caseu/unit}$$

$$\frac{e_0 \mapsto e_0'}{\mathsf{case}\; e_0 \; (\langle \, \rangle \Rightarrow e_1) \mapsto \mathsf{case}\; e_0' \; (\langle \, \rangle \Rightarrow e_1)} \; \mathsf{step/caseu}_0$$

Sums $\tau_1 + \tau_2$

$$\frac{\Gamma \vdash \tau_1\ \mathit{type} \quad \Gamma \vdash \tau_2\ \mathit{type}}{\Gamma \vdash \tau_1 + \tau_2\ \mathit{type}}\ \mathsf{tp/sum}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash \tau_2\ \mathit{type}}{\Gamma \vdash \mathbf{l} \cdot e_1 : \tau_1 + \tau_2}\ \mathsf{tp/left} \qquad \frac{\Gamma \vdash \tau_1\ \mathit{type} \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{r} \cdot e_2 : \tau_1 + \tau_2}\ \mathsf{tp/right}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \sigma \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \sigma}{\Gamma \vdash \mathsf{case}\ e\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \mathbf{r} \cdot x_2 \Rightarrow e_2) : \sigma}\ \mathsf{tp/cases}$$

$$\frac{e_1\ \mathit{value}}{\mathbf{l} \cdot e_1\ \mathit{value}}\ \mathsf{val/left} \qquad \frac{e_2\ \mathit{value}}{\mathbf{r} \cdot e_2\ \mathit{value}}\ \mathsf{val/right}$$

$$\frac{v_1\ \mathit{value}}{\mathsf{case}\ \mathbf{l} \cdot v_1\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \mathbf{r} \cdot x_2 \Rightarrow e_2) \mapsto [v_1/x_1]e_1}\ \mathsf{step/cases/left}$$

$$\frac{v_2\ \mathit{value}}{\mathsf{case}\ \mathbf{r} \cdot v_2\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \mathbf{r} \cdot x_2 \Rightarrow e_2) \mapsto [v_2/x_2]e_2}\ \mathsf{step/cases/right}$$

$$\frac{e_1 \mapsto e_1'}{\mathbf{l} \cdot e_1 \mapsto \mathbf{l} \cdot e_1'}\ \mathsf{step/left} \qquad \frac{e_2 \mapsto e_2'}{\mathbf{r} \cdot e_2 \mapsto \mathbf{r} \cdot e_2'}\ \mathsf{step/right}$$

$$\frac{e_0 \mapsto e_0'}{\mathsf{case}\ e_0\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \mathbf{r} \cdot x_2 \Rightarrow e_2) \mapsto \mathsf{case}\ e_0\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \mathbf{r} \cdot x_2 \Rightarrow e_2)}\ \mathsf{step/cases}_0$$

Empty Type 0

$$\frac{}{\Gamma \vdash 0\ \mathit{type}}\ \mathsf{tp/zero} \qquad \text{(no constructor)} \qquad \frac{\Gamma \vdash e_0 : 0 \quad \Gamma \vdash \tau\ \mathit{type}}{\Gamma \vdash \mathsf{case}\ e_0\ (\,) : \tau}\ \mathsf{tp/casez}$$

$$\text{(no values)} \qquad \frac{e_0 \mapsto e_0'}{\mathsf{case}\ e_0\ (\,) \mapsto \mathsf{case}\ e_0'\ (\,)}\ \mathsf{step/casez}_0$$

Recursive Types $\rho\alpha.\,\tau$

$$\frac{\Gamma, \alpha\ \mathit{type} \vdash \tau\ \mathit{type}}{\Gamma \vdash \rho\alpha.\,\tau\ \mathit{type}}\ \mathsf{tp/rho}$$

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

$$\frac{e\ \mathit{value}}{\mathsf{fold}\ e\ \mathit{value}}\ \mathsf{val/fold} \qquad \frac{v\ \mathit{value}}{\mathsf{unfold}\ (\mathsf{fold}\ v) \mapsto v}\ \mathsf{step/unfold/fold}$$

$$\frac{e \mapsto e'}{\mathsf{fold}\ e \mapsto \mathsf{fold}\ e'}\ \mathsf{step/fold} \qquad \frac{e \mapsto e'}{\mathsf{unfold}\ e \mapsto \mathsf{unfold}\ e'}\ \mathsf{step/unfold}_0$$

Recursion

$$\frac{\Gamma, f : \tau \vdash e : \tau}{\Gamma \vdash \mathsf{fix}\ f{:}\tau.\,e : \tau}\ \mathsf{tp/fix} \qquad \frac{}{\mathsf{fix}\ f.\,e \mapsto [\mathsf{fix}\ f.\,e/f]e}\ \mathsf{step/fix}$$

Lecture Notes on Elaboration

15-814: Types and Programming Languages
Frank Pfenning

Lecture 11
Tuesday, October 6, 2020

1 Introduction

We have spent a lot of time analyzing and designing the essence of a programming language, starting from first principles. The focus has been on the *statics* (the type system), the *dynamics* (the rules for how to evaluate programs), and understanding the relationship between them in a mathematically rigorous way.

There is, of course, a lot more to a real programming language. At the "front end" there is the *concrete syntax* according to which the program text is parsed. The result of parsing is either some *abstract syntax* or an error message if the program is not well-formed according to the grammar defining its syntax. At the "back end" there are concerns about how a language might be executed efficiently, or *compiled* to machine language so it can run even faster. In this course we will say little about issues of grammar, concrete syntax, parsers or parser generators, because we want to focus on the deeper semantic issues where we have accumulated a lot of knowledge about language design.

In today's lecture we will look at *elaboration*, which is a translation mediating between specific forms of concrete syntax and internal representation in abstract syntax. Elaborating the program allows us to provide some conveniences that make it easy to write and read concise programs without giving up the sound underlying principles we have learned about in this course so far.

2 An Example: Binary Numbers

Before binary numbers, we introduce the concrete syntax of LAMBDA when applied to call-by-value functional programs (recognized by the `.cbv` extension). We see a few items of concrete syntax. We use `$` for recursion, both at the level of types (to stand for ρ) and at the level of terms (to stand for fix). Instead of writing $\mathbf{l}.e$ and $\mathbf{r}.e$ (with different fonts being unavailable in the ASCII source) we write $'\mathbf{l}\, e$ and $'\mathbf{r}\, e$ (pronounced "tick l" and "tick r"). Finally, we interpose the keyword `of` between the subject of the case expressions and the branches in order to avoid an ambiguous grammar.

We also see the new kind of declaration **eval** $x = e$ which evaluates $e \mapsto^* v$ and defines x to stand for the resulting value v. Remember that this is quite different from the normal form of e, as we have discussed multiple times.

```
1  type nat = $a. 1 + a    % == 1 + nat
2
3  decl zero : nat
4  decl succ : nat -> nat
5  defn zero = fold ('l ())     % ~ fold (l.<>)
6  defn succ = \n. fold ('r n)
7
8  eval two = succ (succ zero)
9
10 decl pred : nat -> nat
11 defn pred = \n. case (unfold n) of ('l _ => zero | 'r m => m)
12
13 eval one = pred two
14
15 decl plus : nat -> nat -> nat
16 defn plus = $plus. \n. \k.
17   case (unfold n)
18     of ('l _ => k | 'r m => succ (plus m k))
19
20 eval three = plus two one
21
22 decl times : nat -> nat -> nat
23 defn times = $times. \n. \k.
24    case (unfold n)
25      of ('l _ => zero | 'r m => plus (times m k) k)
26
27 eval six = times three two
```

Listing 1: Unary natural numbers in call-by-value LAMBDA

The unary representation of numbers is perfect from the foundational point of view, but impractical. In particular, representation of numbers become very large, and operations on them very slow. But we already know a better representation: binary numbers, which are (finite) sequences of bits 0 and 1.

Binary numbers (type *bin*) are generated by three constructors:

1. *e* : *bin* where e represents 0,
2. *b0* : *bin* $\rightarrow$ *bin* where *b0* $\overline{x}$ represents $2x$, and
3. *b1* : *bin* $\rightarrow$ *bin* where *b1* $\overline{x}$ represents $2x + 1$.

This representation means that we see the least significant bit first. For example, $6 = (110)_2$ wwould be represented by *b0* (*b1* (*b1* *e*)). This "little-endian" representation is well-suited for operations on binary numbers; the representation where we write the bits in the order we are used to not so much (consider, for example, the increment function defined below).

The types of the constructors lead us to the recursive equation

$$\begin{array}{rcl} \mathit{bin} & = & \rho\alpha.\, \alpha + (\alpha + 1) \\ & \cong & \mathit{bin} + (\mathit{bin} + 1) \end{array}$$

and the definitions

$$\begin{array}{rcl} \mathit{b0} & = & \lambda x.\, \mathsf{fold}\ (\mathbf{l} \cdot x) \\ \mathit{b1} & = & \lambda x.\, \mathsf{fold}\ (\mathbf{r} \cdot \mathbf{l} \cdot x) \\ e & = & \mathsf{fold}\ (\mathbf{r} \cdot \mathbf{r} \cdot \langle\,\rangle) \end{array}$$

On this representation we can now define the binary increment function *inc*. We would like it to satisfy the specification

$$\begin{array}{l} \mathit{inc}\ (\mathit{b0}\ x) = \mathit{b1} x \\ \mathit{inc}\ (\mathit{b1}\ x) = \mathit{b0}\ (\mathit{inc}\ x) \\ \mathit{inc}\ (e) = \mathit{b1}\ e \end{array}$$

From this we can derive a closed form definition, where we have to be respect that fact that *inc* is defined *recursively*.

```
1  type bin = $a. a + (a + 1)  % == bin + (bin + 1)
2
3  decl b0 : bin -> bin
4  decl b1 : bin -> bin
5  decl e : bin
6
```

```
7  defn b0 = \x. fold ('l x)
8  defn b1 = \x. fold ('r ('l x))
9  defn e = fold ('r ('r ()))
10
11 decl inc : bin -> bin
12 defn inc = $inc. \x. case (unfold x)
13                           of ( 'l y => b1 y
14                              | 'r y => case y
15                                          of ( 'l z => b0 (inc z)
16                                             | 'r z => b1 e
17                                             )
18                              )
19
20 eval _6 = b0 (b1 (b1 e))
21 eval _7 = inc _6
22 eval _8 = inc _7
```

Listing 2: Binary numbers in call-by-value LAMBDA

We see the output of the last three evaluations

```
1 defn _6 = fold 'l fold 'r 'l fold 'r 'l fold 'r 'r ()
2 defn _7 = fold 'r 'l fold 'r 'l fold 'r 'l fold 'r 'r ()
3 defn _8 = fold 'l fold 'l fold 'l fold 'r 'l fold 'r 'r ()
```

which we can recognize as the binary representations of 6, 7, and 8 where `'l` represents a bit 0, `'r 'l` represents a bit 1, and `'r 'r` represents the terminator for the bit sequence.

One step towards a more natural (and readable) representation is to generalize the binary sum to an variadic sums, which we discuss in Section 4

3 Isomorphism Revisited

The representation of binary numbers has a feature which is common in the representation of complex data, but haven't seen so far: there are multiple different representations of the same data. For example, e and $b0\ e$ both represent the number 0, because $2 \times 0 = 0$. In fact, each number has infinitely many representations: we can just add leading zeros to every representation. In Exercise 1 we explore how to remove this ambiguity from the representation of binary numbers, but this is certainly not possible in other examples.

We could try to show that the translation between unary and binary numbers are an isomorphism (which will fail). For that purpose, we define

the following translations:

$$\begin{array}{rcl} \mathit{nat2bin} & : & \mathit{nat} \to \mathit{bin} \\ \mathit{bin2nat} & : & \mathit{bin} \to \mathit{nat} \end{array}$$

These serve as the "strawman" proposal for a pair of functions witnessing an isomorphism. We write them here in the concrete syntax of LAMBDA.

```
1  decl nat2bin : nat -> bin
2  decl bin2nat : bin -> nat
3
4  (*
5   * nat2bin zero = e
6   * nat2bin (succ n) = inc (nat2bin n)
7   *)
8  defn nat2bin = $nat2bin. \n.
9     case (unfold n) of ('l _ => e | 'r m => inc (nat2bin m))
10
11 (*
12  * bin2nat (b0 x) = times two (bin2nat x)
13  * bin2nat (b1 x) = succ (times two (bin2nat x))
14  * bin2nat (e) = zero
15  *)
16 defn bin2nat = $bin2nat. \x.
17    case (unfold x)
18      of ( 'l y => times two (bin2nat y)
19         | 'r y => case y of ( 'l z => succ (times two (bin2nat z))
20                              | 'r z => zero
21                              )
22         )
```

However, these two functions do not form an isomorphism because any alternative form of a binary number with leading zeros, when mapped to a unary number and back will be standardized in the sense that the leading zeros will be erased.

$$\begin{array}{rcl} \mathit{standardize} & : & \mathit{bin} \to \mathit{bin} \\ \mathit{standardize} & = & \mathit{nat2bin} \circ \mathit{bin2nat} \end{array}$$

On the domain of binary numbers the function *standardize* represents a *retract*, mapping any number to its standard form without leading zeros.

While common, ambiguous representations such as *bin* have their dangers. In particular, we *could* define functions that make no sense at all from the numerical standpoint because they behave differently on different representations of the same number! For example, the function *bad* below returns

true for the standard representation and *false* for a nonstandard one, even though these two representation are supposed to be indistinguishable.

```
1 decl bad : bin -> 1 + 1
2 defn bad = \x. case (unfold x) of ('l x => 'r () | 'r y => 'l () )
3
4 eval tt = bad (b0 e)
5 eval ff = bad e
```

It is therefore important, when working on ambiguous representation, to keep in mind and reason about whether functions are correct with respect to different representations of the same elements. In more general type theories this kind of construction with the guarantee that accompanies it is called a *quotient type*.

4 Variadic Sums

Once we know that the sum is associative and commutative with unit 0 we can introduce a more general notation that is useful for practical purposes: rather than just using labels $\mathbf{l}$ and $\mathbf{r}$ for a binary sum, we can allow a *finite set I* of *tags* or *label* (think of them as strings) and write

$$(i_1 : \tau_1) + \cdots + (i_n : \tau_n)$$

where each summand is marked with a distinct label i. We also write this in abstract syntax as

$$\sum_{i \in I} (i : \tau_i)$$

The empty type 0 arises from $I = \{\,\}$ and we might define

$$\begin{array}{lcl}
\mathit{bool} & = & (\mathbf{true} : 1) + (\mathbf{false} : 1) \\
\mathit{option}\ \tau & = & (\mathbf{none} : 1) + (\mathbf{some} : \tau) \\
\mathit{order} & = & (\mathbf{less} : 1) + (\mathbf{equal} : 1) + (\mathbf{greater} : 1) \\
\mathit{nat} & \cong & (\mathbf{zero} : 1) + (\mathbf{succ} : \mathit{nat}) \\
 & = & \rho\alpha.\, (\mathbf{zero} : 1) + (\mathbf{succ} : \alpha) \\
\mathit{list}\ \tau & \cong & (\mathbf{nil} : 1) + (\mathbf{cons} : \tau \times \mathit{list}\ \tau) \\
 & = & \rho\alpha.\, (\mathbf{nil} : 1) + (\mathbf{cons} : \tau \times \alpha) \\
\mathit{bin} & \cong & (\mathbf{b0} : \mathit{bin}) + (\mathbf{b1} : \mathit{bin}) + (\mathbf{e} : 1) \\
 & = & \rho\alpha.\, (\mathbf{b0} : \alpha) + (\mathbf{b1} : \alpha) + (\mathbf{e} : 1)
\end{array}$$

This generalized form of sum also comes with a generalized constructor (allowing any label of a sum) and case expression (requiring a branch for each label of a sum). For example, we might have the following definitions.

$$
\begin{array}{lcl}
\mathit{bin} & = & \rho\alpha.\,(\mathbf{b0} : \alpha) + (\mathbf{b1} : \alpha) + (\mathbf{e} : 1) \\[4pt]
\mathit{b0} & : & \mathit{bin} \rightarrow \mathit{bin} \\
\mathit{b1} & : & \mathit{bin} \rightarrow \mathit{bin} \\
\mathit{e} & : & \mathit{bin} \\[4pt]
\mathit{b0} & = & \lambda x.\, \mathsf{fold}\ (\mathbf{b0} \cdot x) \\
\mathit{b1} & = & \lambda x.\, \mathsf{fold}\ (\mathbf{b1} \cdot \cdot x) \\
\mathit{e} & = & \mathsf{fold}\ (\mathbf{e} \cdot \langle\,\rangle) \\[4pt]
\mathit{inc} & : & \mathit{bin} \rightarrow \mathit{bin} \\
\mathit{inc} & = & \mathsf{fix}\ \mathit{inc}.\, \lambda x.\, \mathsf{case}\ (\mathsf{unfold}\ x) \\
& & \qquad (\ \mathbf{b0} \cdot y \Rightarrow \mathit{b1}\ y \\
& & \qquad \mid \mathbf{b1} \cdot y \Rightarrow \mathit{b0}\ (\mathit{inc}\ y) \\
& & \qquad \mid \mathbf{e} \cdot _ \Rightarrow \mathit{b1}\ \mathit{e})
\end{array}
$$

5 “Syntactic Sugar”

A simple form of elaboration is to eliminate some simple forms of “syntactic sugar” and translate them into an internal form to simplify downstream processing. A good example are the following definitions:

$$
\begin{array}{lcl}
\mathsf{bool} & \triangleq & (\mathbf{true} : 1) + (\mathbf{false} : 1) \\
\mathsf{true} & \triangleq & \mathbf{true} \cdot \langle\,\rangle \\
\mathsf{false} & \triangleq & \mathbf{false} \cdot \langle\,\rangle \\
\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 & \triangleq & \mathsf{case}\ e_1\ (\mathbf{true} \cdot _ \Rightarrow e_2 \mid \mathbf{false} \cdot _ \Rightarrow e_3)
\end{array}
$$

Here, we used another common convention, name we use an underscore (_) in place of a variable name if that variable does not occur in its scope (here, this scope would be e_2 for the first underscore and e_3 for the second. Such a syntactic transformation could take place before or after type checking.

6 Data Constructors and Pattern Matching

As another example, consider the definition of the natural numbers in unary form:

$$\mathit{nat} = \rho\alpha.\,(\mathbf{zero} : 1) + (\mathbf{succ} : \alpha)$$

This is unnecessarily difficult to read because we have to remember that α really is supposed to stands for *nat* on the right hand. Easier to read is

$$nat \cong (\mathbf{zero} : 1) + (\mathbf{succ} : nat)$$

Moreover, the labels may sometimes be a bit awkward to use, so perhaps we could "automatically" define

$$\begin{array}{lcl} zero & : & 1 \to nat \\ zero & = & \lambda u.\, \mathbf{zero} \cdot u \\ succ & : & nat \to nat \\ succ & = & \lambda n.\, \mathbf{succ} \cdot n \end{array}$$

Notice there the difference between the *function succ* (in italics) and the *label* **succ** (in bold). Maybe we could even go further and eliminate the $1 \to nat$ because we already know that $1 \to \tau \cong \tau$, in which case we would obtain

$$\begin{array}{lcl} zero & : & nat \\ zero & = & \mathbf{zero} \cdot \langle\, \rangle \end{array}$$

Finally, it would be nice if we could simplify pattern matching as well. Instead of, for example,

$$\begin{array}{lcl} pred & : & nat \to nat \\ pred & = & \lambda n.\, \mathsf{case}\ (\mathsf{unfold}\ n)\ (\mathbf{zero} \cdot _ \Rightarrow zero \mid \mathbf{succ} \cdot n' \Rightarrow n') \end{array}$$

it would be easier to read and understand if we could write

$$\begin{array}{lcl} pred & : & nat \to nat \\ pred\ zero & = & zero \\ pred\ (succ\ n') & = & n' \end{array}$$

This would somehow only make sense if "*zero*" was understood not only as a constant of type *nat*, but also that it corresponded to a label **zero** with the same name so we can elaborate it into the case of the internal definition of predecessor shown just before. And similarly for *succ* and **succ**.

In fact, modern functional languages such as Haskell, OCaml, or Standard ML provide syntax for data type definitions that provide essentially the above functionality, and more. In ML we would write:

```
datatype nat = Zero | Succ of nat
fun pred Zero = Zero
  | pred (Succ n') = n'
```

In OCaml it might be

```
type nat = Zero | Succ of nat;;
let pred n = match n with
  | Zero -> Zero
  | Succ n' -> n';;
```

And Haskell:

```
data Nat = Zero | Succ Nat

pred :: Nat -> Nat
pred Zero = Zero
pred (Succ n') = n'
```

The type we gave here for `pred` is optional, but it is often helpful to explicitly state the type of a function. We should also keep in mind that the dynamics of `Zero` and `Succ` is different in Haskell because it is a call-by-need ("lazy") language.

We refer to `Zero` and `Succ` as *data constructors*, which means they are simultaneously functions (or constants in the case of `Zero`) to constructs values of a sum, and labels so we can pattern-match against them.

7 Generalizing Sums

Let's recall our language so far:

$$
\begin{array}{llll}
\text{Types} & \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid 1 \mid \tau_1 + \tau_2 \mid 0 \mid \rho\alpha.\,\tau \\
\text{Expressions} & e & ::= & x & \text{(variables)} \\
& & \mid & \lambda x.\, e \mid e_1\, e_2 & (\to) \\
& & \mid & \langle e_1, e_2 \rangle \mid \mathsf{case}\ e\ (\langle x_1, x_2 \rangle \Rightarrow e') & (\times) \\
& & \mid & \langle\,\rangle \mid \mathsf{case}\ e\ (\langle\,\rangle \Rightarrow e') & (1) \\
& & \mid & \mathbf{l} \cdot e \mid \mathbf{r} \cdot e \mid \mathsf{case}\ e\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \mathbf{r} \cdot x_2 \Rightarrow e_2) & (+) \\
& & \mid & \mathsf{case}\ e\ (\,) & (0) \\
& & \mid & \mathsf{fold}\ e \mid \mathsf{unfold}\ e & (\rho) \\
& & \mid & f \mid \mathsf{fix}\ f.\, e & \text{(recursion)}
\end{array}
$$

Except for functions and recursive types, the destructors are of the form $\mathsf{case}\ e\ (\ldots)$. We will now unify these constructs even more, replacing the

primitive unfold e by a new one, case e (fold $x \Rightarrow e'$). We can then define *Unfold* as a function

$$\begin{array}{lcl} \textit{Unfold} & : & \rho\alpha.\,\tau \to [\rho\alpha.\,\tau/\alpha]\tau \\ \textit{Unfold} & \triangleq & \lambda x.\,\mathsf{case}\ x\ (\mathsf{fold}\ x \Rightarrow x) \end{array}$$

See Exercise 3 for more on this restructuring of the language.

Streamlining our language a little bit further, we now officially generalize the sum from binary to n-ary, allowing labels i to be drawn from a finite index set I. The case construct for the sums then has a branch for each $i \in I$. Our previous constructs are a special case, with $\tau_1 + \tau_2 \triangleq \sum_{i\in\{\mathbf{l},\mathbf{r}\}}(i : \tau_i) = (\mathbf{l} : \tau_1) + (\mathbf{r} : \tau_2)$ and $0 \triangleq \sum_{i\in\emptyset}(i : \tau_i)$.

$$\begin{array}{llcll} \text{Types} & \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid 1 \mid \sum_{i\in I}(i : \tau_i) \mid \rho\alpha.\,\tau & \\ \text{Expressions} & e & ::= & x & \text{(variables)} \\ & & \mid & \lambda x.\,e \mid e_1\,e_2 & (\to) \\ & & \mid & \langle e_1, e_2\rangle \mid \mathsf{case}\ e\ (\langle x_1, x_2\rangle \Rightarrow e') & (\times) \\ & & \mid & \langle\,\rangle \mid \mathsf{case}\ e\ (\langle\,\rangle \Rightarrow e') & (1) \\ & & \mid & i \cdot e \mid \mathsf{case}\ e\ (i \cdot x \Rightarrow e')_{i\in I} & (\sum) \\ & & \mid & \mathsf{fold}\ e \mid \mathsf{case}\ e\ (\mathsf{fold}\ x \Rightarrow e') & (\rho) \\ & & \mid & f \mid \mathsf{fix}\ f.\,e & \text{(recursion)} \end{array}$$

Except for functions, all destructors are now case-expressions. Functions are different because values are of the form $\lambda x.\,e$ that we cannot match against because we assumed that they are not observable outcomes of computation.

For sums, we have the following generalized statics and dynamics. Key is that we have to check all branches of a case expressions, and all of them

have the same type τ'.

$$\frac{k \in I \quad \Gamma \vdash e : \tau_k}{\Gamma \vdash k \cdot e : \sum_{i \in I}(i : \tau_i)} \; \mathsf{tp/sum}$$

$$\frac{\Gamma \vdash e : \sum_{i \in I}(i : \tau_i) \quad \Gamma, x_i : \tau_i \vdash e'_i : \tau' \quad (\text{for all } i \in I)}{\Gamma \vdash \mathsf{case}\; e\; (i \cdot x_i \Rightarrow e'_i)_{i \in I} : \tau'} \; \mathsf{tp/cases}$$

$$\frac{e \; \mathit{value}}{i \cdot e \; \mathit{value}} \; \mathsf{val/sum}$$

$$\frac{e \mapsto e'}{i \cdot e \mapsto i \cdot e'} \; \mathsf{step/inject}$$

$$\frac{e_0 \mapsto e'_0}{\mathsf{case}\; e_0\; (i \cdot x_i \Rightarrow e'_i)_{i \in I} \mapsto \mathsf{case}\; e'_0\; (i \cdot x_i \Rightarrow e'_i)_{i \in I}} \; \mathsf{step/cases}_0$$

$$\frac{k \in I \quad v \; \mathit{value}}{\mathsf{case}\; (k \cdot v)\; (i \cdot x_i \Rightarrow e'_i)_{i \in I} \mapsto [v/x_k]e'_k} \; \mathsf{step/cases/inject}$$

8 Nesting Case Expressions

As another example, let's consider a function *half* on natural numbers that is supposed to round down. We write it down in a pattern-matching style.

$$\begin{array}{lcl} \mathit{half} & : & \mathit{nat} \rightarrow \mathit{nat} \\ \mathit{half}\; \mathit{zero} & = & \mathit{zero} \\ \mathit{half}\; (\mathit{succ}\; \mathit{zero}) & = & \mathit{zero} \\ \mathit{half}\; (\mathit{succ}\; (\mathit{succ}\; n'')) & = & \mathit{succ}\; (\mathit{half}\; n'') \end{array}$$

This could be elaborated into two nested case expressions and a use of recursion. To avoid an even deeper nesting of cases, we use *Unfold* as defined in the previous section.

$$\begin{array}{l} \mathit{half} = \mathsf{fix}\; h.\, \lambda n.\, \mathsf{case}\; (\mathit{Unfold}\; n)\; (\, \mathbf{zero} \cdot _ \Rightarrow \mathit{zero} \\ \qquad \mid \mathbf{succ} \cdot n' \Rightarrow \mathsf{case}\; (\mathit{Unfold}\; n')\; (\, \mathbf{zero} \cdot _ \Rightarrow \mathit{zero} \\ \qquad\qquad \mid \mathbf{succ} \cdot n'' \Rightarrow \mathit{succ}\; (h\; n''))) \end{array}$$

Such nested case expressions naturally lead to the question on how to define arbitrarily nested patterns and how they should be typed and evaluated, which we will discuss in the next lecture.

Exercises

Exercise 1 It is often intuitive to define types in a mutually recursive way. As a simple example, consider how to define binary numbers in *standard form*, that is, not allowing leading zeros. We define binary numbers in standard form (*std*) mutually recursively with strictly positive binary numbers (*pos*).

$$\begin{array}{rcl} \mathit{std} & \cong & (\mathbf{e} : 1) + (\mathbf{b0} : \mathit{pos}) + (\mathbf{b1} : \mathit{std}) \\ \mathit{pos} & \cong & (\mathbf{b0} : \mathit{pos}) + (\mathbf{b1} : \mathit{std}) \end{array}$$

(i) Using only *std*, *pos*, and function types formed from them, give all types of *e*, *b0*, and *b1* defined as follows:

$$\begin{array}{rcl} \mathit{b0} & = & \lambda x.\, \mathsf{fold}\ (\mathbf{b0} \cdot x) \\ \mathit{b1} & = & \lambda x.\, \mathsf{fold}\ (\mathbf{b1} \cdot x) \\ \mathit{e} & = & \mathsf{fold}\ (\mathbf{e} \cdot \langle\,\rangle) \end{array}$$

(ii) Define the types *std* and *pos* explicitly in our language using the ρ type former so that the isomorphisms stated above hold.

(iii) Does the function *inc* from Section 4 have type $\mathit{std} \to \mathit{pos}$? You may use all the types for *b0*, *b1* and *e* you derived in part (i). Then either explain where the typing fails or indicate that it has that type. You do not need to write out a typing derivation.

(iv) Write a function $\mathit{pred} : \mathit{pos} \to \mathit{std}$ that returns the predecessor of a strictly positive binary number. You must make sure your function is correctly typed, where again you may use all the types from part (i).

Exercise 2 It is often convenient to define functions by mutual recursion. As a simple example, consider the following two functions on bit strings determining if it has *even or odd parity*.

$$\begin{array}{rcl} \mathit{bin} & \cong & (\mathbf{e} : 1) + (\mathbf{b0} : \mathit{bin}) + (\mathbf{b1} : \mathit{bin}) \\[1ex] \mathit{even} & : & \mathit{bin} \to \mathit{bool} \\ \mathit{odd} & : & \mathit{bin} \to \mathit{bool} \\[1ex] \mathit{even}\ \mathit{e} & = & \mathit{true} \\ \mathit{even}\ (\mathit{b0}\ x) & = & \mathit{even}\ x \\ \mathit{even}\ (\mathit{b1}\ x) & = & \mathit{odd}\ x \\[1ex] \mathit{odd}\ \mathit{e} & = & \mathit{false} \\ \mathit{odd}\ (\mathit{b0}\ x) & = & \mathit{odd}\ x \\ \mathit{odd}\ (\mathit{b1}\ x) & = & \mathit{even}\ x \end{array}$$

(i) Write a function *parity* with a single fixed point constructor and use it to define *even* and *odd*. Also, state the type of your *parity* function explicitly.

(ii) More generally, our simple recipe for implementing a recursively specified function using the fixed point constructor in our call-by-value language goes from the specification

$$\begin{array}{lcl} f & : & \tau_1 \to \tau_2 \\ f\,x & = & h\,f\,x \end{array}$$

to the implementation

$$f \quad = \quad \mathsf{fix}\, g.\, \lambda x.\, h\, g\, x$$

It is easy to misread these, so remember that by our syntactic convention, $h\,f\,x$ stands for $(h\,f)\,x$ and similarly for $h\,g\,x$. Give the type of h and show by calculation that f satisfies the given specification by considering $f\,v$ for an arbitrary value v of type τ_1.

(iii) A more general, *mutually recursive* specification would be

$$\begin{array}{lcl} f & : & \tau_1 \to \tau_2 \\ g & : & \sigma_1 \to \sigma_2 \\ f\,x & = & h_1\,f\,g\,x \\ g\,y & = & h_2\,f\,g\,y \end{array}$$

Give the types of h_1 and h_2.

(iv) Show how to explicitly define f and g in our language from h_1 and h_2 using the fixed point constructor and verify its correctness by calculation as in part (ii). You may use any other types in the language introduced so far (pairs, unit, sums, polymorphic, and recursive types).

Exercise 3 In the language where the primitive unfold has been replaced by pattern matching, we can define the following two functions:

$$\begin{array}{lcl} \mathit{Unfold} & : & \rho\alpha.\,\tau \to [\rho\alpha.\,\tau/\alpha]\tau \\ \mathit{Unfold} & = & \lambda x.\, \mathsf{case}\; x\; (\mathsf{fold}\; x \Rightarrow x) \\[1ex] \mathit{Fold} & : & [\rho\alpha.\,\tau/\alpha]\tau \to \rho\alpha.\,\tau \\ \mathit{Fold} & = & \lambda x.\, \mathsf{fold}\; x \end{array}$$

Prove that *Fold* and *Unfold* are witnessing a type isomorphism.

Lecture Notes on
Pattern Matching

15-814: Types and Programming Languages
Frank Pfenning

Lecture 12
Thursday, October 8, 2020

1 Introduction

As we have seen in the last lecture, tagged sums together with pattern matching allow us to combine the elimination forms for a variety of types, namely products $\tau_1 \times \tau_2$, unit 1, sums $\sum_{i \in I}(i : \tau_i)$, and recursive types $\rho\alpha.\,\tau$. Allowing patterns to be nested allows for shorter, more easily understandable programs, but they eventually lead to more fundamental changes in the language statics and dynamics. In this lecture we will make these changes to illustrate how a still foundational language can start to become closer and closer to a practical functional language.

2 Nested Cases

As a simple example from the last lecture, consider the specification of a function that divides a unary number by two, rounding down.

$$
\begin{array}{lcl}
\mathit{nat} & = & \rho\alpha.\,(\mathbf{zero} : 1) + (\mathbf{succ} : \alpha) \\
\mathit{half} & : & \mathit{nat} \to \mathit{nat} \\
\mathit{half}\ \mathit{zero} & = & \mathit{zero} \\
\mathit{half}\ (\mathit{succ}\ \mathit{zero}) & = & \mathit{zero} \\
\mathit{half}\ (\mathit{succ}\ (\mathit{succ}\ n'')) & = & \mathit{succ}\ (\mathit{half}\ n'')
\end{array}
$$

With nested patterns, we could write this as

$$
\begin{array}{l}
\mathit{half} = \mathsf{fix}\,\mathit{half}.\,\lambda n. \\
\quad \mathsf{case}\ n\ \ (\,\mathsf{fold}\ \mathbf{zero}\cdot\langle\,\rangle \Rightarrow \mathit{zero} \\
\qquad\qquad \mid \mathsf{fold}\ \mathbf{succ}\cdot\mathsf{fold}\ \mathbf{zero}\cdot\langle\,\rangle \Rightarrow \mathit{zero} \\
\qquad\qquad \mid \mathsf{fold}\ \mathbf{succ}\cdot\mathsf{fold}\ \mathbf{succ}\cdot n'' \Rightarrow \mathit{succ}\ (\mathit{half}\ n'')\,)
\end{array}
$$

This is now quite close to the specification, but using fold and tags **zero** and **succ** instead of using the constructor functions *zero* and *succ* we use in the mathematical specification.

This example also shows why we prefer the destructor for recursive types to be a fold pattern rather than the primitive unfold: without it, we would have to interrupt our nested pattern and distinguish cases further after unfolding subterms explicitly. Using it as a pattern constructor is also justified by the fact that it is eager, so it exposes a value underneath that we can match against.

3 General Pattern Matching

Based on these examples, we now unify all the different case expressions into a single one. For this, we need two new categories of syntax: *branches* bs and *patterns* p. Patterns are either variables, or value constructors for one of types (omitting those that are opaque). We elide here fixed points as well as values whose structure should not be observable, namely functions $\lambda x.\, e$ and type function $\Lambda\alpha.\, e$.

$$
\begin{array}{lrcl}
\text{Expressions} & e & ::= & x \mid \langle e_1, e_2\rangle \mid \langle\,\rangle \mid i\cdot e \mid \mathsf{fold}\ e \mid \mathsf{case}\ e\ (bs) \mid \ldots \\
\text{Patterns} & p & ::= & x \mid \langle p_1, p_2\rangle \mid \langle\,\rangle \mid i\cdot p \mid \mathsf{fold}\ p \\
\text{Branches} & bs & ::= & \cdot \mid (p \Rightarrow e \mid bs)
\end{array}
$$

Because we have new forms of expression, there will also be new judgments for typing the constructs. Let's see what these might be by starting with the rule for case expressions.

$$
\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau \triangleright bs : \sigma}{\Gamma \vdash \mathsf{case}\ e\ (bs) : \sigma}\ \mathsf{case}
$$

The new judgment here is

$$
\Gamma \vdash \tau \triangleright bs : \sigma
$$

We read this as

> *Match a case subject of type τ against the branches bs, each of which must have type σ.*

The reason all branches must have the same type is the same as for the conditionals or branching over a sum: we don't know which branch will be taken when the program runs. Furthermore, each pattern in bs should match the type τ. Because there are two alternatives for branches in the syntax, we have two typing rules for branches. The first (tp/bs/alt) checks the first branch and then the remaining ones. The second (tp/bs/none) expresses that once all branches have been checked, there are no further constraints on τ and σ.

$$\frac{\Gamma' \Vdash p : \tau \quad \Gamma, \Gamma' \vdash e : \sigma \quad \Gamma \vdash \tau \triangleright bs : \sigma}{\Gamma \vdash \tau \triangleright (p \Rightarrow e \mid bs) : \sigma}\ \mathsf{tp/bs/alt} \qquad \frac{}{\Gamma \vdash \tau \triangleright (\cdot) : \sigma}\ \mathsf{tp/bs/none}$$

The first rule here uses a new judgment, $\Gamma' \Vdash p : \tau$. This is almost like the judgment $\Gamma \vdash p : \tau$, noting that every pattern is also an expression. However, it is more restrictive in that the variables in Γ' must be *exactly* the variables in p and, moreover, variables in p may be not occur more than once.[1] We define it with the rules below. When we think about the rules in this set, it may be helpful to keep in mind that when type-checking we know the type τ and the pattern p and we try to *generate* the context Γ', assigning a type to each free variable in p (assuming such a Γ' exists; otherwise there will be no derivation for the given p and τ).

$$\frac{}{x : \tau \Vdash x : \tau}\ \mathsf{pat/var} \qquad \frac{\Gamma_1 \Vdash p_1 : \tau_1 \quad \Gamma_2 \Vdash p_2 : \tau_2}{\Gamma_1, \Gamma_2 \Vdash \langle p_1, p_2 \rangle : \tau_1 \times \tau_2}\ \mathsf{pat/pair} \qquad \frac{}{\cdot \Vdash \langle\,\rangle : 1}\ \mathsf{pat/unit}$$

$$\frac{(k \in I) \quad \Gamma \Vdash p : \tau_k}{\Gamma \Vdash k \cdot p : \sum_{i \in I}(i : \tau_i)}\ \mathsf{pat/inject} \qquad \frac{\Gamma \Vdash p : [\rho\alpha.\, \tau/\alpha]\tau}{\Gamma \Vdash \mathsf{fold}\ p : \rho\alpha.\, \tau}\ \mathsf{pat/fold}$$

It is implicit in the rule for pairs that Γ_1 and Γ_2 are disjoint in their variables, which means that patterns may contain neither duplicate variables nor extraneous variables. For example, we have

$$x : \mathit{nat} \Vdash \mathsf{fold}\ \mathbf{succ} \cdot x : \mathit{nat}$$

but we can *not* have

$$x : \mathit{nat}, y : \mathit{nat} \Vdash \mathsf{fold}\ \mathbf{succ} \cdot x : \mathit{nat}$$

because such a judgment would allow e in the branch

$$\mathsf{fold}\ \mathbf{succ} \cdot x \Rightarrow e$$

[1] In lecture, we did not distinguish this judgment, but without this distinction this rule would be ambiguous.

to mention y (which will not be bound when the value is matched against a pattern). Therefore, the judgment $\Gamma \Vdash p : \tau$ must be "tight" in the sense that Γ contains precisely the variables in p.

For an ordinary hypothetical judgment $\Gamma \vdash e : \tau$ we have certain properties, the most important of which is substitution. But we also have *weakening* which means we can always adjoin another hypothesis without changing the validity of the derivation. That is, if $\Gamma \vdash e : \tau$ then also $\Gamma, x : \sigma \vdash e : \tau$ as long as $\Gamma, x : \sigma$ is a well-formed context. However, this is *not* the case for the typing of patterns, as explained above. This is an example of a *linear hypothetical judgment* where all hypotheses must be used exactly once.

This new set of typing rules is still *syntax-directed*, which now includes not only the typing of expressions, but also the typing of branches and patterns.

4 An Example: Equality on Binary Numbers

Before formalizing the operational semantics, we return to the binary numbers and write a function to increment and then a second one to test equality. We use here the concrete syntax of LAMBDA. Tags are preceded by a tick mark to distinguish them syntactically from variables. They generalize `'l` and `'r` from the binary sums. We begin with the "boilerplate" code, defining a recursive type and then the constructors with their types. In a slight deviation from the previous encoding, we say that $e : 1 \rightarrow \mathit{bin}$ so that each constructor, uniformly, takes the type of the correspondingly tagged value as an argument.

```
1  type bin = $a. ('b0 : a) + ('b1 : a) + ('e : 1)
2
3  decl b0 : bin -> bin
4  decl b1 : bin -> bin
5  decl e : 1 -> bin
6
7  defn b0 = \x. fold 'b0 x
8  defn b1 = \x. fold 'b1 x
9  defn e = \u. fold 'e u
```

Incrementing a number is now straightforward, using a simple pattern match based on the three possible tags of a value of type *bin*.

```
1 decl inc : bin -> bin
2 defn inc = $inc. \x.
3    case x of ( fold 'b0 y => b1 y
4              | fold 'b1 y => b0 (inc y)
5              | fold 'e _ => b1 (e ()) )
```

A first cut in the implementation of equality uses pattern matching against a pair of binary numbers.

```
1 type bool = ('true : 1) + ('false : 1)
2 decl true : bool
3 decl false : bool
4 defn true = 'true ()
5 defn false = 'false ()
6
7 % warning: this implementation is incorrect!
8 decl eq : bin -> bin -> bool
9 defn eq = $eq. \x. \y.
10    case (x,y) of ( (fold 'b0 u, fold 'b0 w) => eq u w
11                  | (fold 'b1 u, fold 'b1 w) => eq u w
12                  | (fold 'e  _, fold 'e  _) => true )
```

This implementation is clearly incorrect, because we somehow have to say "*when none of the other patterns match, return false*". But this is easy, either using variables that don't occur or the special wildcard which syntactically reminds us of this fact.

```
1 % warning: this implementation is still incorrect!
2 decl eq : bin -> bin -> bool
3 defn eq = $eq. \x. \y.
4   case (x,y) of ( (fold 'b0 u, fold 'b0 w) => eq u w
5                 | (fold 'b1 u, fold 'b1 w) => eq u w
6                 | (fold 'e  _, fold 'e  _) => true
7                 | (_, _) => false )
```

In fact, we could also use a single _ to match against the pair in this last catch-all case.

This implementation, however, is still incorrect. See if you can spot and fix the bug before moving on to the next page.

The problem is that the representation allows leading zeros, so that *eq* $(e\ \langle\rangle)$ $(b0\ (e\ \langle\rangle))$ should return *true* but the current implementation returns *false*.

We could fix the problem by *standardizing* the numbers and then using equality only on numbers in standard form. It is easy to make a mistake there, however, unless you also track standard forms in the types, as suggested in Exercise L11.1.

Alternatively, we can change the definition to account for leading zeros by stripping them away during the comparisons against zero (represented here by the pattern fold $\mathbf{e} \cdot _$).

```
1 % relax: this implementation is now correct!
2 decl eq : bin -> bin -> bool
3 defn eq = $eq. \x. \y.
4   case (x,y) of ( (fold 'b0 u, fold 'b0 w) => eq u w
5                 | (fold 'b1 u, fold 'b1 w) => eq u w
6                 | (fold 'e  _, fold 'b0 w) => eq x w
7                 | (fold 'b0 u, fold 'e  _) => eq u y
8                 | (fold 'e  _, fold 'e  _) => true
9                 | _                        => false )
```

This definition now works correctly. Note that in the two new branches that strip tags `'b0'` we exploit the fact that we know the one of the elements of the pair are actually zero and are bound to a variable already (the function arguments x and y, respectively). We could write the slightly more verbose `eq (e ()) w` and `eq u (e ())` instead.

5 Dynamics of Pattern Matching

The dynamics now also has to deal with pattern matching, and up to a certain point it seems less complicated. When we match a value v against a pattern p, this match either has to fail or return to us a substitution η for all the variables in p. We write this as either $v = [\eta]p$ or "*there is no* η *with* $v = [\eta]p$". This η is a simultaneous substitution for all the variables in p which we write as $(v_1/x_1, \ldots, v_n/x_n)$. Matching proceeds sequentially through the patterns. If it reaches the end of the branches and no case has matched, it transitions to raising a Match exception, which is a new possible

outcome of a computation.

$$\frac{e_0 \mapsto e_0'}{\mathsf{case}\ e_0\ (bs) \mapsto \mathsf{case}\ e_0'\ (bs)}\ \mathsf{step/case}_0$$

$$\frac{v\ \mathit{value} \quad v = [\eta]p}{\mathsf{case}\ v\ (p \Rightarrow e \mid bs) \mapsto [\eta]e}\ \mathsf{step/case/match}$$

$$\frac{v\ \mathit{value} \quad \text{there is no } \eta \text{ with } v = [\eta]p \quad \mathsf{case}\ v\ (bs) \mapsto e'}{\mathsf{case}\ v\ (p \Rightarrow e \mid B) \mapsto e'}\ \mathsf{step/case/nomatch}$$

$$\frac{v\ \mathit{value}}{\mathsf{case}\ v\ (\cdot) \mapsto \mathsf{raise\ Match}}\ \mathsf{step/case/none}$$

If we allow raise Match to have every possible type, then the preservation theorem still goes through. Furthermore, since it is not a value, the canonical forms theorem will continue to hold. However, the progress theorem now has to change: a closed well-typed expression either can take a step or is a value or raises a match exception.

This may be somewhat unsatisfactory because the slogan "*well-typed programs do not go wrong*" no longer applies in its purest form. However, the progress theorem (once carefully spelled out) still characterizes the possible outcomes of computations exactly.

In order to avoid this unpleasantness, in Standard ML (SML) it is assumed that pattern matches are exhaustive. If the compiler determines that a given set of patterns is not, it adds a catch-all final branch at the end. However, this branch reads "_ $\Rightarrow$ raise Match" (exploiting the presence of exceptions in SML) which is therefore no different from the semantics we gave above.

We will complete the discussion of pattern matching and exceptions in the next lecture.

Exercises

Exercise 1 In this exercise we explore how to make the rules for pattern matching slightly less abstract.

(i) Define $v \triangleright p \mapsto [\eta]$, as a three-place judgment using inference rules. This judgment should be derivable exactly if $v = [\eta]p$ (but you do not need to prove that).

(ii) Define $\eta : \Gamma$ as a two-place judgment using inference rules. This judgment should be derivable if η substitutes a value of suitable type for each variable in Γ.

(iii) Prove that if $\eta : \Gamma$ and $\Gamma \Vdash p : \tau$ then $[\eta]p$ *value* and $\cdot \vdash [\eta]p : \tau$.

(iv) Define a new judgment $v \not\triangleright p$ using inference rules. This judgment should be derivable exactly if there is no substitution η such that $v = [\eta]p$.

(v) It should be the case that for any v and p, either $v \triangleright p \mapsto \eta$ or $v \not\triangleright p$. State any additional assumptions you may need (say, on the typing of v or p) and sketch the proof. Include two representative cases.

(vi) Revise the dynamics for pattern matching to use the new judgments devised above.

(vii) Revise the proof of *preservation* to account for general pattern matching. Essentially, remove all the cases for individual destructors and then add in a case for the generalized case construct. The key question is how to use the properties you have developed above. If you need any additional properties of simultaneous substitutions η please state them, but you do not need to prove them.

Exercise 2 In this exercise we explore the restriction that patterns cannot have any repeated variables.

(i) Explain why in our language we cannot reasonably allow patterns with repeated variables. Be as concrete as possible.

(ii) Explore to which extent repeated pattern variables might make sense and revise the judgment $\Gamma \Vdash p : \tau$ accordingly.

Lecture Notes on Exceptions

15-814: Types and Programming Languages
Frank Pfenning

Lecture 13
Tuesday, October 13, 2020

1 Introduction

In the previous lecture we introduced general pattern matching, which naturally led to considering an exception if no branch matched. In this lecture we continue our investigation of exceptions. As always, we consider statics and dynamics and the important theorems showing that they cohere.

2 Preservation for Exceptions

Recall the typing of case-expressions and branches for general pattern matching from the last lecture:

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau \triangleright bs : \sigma}{\Gamma \vdash \mathsf{case}\ e\ (bs) : \sigma}\ \mathsf{case}$$

$$\frac{\Gamma' \Vdash p : \tau \quad \Gamma, \Gamma' \vdash e : \sigma \quad \Gamma \vdash \tau \triangleright bs : \sigma}{\Gamma \vdash \tau \triangleright (p \Rightarrow e \mid bs) : \sigma}\ \mathsf{tp/bs/alt} \qquad \frac{}{\Gamma \vdash \tau \triangleright (\cdot) : \sigma}\ \mathsf{tp/bs/none}$$

A key observation here is that when we reach the empty list of branches (rule tp/bs/none) the type σ can be anything—usually, it is determined from the other branches.

Now recall the dynamics of pattern matching from the last lecture.

$$\frac{e_0 \mapsto e_0'}{\mathsf{case}\ e_0\ (bs) \mapsto \mathsf{case}\ e_0'\ (bs)}\ \mathsf{step/case_0}$$

$$\frac{v\ \mathit{value} \quad v = [\eta]p}{\mathsf{case}\ v\ (p \Rightarrow e \mid bs) \mapsto [\eta]e}\ \mathsf{step/case/match}$$

$$\frac{v\ \mathit{value} \quad \text{there is no } \eta \text{ with } v = [\eta]p \quad \mathsf{case}\ v\ (bs) \mapsto e'}{\mathsf{case}\ v\ (p \Rightarrow e \mid B) \mapsto e'}\ \mathsf{step/case/nomatch}$$

$$\frac{v\ \mathit{value}}{\mathsf{case}\ v\ (\cdot) \mapsto \mathsf{raise\ Match}}\ \mathsf{step/case/none}$$

Here we imagine that we extended the syntax of expressions

Expressions	e	$::=$	$\ldots \mid \mathsf{case}\ e\ (bs) \mid \mathsf{raise}\ E$
Exceptions	E	$::=$	$\mathsf{Match} \mid \ldots$

where there may be other (for now unspecified) exceptions such as DivByZero.

In order to obtain type preservation, we need raise E to have all possible types, because the expression on the left-hand side of the step/case/none rule (namely case v $(\cdot)$) can have any type.

$$\frac{}{\Gamma \vdash \mathsf{raise\ Match} : \tau}\ \mathsf{tp/raise}$$

Type preservation then obviously holds for the only rule so far that involves raising an exception. We just need to make sure that as we explore the dynamics of raising an exceptions preservation continues to hold.

3 Progress for Exceptions

We are aiming at the following version of the progress theorem.

Theorem 1 (Progress with Exceptions, v1) *If $\cdot \vdash e : \tau$ then*

(i) either $e \mapsto e'$ for some e',

(ii) or e val,

(iii) or $e = \mathsf{raise}\ E$ *for an exception* E.

Trying to prove this will uncover the fact that, currently, this theorem is false for our language. Consider, as a simple example, $\langle \mathsf{raise}\ \mathsf{Match}, \langle\,\rangle\rangle$. This has type $\tau \times 1$ for any τ, and yet it is stuck: it can not transition, it is not a value, and it is not of the form $\mathsf{raise}\ E$. To remedy this shortcoming, we need to add rules to the dynamics to propagate an exception to the top level. This is awkward, because we need to do it for every kind of expression we already have! This is a shortcoming of this particular style of defining the dynamics of our language, compounded by the fact that exceptions are a control construct, in some sense unrelated to our type structure.

We only show the rules related to pairs.

$$\frac{}{\langle \mathsf{raise}\ E, e\rangle \mapsto \mathsf{raise}\ E}\ \mathsf{step/pair/raise}_1 \qquad \frac{v\ \mathit{value}}{\langle v, \mathsf{raise}\ E\rangle \mapsto \mathsf{raise}\ E}\ \mathsf{step/pair/raise}_2$$

$$\frac{}{\mathsf{case}\ (\mathsf{raise}\ E)\ B \mapsto \mathsf{raise}\ E}\ \mathsf{step/case/raise}$$

It is insignificant here whether we have general pattern matching, or pattern matching specialized to pairs as in earlier versions of our language.

Now we can prove the progress theorem as usual.

Proof: (Progress with Exceptions, Theorem 1) By rule on induction on the derivation of $\cdot \vdash e : \tau$. In comparison with earlier proofs, when we apply the induction hypothesis we obtain three cases to distinguish. In case a subexpression raises an exception, the expression does as well (as long as it is not a value) because we have added enough rules to propagate exception to the top level. ☐

4 Catching Exceptions

Most languages allow programs not only to raise exceptions but also to catch them. Let's consider the simplest such construct, $\mathsf{try}\ e_1\ e_2$. The intention is for it to evaluate e_1 and return its value if that is successful. If it raises an exception, evaluate e_2 instead. This time, we begin with the dynamics.

$$\frac{e_1 \mapsto e_1'}{\mathsf{try}\ e_1\ e_2 \mapsto \mathsf{try}\ e_1'\ e_2}\ \mathsf{step/try}_0 \qquad \frac{v_1\ \mathit{value}}{\mathsf{try}\ v_1\ e_2 \mapsto v_1}\ \mathsf{step/try/value}$$

$$\frac{}{\mathsf{try}\ (\mathsf{raise}\ E)\ e_2 \mapsto e_2}\ \mathsf{step/try/raise}$$

What type do we need to assign to try e_1 e_2 in order to guarantee type preservation. We start with what we know:

$$\frac{\Gamma \vdash e_1 : \boxed{} \quad \Gamma \vdash e_2 : \boxed{}}{\Gamma \vdash \mathsf{try}\ e_1\ e_2 : \boxed{}}\ \mathsf{tp/try}$$

We should be able to "try" an expression of arbitrary type τ, so

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \boxed{}}{\Gamma \vdash \mathsf{try}\ e_1\ e_2 : \boxed{}}\ \mathsf{tp/try}$$

Because of the rule step/try/value, the type of the overall expression needs to be equal to τ as well.

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \boxed{}}{\Gamma \vdash \mathsf{try}\ e_1\ e_2 : \tau}\ \mathsf{tp/try}$$

Finally, in case e_1 fails we step to e_2, so we also must have $e_2 : \tau$.

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{try}\ e_1\ e_2 : \tau}\ \mathsf{tp/try}$$

One issue here is that in e_2 we cannot tell which exception may have been raised, even if we may want to take different actions for different exceptions. That is, we would like to be able to *match* against different exceptions. The generalizations do not introduce any new ideas, so we leave it to Exercise 1 to work out the details.

Exceptions in this lecture and Exercise 1 are not first class, which means that exceptions are not values. This in turn means that functions cannot take exceptions as arguments or return them. If we want exceptions to carry values (for example, error messages) then either exceptions and expression will be mutually recursive syntactic classes, or we lift exceptions and make them first class. The merits of this approach are debatable, but its formalization is not much more difficult than what we have already done (see [Har16, Chapter 29]).

Exercises

Exercise 1 We would like to generalize the try construct to so it can branch on the exception that may have been raised. So we have

Expressions	e	$::=$	$\ldots \mid \mathsf{raise}\ E \mid \mathsf{try}\ e\ (ms)$
Exceptions	E	$::=$	$\mathsf{Match} \mid \mathsf{DivByZero} \mid \ldots$
Exception Handlers	ms	$::=$	$\cdot \mid (E \Rightarrow e \mid ms)$

Note that exception handlers are not already covered by regular pattern matching, because exceptions are neither values nor patterns.

1. Write out typing rules for the generalized try construct and exception handlers.
2. Write out the dynamics for the new constructs. Exception handlers should be tried in order.

You do not have to prove preservation or progress, but you should make sure your rules posses these properties (when taken together with the language we have developed in the course so far).

Exercise 2 We would like to generalize exceptions further so they can be returned by functions, passed as arguments, and dynamically created. For this purpose we create a new type exn. We think of the type exn as a disjoint sum

$$\mathsf{exn} = (\mathbf{Match} : 1) + (\mathbf{DivByZero} : 1) + \cdots$$

except that we can add new alternatives to the sum with a declaration

$$\mathsf{exn} = \mathsf{exn} + (i : \tau)$$

For example, in the absence of strings in the language, we could number different error exceptions instead of relying on the general Match by including

$$\mathsf{exn} = \mathsf{exn} + (\mathbf{Error} : \mathit{nat})$$

and then let the programmer raise Error k to indicate error number k.

A key property to keep in mind that $e : \mathit{exn}$ is an ordinary expression (and its value can be passed around) and raise e requires e to be an exception that can be raised, changing the control flow. Generalized pattern matching should now work to match against exceptions, as they are ordinary values.

Formally develop such a language extension, including the abstract syntax of the new constructs, statics, dynamics, precise statement and key cases in the proofs of preservation and progress.

References

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

Lecture Notes on
The K Machine

15-814: Types and Programming Languages
Frank Pfenning

Lecture 14
Thursday, October 15, 2020

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. The standard theorems to validate the statics and dynamics are progress and preservation, relying also on canonical forms. We have also seen the generic principles such as recursion and exceptions can be integrated into our language elegantly, with the necessary modifications of the progress theorem. 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 briefly 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 our own language. This is an example of a so-called *metacircular interpreter*.

2 Introducing the K Machine

Let's review the dynamics of functions.

$$\frac{}{\lambda x.\, e \; \mathit{value}}\ \textsf{val/lam}$$

$$\frac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2}\ \textsf{step/app}_1 \qquad \frac{v_1\ \mathit{value} \quad e_2 \mapsto e_2'}{v_1\, e_2 \mapsto v_1\, e_2'}\ \textsf{step/app}_2$$

$$\frac{}{(\lambda x.\, e_1')\, v_2 \mapsto [v_2/x]e_1'}\ \textsf{step/app/lam}$$

The rules $\textsf{step/app}_1$ and $\textsf{step/app}_2$ 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 $\textsf{step/beta}$ is the "actual" computation step, 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

$k \triangleright e$	evaluate e with continuation k
$k \triangleleft v$	return value v to continuation k

In the second form, we will always have v *value*. 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:

$$\begin{array}{rcl@{\quad}c@{\quad}rcl}
k & \triangleright & \lambda x.\, e & \mapsto & k & \triangleleft & \lambda x.\, e \\
k & \triangleright & e_1\, e_2 & \mapsto & k \circ (_\; e_2) & \triangleright & e_1 \\
k \circ (_\; e_2) & \triangleleft & v_1 & \mapsto & k \circ (v_1\; _) & \triangleright & e_2 \\
k \circ ((\lambda x.\, e_1')\; _) & \triangleleft & v_2 & \mapsto & k & \triangleright & [v_2/x]e_1'
\end{array}$$

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.

$$
\begin{array}{lrcl}
 & \epsilon & \triangleright & ((\lambda x.\, \lambda y.\, x)\, v_1)\, v_2 \\
\mapsto & \epsilon \circ (_\; v_2) & \triangleright & (\lambda x.\, \lambda y.\, x)\, v_1 \\
\mapsto & \epsilon \circ (_\; v_2) \circ (_\; v_1) & \triangleright & \lambda x.\, \lambda y.\, x \\
\mapsto & \epsilon \circ (_\; v_2) \circ (_\; v_1) & \triangleleft & \lambda x.\, \lambda y.\, x \\
\mapsto & \epsilon \circ (_\; v_2) \circ ((\lambda x.\, \lambda y.\, x)\; _) & \triangleright & v_1 \\
\mapsto^* & \epsilon \circ (_\; v_2) \circ ((\lambda x.\, \lambda y.\, x)\; _) & \triangleleft & v_1 \\
\mapsto & \epsilon \circ (_\; v_2) & \triangleright & \lambda y.\, v_1 \\
\mapsto & \epsilon \circ (_\; v_2) & \triangleleft & \lambda y.\, v_1 \\
\mapsto & \epsilon \circ ((\lambda y.\, v_1)\; _) & \triangleright & v_2 \\
\mapsto^* & \epsilon \circ ((\lambda y.\, v_1)\; _) & \triangleleft & v_2 \\
\mapsto & \epsilon & \triangleright & v_1 \\
\mapsto^* & \epsilon & \triangleleft & v_1
\end{array}
$$

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 $\tau_1 \times \tau_2$. In lecture we actually did sums, but the same pattern

emerges for both. Recall the rules:[1]

$$\frac{v_1\ \mathit{value} \quad v_2\ \mathit{value}}{\langle v_1, v_2\rangle\ \mathit{value}}\ \mathsf{val/pair}$$

$$\frac{e_1 \mapsto e_1'}{\langle e_1, e_2\rangle \mapsto \langle e_1', e_2\rangle}\ \mathsf{step/pair}_1 \qquad \frac{v_1\ \mathit{value} \quad e_2 \mapsto e_2'}{\langle v_1, e_2\rangle \mapsto \langle v_1, e_2'\rangle}\ \mathsf{step/pair}_2$$

$$\frac{e_0 \mapsto e_0'}{\mathsf{case}\ e_0\ (\langle x_1, x_2\rangle \Rightarrow e) \mapsto \mathsf{case}\ e_0'\ (\langle x_1, x_2\rangle \Rightarrow e)}\ \mathsf{step/casep}_0$$

$$\frac{v_1\ \mathit{value} \quad v_2\ \mathit{value}}{\mathsf{case}\ \langle v_1, v_2\rangle\ (\langle x_1, x_2\rangle \Rightarrow e) \mapsto [v_1/x_1, v_2/x_2]e}\ \mathsf{step/casep/pair}$$

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.

$$k \triangleright \mathsf{case}\ e_0\ (\langle x_1, x_2\rangle \Rightarrow e) \quad \mapsto \quad k \circ \mathsf{case}\ _\ (\langle x_1, x_2\rangle \Rightarrow e) \triangleright e_0$$

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.

$$k \circ \mathsf{case}\ _\ (\langle x_1, x_2\rangle \Rightarrow e) \triangleleft \langle v_1, v_2\rangle \quad \mapsto \quad k \triangleright [v_1/x_1, v_2/x_2]e$$

[1]We return here to the usual destructors instead of general pattern matching, as a matter of simplicity. See **??** for the more general language.

To summarize:

$$\begin{array}{rcl}
k \triangleright \langle e_1, e_2 \rangle & \mapsto & k \circ \langle _, e_2 \rangle \triangleright e_1 \\
k \circ \langle _, e_2 \rangle \triangleleft v_1 & \mapsto & k \circ \langle v_1, _ \rangle \triangleright e_2 \\
k \circ \langle v_1, _ \rangle \triangleleft v_2 & \mapsto & k \triangleleft \langle v_1, v_2 \rangle \\
k \triangleright \mathsf{case}\ e_0\ (\langle x_1, x_2 \rangle \Rightarrow e) & \mapsto & k \circ \mathsf{case}\ _\ (\langle x_1, x_2 \rangle \Rightarrow e) \triangleright e_0 \\
k \circ \mathsf{case}\ _\ (\langle x_1, x_2 \rangle \Rightarrow e) \triangleleft \langle v_1, v_2 \rangle & \mapsto & k \triangleright [v_1/x_1, v_2/x_2]e
\end{array}$$

$$\begin{array}{llcll}
\text{Continuations} & k & ::= & \epsilon & \\
& & | & k \circ (_\ e_2) \mid k \circ (v_1\ _) & (\to) \\
& & | & k \circ \langle _, e_2 \rangle \mid k \circ \langle v_1, _ \rangle \mid k \circ \mathsf{case}\ _\ (\langle x_1, x_2 \rangle \Rightarrow e) & (\times)
\end{array}$$

6 Typing the K Machine

We postpone a correctness proof for the K machine to the beginning of next lecture. For now, we study the statics of the 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:

$$\frac{k \div \tau_1 \Rightarrow \sigma \quad \cdot \vdash e_2 : \tau_2}{k \circ (_\ e_2) \div (\tau_2 \to \tau_1) \Rightarrow \sigma}$$

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

$$\frac{k \div \tau_1 \Rightarrow \sigma \quad \cdot \vdash v_1 : \tau_2 \to \tau_1 \quad v_1 \; \textit{value}}{k \circ (v_1 \; _) \div \tau_2 \Rightarrow \sigma}$$

Pairs follow a similar pattern and we just show the rules.

$$\frac{k \div (\tau_1 \times \tau_2) \Rightarrow \sigma \quad \cdot \vdash e_2 : \tau_2}{k \circ \langle _, e_2 \rangle \div \tau_1 \Rightarrow \sigma} \qquad \frac{k \div (\tau_1 \times \tau_2) \Rightarrow \sigma \quad \cdot \vdash v_1 : \tau_1 \quad v_1 \; \textit{value}}{k \circ \langle v_1, _ \rangle \div \tau_2 \Rightarrow \sigma}$$

$$\frac{k \div \tau' \Rightarrow \sigma \quad x_1 : \tau_1, x_2 : \tau_2 \vdash e' : \tau'}{k \circ \mathsf{case} \; _ \; (\langle x_1, x_2 \rangle \Rightarrow e') \div (\tau_1 \times \tau_2) \Rightarrow \sigma}$$

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.

7 Implementing the K Machine

We now proceed to implement the K machine for our language within our language, using LAMBDA's concrete syntax. Because we are implementing the language within itself, this is called a *metacircular interpreter*. We need to be careful to distinguish the *metalanguage* in which we write our interpreter from the *object language* in which should be able to execute programs.

As a matter of convenience and readability (but not a matter of essence), we will use varyadic sums and nested pattern matching in the metalanguage, and binary sums and simple pattern matching in the object language.

In these notes we only show the cases for functions $\tau_1 \to \tau_2$ and sums $\tau_1 + \tau_2$ in the object language; the other cases follow the same patterns and pose only minor challenges (see **??**).

The first beautiful idea of the metacircular interpreter is to implement object-language variables by meta-language variables. This means that object-level functions are implemented via meta-level functions, but at different types. As a result, we will not need to implement *substitution*, because applying the meta-level function will have the effect of implementing object-level substitution.

But first the type E of object-level expressions. It is a (recursive) sum type where each constructor and destructor has a separate summand. We start with just functions.

```
1 type E = $E. ('lam : E -> E) + ('app : E * E)
```

There is no case for variables, since they are represented by meta-language variable. For example, using $\ulcorner \cdot \urcorner$ for the representation function for expression in our language, we have

$$
\begin{array}{lcl}
E & = & \rho E.\, (\mathbf{lam} : E \to E) + (\mathbf{app} : E \times E) \\
\ulcorner \lambda x.\, e \urcorner & = & \mathsf{fold}\ \mathbf{lam} \cdot (\lambda x.\, \ulcorner e \urcorner) \\
\ulcorner x \urcorner & = & x \\
\ulcorner e_1\, e_2 \urcorner & = & \mathsf{fold}\ \mathbf{app} \cdot \langle \ulcorner e_1 \urcorner, \ulcorner e_2 \urcorner \rangle
\end{array}
$$

Here are two examples of this representation in our language

```
1 decl I : E
2 defn I = fold 'lam (\x. x)
3
4 decl omega : E
5 defn omega = fold 'lam (\x. fold 'app (x, x))
```

We can now define some "boilerplate" code, namely the meta-level constructor functions. To make them more easily readable, we give them in curried form.

```
1 decl lam : (E -> E) -> E
2 decl app : E -> E -> E
3
4 defn lam = \f. fold 'lam f
5 defn app = \e1. \e2. fold 'app (e1, e2)
```

The next step is to think about the representation of the stack. We represent this as a *meta-level function* from values to values. Since we don't have a separate type of object-level values (at the moment), they are represented as expressions and it is up to us, as the meta-programmer, to ensure that they are only applied the object-level values.

The interpreter is defined by two functions, *eval* and *retn*. The first function *eval* will have the property that

$$k \triangleright e \mapsto^* k \triangleleft v \quad \text{if and only if} \quad \mathit{eval}\, \ulcorner e \urcorner\, \ulcorner k \urcorner \mapsto^* \mathit{retn}\, \ulcorner v \urcorner\, \ulcorner k \urcorner$$

while *retn* $\ulcorner k \urcorner\ \ulcorner v \urcorner$ represents $k \triangleleft v$. This second function is immediate, because returning a value to a continuation simply *applies* the continuation (represented as a function) to the value.

```
1 decl retn_ : E -> (E -> E) -> E  (* k < v  ===  retn v k *)
2 defn retn_ = \v. \k. k v
```

We use here an underscore to complete the name `retn_` because the next function, `eval_` would clash with LAMBDA's **eval** keyword. For uniformity, we use this disambiguation for both functions.

The main function *eval* e k evaluates e and passes the value to k (instead of returning it). Its first case is fairly simple: when the expression e is a λ-abstraction, it is already a value and we return it to the continuation k.

```
1 decl eval_ : E -> (E -> E) -> E  (* k > e  ===  eval e k *)
2 defn eval_ = $eval_. \e. \k.
3     case e of ( fold 'lam _ => retn_ e k
4               | ... )
```

The second case is that of an application $e_1\, e_2$. We first have to evaluate e_1, with a continuation that evaluates e_2 next. That is,

```
1 decl eval_ : E -> (E -> E) -> E  (* k > e  ===  eval e k *)
2 defn eval_ = $eval_. \e. \k.
3     case e of ( fold 'lam _ => retn_ e k
4               | fold 'app (e1, e2) =>
5                    eval_ e1 (\v1. eval_ e2 (\v2. ... )))
```

The rules of the K machine dictated that once we have evaluated both e_1 and e_2 to v_1 and v_2, respectively, then $v_1 = \lambda x.\, e_1'$ and we then need to evaluate $[v_2/x]e_1'$. We accomplish this in two steps: first, we match v_1 against the representation of λ-expression. This exposes the underlying meta-level function f. We then perform the substitution by applying f to v_2.

```
1 decl eval_ : E -> (E -> E) -> E  (* k > e  ===  eval e k *)
2 defn eval_ = $eval_. \e. \k.
3     case e of ( fold 'lam _ => retn_ e k
4               | fold 'app (e1, e2) =>
5                    eval_ e1 (\v1. eval_ e2 (\v2.
6                       case v1 of (fold 'lam f => eval_ (f v2) k))) )
```

However, this is not the final return value. Instead we pass it to the continuation k that expects the value of $e_1\, e_2$ (which will be computed as the value of $f\, v_2$).

At the "top level" the *evaluate* function passes the initial continuation to *eval*, which is $\lambda v.\, v$ corresponding to the empty stack ϵ.

```
1 decl evaluate : E -> E
2 defn evaluate = \e. eval_ e (\v. v)
```

An interesting property of this representation is that to some extent visibility on the object language is inherited from visibility in the metalanguage. For example,

```
1 eval ii = evaluate (app I I)
```

shows us

```
1 % 28 evaluation steps
2 decl ii : E
3 defn ii = fold 'lam ---
```

In other words, we do not see the normal form because none is computed: arbitrary closed λ-expressions are values in both the object language and metalanguage.

We also see that there is a significant overhead in the interpreter: an expression which takes 1 step to reach a normal form in the small-step dynamics takes 28 steps in the metainterpreter. Of course, we imagine the metainterpreter could be compiled, so in the end it may be efficient enough for many purposes.

For binary sums, the same techniques apply. Key is to represent the branches of a case statement as functions from the tagged value to the result.

$$\begin{array}{lcl}
\ulcorner \mathbf{l} \cdot e \urcorner & = & \mathsf{fold}\ \mathbf{left} \cdot \ulcorner e \urcorner \\
\ulcorner \mathbf{r} \cdot e \urcorner & = & \mathsf{fold}\ \mathbf{right} \cdot \ulcorner e \urcorner \\
\ulcorner \mathbf{case}\ e\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \mathbf{r} \cdot x_2 \Rightarrow e_2) \urcorner & = & \mathsf{fold}\ \mathbf{cases} \cdot \langle \ulcorner e \urcorner, \langle \lambda x_1. \ulcorner e_1 \urcorner, \lambda x_2. \ulcorner e_2 \urcorner \rangle \rangle
\end{array}$$

We first show the extension of the type and the constructor functions.

```
1  type E = $E. ('lam : E -> E) + ('app : E * E)
2            + ('left : E) + ('right : E)
3               + ('cases : E * (E -> E) * (E -> E))
4
5  decl left : E -> E
6  decl right : E -> E
7  decl cases : E -> (E -> E) -> (E -> E) -> E
8
9  defn left = \e. fold 'left e
10 defn right = \e. fold 'right e
11 defn cases = \e. \b1. \b2. fold 'cases (e, b1, b2)
```

Below is the completed cade for evaluation for left and right injection into a sum as well as distinguishing cases over sums. Again, we avoid implementing substitutions by exploiting function application in the meta-level, where each branch is represented as a function.

```
decl eval_ : E -> (E -> E) -> E  (* k > e  ===  eval e k *)
defn eval_ = $eval_. \e. \k.
    case e of ( fold 'lam _ => retn_ e k      % b/c \x.e value
              | fold 'app (e1, e2) =>
                  eval_ e1 (\v1. eval_ e2 (\v2.
                     case v1 of (fold 'lam f => eval_ (f v2) k)))
              | fold 'left e => eval_ e (\v. retn_ (left v) k)
              | fold 'right e => eval_ e (\v. retn_ (right v) k)
              | fold 'cases (e,b1,b2) => eval_ e (\v.
                  case v of ( fold 'left v1 => eval_ (b1 v1) k
                            | fold 'right v2 => eval_ (b2 v2) k
                            ))
              )
```

The complete code with the examples can be found in cps-live.cbv.

It is now very easy to change the language semantics. For example, say we wanted to make the object language to be call-by-name. We would modify the line for application from

```
              | fold 'app (e1, e2) =>
                  eval_ e1 (\v1. eval_ e2 (\v2.
                     case v1 of (fold 'lam f => eval_ (f v2) k)))
```

to passing e_2 directly to f instead of evaluting it first

```
              | fold 'app (e1, e2) =>
                  eval_ e1 (\v1.
                     case v1 of (fold 'lam f => eval_ (f e2) k))
```

A similar modification can be made if we wanted sums to be lazy as well.

This style of programming is called *continuation-passing style*, which is a perfect match for the K machine. This kind of interpreter is also called a *definitional interpreter* [**?**] since it can be seen as providing a dynamics to the object language.

8 Whither Types?

We have represented all expressions in the object language with the same type E in the metalanguage. This means we can evaluate even expressions which have no type, such as *omega* in our example. To complete our implementation of the object language we should also provide a object-language type-checker in the metalanguage. We may return to this in a future lecture; for now we are content to have implemented the dynamics.

A metalanguage term of type E that is not the representation of a well-typed term in the object language may lead to a runtime exception when

we distinguishes cases among the result of evaluation, which happens in the implementation of every destructor (in our code here, application and case over binary sums). These meta-level case expressions are not exhaustive pattern matches, but assume the represented term (and therefore also its value) are well-typed at the object level. This is an example of an *representation invariant*, and a fairly trick one, and shows that we should not expect in general that all pattern matches be exhaustive.

Exercises

Exercise 1 Extend the K Machine for the following constructs, in each case writing out new continuations as necessary and giving both stepping and typing rules.

1. Constructor and destructor for the unit type 1.
2. Constructor and destructor for the sum type $\sum_{i \in I}(i : \tau_i)$.
3. Constructor and destructor for recursive types $\rho\alpha.\, \tau$.
4. The fixed point expression $\mathsf{fix}\ f.\, e$.
5. Constructor and destructors for lazy pairs $\tau_1 \,\&\, \tau_2$ (see Exercise L8.6).

Exercise 2 Extend the metacircular interpreter of the K machine as designed in **??**. You do not need to show any rules for the machine, but you should write example code to exercise all the new features in your interpreter.

Exercise 3 Extend the K machine for general (nested) pattern matching. Give any possible new machine states explicit, and show both the typing and stepping rules for the machine. As part of this, you will have to deal with exceptions. Consider only the simplest case where exceptions cannot be caught.

Exercise 4 Distinguish a type *V* of values from expressions so that, for example, we never accidentally pass an unevaluated expression to a continuation and that the final answer is also a value. State the types of *eval* and *retn*. How much of the interpreter do you need to rewrite to guarantee this property?

References

[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
Types as Propositions

15-814: Types and Programming Languages
Frank Pfenning

Lecture 15
Tuesday, October 20, 2020

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:

$$\dfrac{\dfrac{A \wedge (B \wedge C)\ \mathit{true}}{B \wedge C\ \mathit{true}} \wedge E_2}{B\ \mathit{true}} \wedge E_1$$

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

$$\begin{array}{ccccc} & & K_1 \;\; \cdots \;\; K_m & & \\ & & \vdots & & \\ J_1 & \cdots & J_i & \cdots & J_n \\ & & \vdots & & \\ & & J & & \end{array}$$

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,

$$\dfrac{\dfrac{A \wedge B \; \mathit{true}}{B \; \mathit{true}} \wedge E_2 \quad \dfrac{A \wedge B \; \mathit{true}}{A \; \mathit{true}} \wedge E_1}{B \wedge A \; \mathit{true}} \wedge I$$

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 $A \supset (B \supset (A \wedge B))$.

$$\cfrac{\cfrac{\cfrac{\cfrac{}{A\ \mathit{true}}\,u \quad \cfrac{}{B\ \mathit{true}}\,w}{A \wedge B\ \mathit{true}}\,\wedge I}{B \supset (A \wedge B)\ \mathit{true}}\,\supset I^w}{A \supset (B \supset (A \wedge B))\ \mathit{true}}\,\supset I^u$$

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*.

$$\frac{A \supset B\ \mathit{true} \quad A\ \mathit{true}}{B\ \mathit{true}}\,\supset E$$

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.

$$
\dfrac{\dfrac{\dfrac{}{A \wedge B\ \mathit{true}}\,u}{B\ \mathit{true}}\,{\wedge}E_2 \qquad \dfrac{\dfrac{}{A \wedge B\ \mathit{true}}\,u}{A\ \mathit{true}}\,{\wedge}E_1}{\dfrac{B \wedge A\ \mathit{true}}{(A \wedge B) \supset (B \wedge A)\ \mathit{true}}\,{\supset}I^u}\,{\wedge}I
$$

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.

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{A \supset (B \land C)\ \mathit{true}}\ u \quad \overline{A\ \mathit{true}}\ w}{B \land C\ \mathit{true}}\ {\supset}E}{B\ \mathit{true}}\ {\land}E_1}{A \supset B\ \mathit{true}}\ {\supset}I^w \quad \dfrac{\dfrac{\dfrac{\overline{A \supset (B \land C)\ \mathit{true}}\ u \quad \overline{A\ \mathit{true}}\ v}{B \land C\ \mathit{true}}\ {\supset}E}{C\ \mathit{true}}\ {\land}E_2}{A \supset C\ \mathit{true}}\ {\supset}I^v}{\dfrac{(A \supset B) \land (A \supset C)\ \mathit{true}}{(A \supset (B \land C)) \supset ((A \supset B) \land (A \supset C))\ \mathit{true}}\ {\supset}I^u}\ {\land}I$$

5 Disjunction and Falsehood

So far we have explained the meaning of conjunction, truth, and implication. The disjunction "A *or* B" (written as $A \lor B$) is more difficult, but does not require any new judgment forms. Disjunction is characterized by two introduction rules: $A \lor B$ is true, if either A or B is true.

$$\dfrac{A\ \mathit{true}}{A \lor B\ \mathit{true}}\ {\lor}I_1 \qquad \dfrac{B\ \mathit{true}}{A \lor B\ \mathit{true}}\ {\lor}I_2$$

Now it would be incorrect to have an elimination rule such as

$$\dfrac{A \lor B\ \mathit{true}}{A\ \mathit{true}}\ {\lor}E_1?$$

because even if we know that $A \lor B$ 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:

$$\dfrac{\dfrac{\overline{\top\ \mathit{true}}\ {\top}I}{A \lor \top\ \mathit{true}}\ {\lor}I_2}{A\ \mathit{true}}\ {\lor}E_1?$$

Thus we take a different approach. If we know that $A \lor B$ 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:

$$\dfrac{A \lor B\ \mathit{true} \quad \begin{array}{c}\overline{A\ \mathit{true}}\ u \\ \vdots \\ C\ \mathit{true}\end{array} \quad \begin{array}{c}\overline{B\ \mathit{true}}\ w \\ \vdots \\ C\ \mathit{true}\end{array}}{C\ \mathit{true}}\ {\lor}E^{u,w}$$

If we know that $A \vee B$ *true* then we also know C *true*, if that follows both in the case where $A \vee B$ *true* because A is true and in the case where $A \vee B$ *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 $A \vee B$ *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 $A\ true$ 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.

$$\begin{array}{c} \vdots \\ (A \vee B) \supset (B \vee A)\ \textit{true} \end{array}$$

We begin with an implication introduction.

$$\dfrac{\begin{array}{c} \dfrac{}{A \vee B\ \textit{true}}\,u \\ \vdots \\ B \vee A\ \textit{true} \end{array}}{(A \vee B) \supset (B \vee A)\ \textit{true}}\ {\supset}I^u$$

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 $A \vee B$! So first we need to distinguish the two cases via the rule of disjunction elimination.

$$\dfrac{\dfrac{\dfrac{}{A \vee B\ \textit{true}}\,u \quad \begin{array}{c} \dfrac{}{A\ \textit{true}}\,v \\ \vdots \\ B \vee A\ \textit{true} \end{array} \quad \begin{array}{c} \dfrac{}{B\ \textit{true}}\,w \\ \vdots \\ B \vee A\ \textit{true} \end{array}}{B \vee A\ \textit{true}}\ {\vee}E^{v,w}}{(A \vee B) \supset (B \vee A)\ \textit{true}}\ {\supset}I^u$$

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.

$$\dfrac{\dfrac{}{A \vee B\ \mathit{true}}\, u \quad \dfrac{\dfrac{}{A\ \mathit{true}}\, v}{B \vee A\ \mathit{true}}\, \vee I_2 \quad \dfrac{\dfrac{}{B\ \mathit{true}}\, w}{B \vee A\ \mathit{true}}\, \vee I_1}{\dfrac{B \vee A\ \mathit{true}}{(A \vee B) \supset (B \vee A)\ \mathit{true}}\, \supset I^u}\, \vee E^{v,w}$$

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 $A \vee (A \supset B)$ 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 **Elimination Rules**

$$\frac{A\ \mathit{true} \quad B\ \mathit{true}}{A \wedge B\ \mathit{true}} \wedge I \qquad \frac{A \wedge B\ \mathit{true}}{A\ \mathit{true}} \wedge E_1 \quad \frac{A \wedge B\ \mathit{true}}{B\ \mathit{true}} \wedge E_2$$

$$\frac{}{\top\ \mathit{true}} \top I \qquad \textit{no } \top E \textit{ rule}$$

$$\frac{\begin{array}{c}\overline{A\ \mathit{true}}\ u \\ \vdots \\ B\ \mathit{true}\end{array}}{A \supset B\ \mathit{true}} \supset I^u \qquad \frac{A \supset B\ \mathit{true} \quad A\ \mathit{true}}{B\ \mathit{true}} \supset E$$

$$\frac{A\ \mathit{true}}{A \vee B\ \mathit{true}} \vee I_1 \quad \frac{B\ \mathit{true}}{A \vee B\ \mathit{true}} \vee I_2 \qquad \frac{A \vee B\ \mathit{true} \quad \begin{array}{c}\overline{A\ \mathit{true}}\ u \\ \vdots \\ C\ \mathit{true}\end{array} \quad \begin{array}{c}\overline{B\ \mathit{true}}\ w \\ \vdots \\ C\ \mathit{true}\end{array}}{C\ \mathit{true}} \vee E^{u,w}$$

$$\textit{no } \bot I \textit{ rule} \qquad \frac{\bot\ \mathit{true}}{C\ \mathit{true}} \bot E$$

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:

$M : A$ $\quad$ 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.

$$\frac{M : A \wedge B}{\mathsf{fst}\, M : A} \wedge E_1 \qquad \frac{M : A \wedge B}{\mathsf{snd}\, M : B} \wedge E_2$$

Hence the conjunction $A \wedge B$ proposition corresponds to the (lazy) product type $A \,\&\, B$. 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 $A \,\&\, B$ requires a program M of type A and a program N of type B (as in $\wedge I$). Given a pair M of type $A \,\&\, B$, its first component of type A can be retrieved by the projection $\mathsf{fst}\ M$ (as in $\wedge E_1$), its second component of type B by the projection $\mathsf{snd}\ M$ (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 lazy product, also written as $\top$. 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*.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge B \; \mathit{true}}\, u}{B \; \mathit{true}} \wedge E_2 \quad \dfrac{\dfrac{}{A \wedge B \; \mathit{true}}\, u}{A \; \mathit{true}} \wedge E_1}{B \wedge A \; \mathit{true}} \wedge I}{(A \wedge B) \supset (B \wedge A) \; \mathit{true}} \supset\! I^u$$

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 $\langle N, M \rangle$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{u : A \wedge B}\,u}{\mathsf{snd}\ u : B}\,\wedge E_2 \quad \dfrac{\dfrac{}{u : A \wedge B}\,u}{\mathsf{fst}\ u : A}\,\wedge E_1}{\langle\!|\mathsf{snd}\ u, \mathsf{fst}\ u|\!\rangle : B \wedge A}\,\wedge I}{(\lambda u.\ \langle\!|\mathsf{snd}\ u, \mathsf{fst}\ u|\!\rangle) : (A \wedge B) \supset (B \wedge A)}\,\supset I^u$$

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.

$$\dfrac{M : A \vee B \qquad \begin{array}{c}\dfrac{}{u : A}\,u \\ \vdots \\ N : C\end{array} \qquad \begin{array}{c}\dfrac{}{w : B}\,w \\ \vdots \\ P : C\end{array}}{\mathsf{case}\ M\ (l \cdot u \Rightarrow N \mid r \cdot w \Rightarrow P) : C}\,\vee E^{u,w}$$

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 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 ( ).

$$\dfrac{M : \bot}{\mathsf{case}\ M\ (\,) : C}\,\bot E$$

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:

$$\lambda u.\, \langle\!|(\lambda w.\, \mathsf{fst}\ (u\, w)), (\lambda v.\, \mathsf{snd}\ (u\, v))|\!\rangle$$

The following deduction provides the evidence:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{u : A \supset (B \wedge C)}\,u \quad \dfrac{}{w : A}\,w}{u\, w : B \wedge C}\,{\supset}E}{\mathsf{fst}\ (u\, w) : B}\,{\wedge}E_1}{\lambda w.\, \mathsf{fst}\ (u\, w) : A \supset B}\,{\supset}I^w \quad \dfrac{\dfrac{\dfrac{\dfrac{}{u : A \supset (B \wedge C)}\,u \quad \dfrac{}{v : A}\,v}{u\, v : B \wedge C}\,{\supset}E}{\mathsf{snd}\ (u\, v) : C}\,{\wedge}E_2}{\lambda v.\, \mathsf{snd}\ (u\, v) : A \supset C}\,{\supset}I^v}{\langle\!|(\lambda w.\, \mathsf{fst}\ (u\, w)), (\lambda v.\, \mathsf{snd}\ (u\, v))|\!\rangle : (A \supset B) \wedge (A \supset C)}\,{\wedge}I}{\lambda u.\, \langle\!|(\lambda w.\, \mathsf{fst}\ (u\, w)), (\lambda v.\, \mathsf{snd}\ (u\, v))|\!\rangle : (A \supset (B \wedge C)) \supset ((A \supset B) \wedge (A \supset C))}\,{\supset}I^u$$

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.

$$\begin{array}{rcl} \mathsf{fst}\,\langle\!|M, N|\!\rangle & \longrightarrow & M \\ \mathsf{snd}\,\langle\!|M, N|\!\rangle & \longrightarrow & N \end{array}$$

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:

$$\dfrac{\dfrac{M : A \quad N : B}{\langle\!|M, N|\!\rangle : A \wedge B}\wedge I}{\mathsf{fst}\,\langle\!|M, N|\!\rangle : A}\wedge E_1 \quad \longrightarrow \quad M : A$$

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:

$$\dfrac{\dfrac{\begin{array}{c}\overline{u : A}\;u \\ \vdots \\ M : B\end{array}}{\lambda u.\, M : A \supset B}\supset I^u \qquad N : A}{(\lambda u.\, M)\, N : B}\supset E \quad \longrightarrow \quad [N/u]M$$

Disjunction. The constructors inject into a sum types; the destructor distinguishes cases. We need to use substitution again.

$$\begin{array}{rcl}\mathsf{case}\; l \cdot M\; (l \cdot u \Rightarrow N \mid r \cdot w \Rightarrow P) & \longrightarrow & [M/u]N \\ \mathsf{case}\; r \cdot M\; (l \cdot u \Rightarrow N \mid r \cdot w \Rightarrow P) & \longrightarrow & [M/w]P\end{array}$$

The analogy with the logical reduction again works, for example:

$$\dfrac{\dfrac{M : A}{l \cdot M : A \vee B}\vee I_1 \qquad \begin{array}{c}\overline{u : A}\;u \\ \vdots \\ N : C\end{array} \qquad \begin{array}{c}\overline{w : B}\;w \\ \vdots \\ P : C\end{array}}{\mathsf{case}\; l \cdot M\; (l \cdot u \Rightarrow N \mid r \cdot w \Rightarrow P) : C}\vee E^{u,w} \quad \longrightarrow \quad [M/u]N$$

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**
$A \wedge B$	$\tau \,\&\, \sigma$
$A \supset B$	$\tau \to \sigma$
$A \vee B$	$\tau + \sigma$
$\top$	$\top$
$\bot$	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

$$\frac{M : A \quad N : B}{\langle\!| M, N |\!\rangle : A \wedge B} \wedge I$$

$$\frac{M : A \wedge B}{\mathsf{fst}\ M : A} \wedge E_1 \qquad \frac{M : A \wedge B}{\mathsf{snd}\ M : B} \wedge E_2$$

$$\frac{}{\langle\!|\ |\!\rangle : \top} \top I$$

no destructor for $\top$

$$\frac{\begin{array}{c}\overline{u : A}\ u \\ \vdots \\ M : B\end{array}}{\lambda u.\, M : A \supset B} \supset I^u \qquad \frac{M : A \supset B \quad N : A}{M\, N : B} \supset E$$

$$\frac{M : A}{M \cdot l : A \vee B} \vee I_1 \qquad \frac{N : B}{N \cdot r : A \vee B} \vee I_2$$

$$\frac{M : A \vee B \quad \begin{array}{c}\overline{u : A}\ u \\ \vdots \\ N : C\end{array} \quad \begin{array}{c}\overline{w : B}\ w \\ \vdots \\ P : C\end{array}}{\mathsf{case}\ M\ (l \cdot u \Rightarrow N \mid r \cdot w \Rightarrow P) : C} \vee E^{u,w}$$

no constructor for $\bot$

$$\frac{M : \bot}{\mathsf{case}\ M\ (\,) : C} \bot E$$

Figure 2: Proof term assignment for natural deduction

$$
\begin{array}{rcl}
\mathsf{fst}\ \langle\!|M, N|\!\rangle & \longrightarrow & M \\
\mathsf{snd}\ \langle\!|M, N|\!\rangle & \longrightarrow & N \\
\text{no reduction for } \langle\!|\ |\!\rangle & & \\
(\lambda u.\, M)\, N & \longrightarrow & [N/u]M \\
\mathsf{case}\ (l \cdot M)\ (l \cdot u \Rightarrow N \mid r \cdot w \Rightarrow P) & \longrightarrow & [M/u]N \\
\mathsf{case}\ (r \cdot M)\ (l \cdot u \Rightarrow N \mid r \cdot w \Rightarrow P) & \longrightarrow & [M/w]P \\
\text{no reduction for } \mathsf{case}\ M\ (\,) & &
\end{array}
$$

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 $\mathbf{fix}\, x.\, e$ 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.

$$\dfrac{\dfrac{\begin{matrix}\mathcal{D}\\ A\ \mathit{true}\end{matrix} \quad \begin{matrix}\mathcal{E}\\ B\ \mathit{true}\end{matrix}}{A \wedge B\ \mathit{true}}\,\wedge I}{A\ \mathit{true}}\,\wedge E_1 \quad \Longrightarrow_R \quad \begin{matrix}\mathcal{D}\\ A\ \mathit{true}\end{matrix}$$

$$\dfrac{\dfrac{\begin{matrix}\mathcal{D}\\ A\ \mathit{true}\end{matrix} \quad \begin{matrix}\mathcal{E}\\ B\ \mathit{true}\end{matrix}}{A \wedge B\ \mathit{true}}\,\wedge I}{B\ \mathit{true}}\,\wedge E_2 \quad \Longrightarrow_R \quad \begin{matrix}\mathcal{E}\\ B\ \mathit{true}\end{matrix}$$

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.

$$\begin{matrix}\mathcal{D}\\ A \wedge B\ \mathit{true}\end{matrix} \quad \Longrightarrow_E \quad \dfrac{\dfrac{\begin{matrix}\mathcal{D}\\ A \wedge B\ \mathit{true}\end{matrix}}{A\ \mathit{true}}\,\wedge E_1 \quad \dfrac{\begin{matrix}\mathcal{D}\\ A \wedge B\ \mathit{true}\end{matrix}}{B\ \mathit{true}}\,\wedge E_2}{A \wedge B\ \mathit{true}}\,\wedge I$$

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
>
> $$\begin{array}{c} \overline{A\ \mathit{true}}\ u \\ \mathcal{E} \\ B\ \mathit{true} \end{array}$$
>
> 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.

$$\dfrac{\dfrac{\begin{array}{c} \overline{A\ \mathit{true}}\ u \\ \mathcal{E} \\ B\ \mathit{true} \end{array}}{A \supset B\ \mathit{true}} \supset I^u \quad \begin{array}{c} \mathcal{D} \\ A\ \mathit{true} \end{array}}{B\ \mathit{true}} \supset E \quad \Longrightarrow_R \quad \begin{array}{c} \dfrac{\mathcal{D}}{A\ \mathit{true}}\ u \\ \mathcal{E} \\ B\ \mathit{true} \end{array}$$

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.

$$\begin{array}{c}\mathcal{D}\\ A \supset B\ \textit{true}\end{array} \quad \Longrightarrow_E \quad \dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ A\supset B\ \textit{true}\end{array} \quad \dfrac{}{A\ \textit{true}}\,u}{B\ \textit{true}}\,{\supset}E}{A\supset B\ \textit{true}}\,{\supset}I^u$$

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.

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ A\ \textit{true}\end{array}}{A\vee B\ \textit{true}}\,{\vee}I_L \quad \begin{array}{c}\dfrac{}{A\ \textit{true}}\,u\\ \mathcal{E}\\ C\ \textit{true}\end{array} \quad \begin{array}{c}\dfrac{}{B\ \textit{true}}\,w\\ \mathcal{F}\\ C\ \textit{true}\end{array}}{C\ \textit{true}}\,{\vee}E^{u,w} \quad \Longrightarrow_R \quad \begin{array}{c}\dfrac{\mathcal{D}}{A\ \textit{true}}\,u\\ \mathcal{E}\\ C\ \textit{true}\end{array}$$

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ B\ \textit{true}\end{array}}{A\vee B\ \textit{true}}\,{\vee}I_R \quad \begin{array}{c}\dfrac{}{A\ \textit{true}}\,u\\ \mathcal{E}\\ C\ \textit{true}\end{array} \quad \begin{array}{c}\dfrac{}{B\ \textit{true}}\,w\\ \mathcal{F}\\ C\ \textit{true}\end{array}}{C\ \textit{true}}\,{\vee}E^{u,w} \quad \Longrightarrow_R \quad \begin{array}{c}\dfrac{\mathcal{D}}{B\ \textit{true}}\,w\\ \mathcal{F}\\ C\ \textit{true}\end{array}$$

An example of a rule that would not be locally sound is

$$\dfrac{A\vee B\ \textit{true}}{A\ \textit{true}}\,{\vee}E_1?$$

and, indeed, we would not be able to reduce

$$\dfrac{\dfrac{B\ \textit{true}}{A\vee B\ \textit{true}}\,{\vee}I_R}{A\ \textit{true}}\,{\vee}E_1?$$

In fact we can now derive a contradiction from no assumption, which means the whole system is incorrect.

$$\dfrac{\dfrac{\dfrac{}{\top\ \textit{true}}\,{\top}I}{\bot\vee\top\ \textit{true}}\,{\vee}I_R}{\bot\ \textit{true}}\,{\vee}E_1?$$

Local completeness of disjunction distinguishes cases on the known $A \vee B$ *true*, using $A \vee B$ *true* as the conclusion.

$$\begin{array}{c} \mathcal{D} \\ A \vee B\ \mathit{true} \end{array} \quad \Longrightarrow_E \quad \dfrac{\begin{array}{c}\mathcal{D}\\ A \vee B\ \mathit{true}\end{array} \quad \dfrac{\overline{A\ \mathit{true}}^{\,u}}{A \vee B\ \mathit{true}}\vee I_L \quad \dfrac{\overline{B\ \mathit{true}}^{\,w}}{A \vee B\ \mathit{true}}\vee I_R}{A \vee B\ \mathit{true}}\vee E^{u,w}$$

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 $A \vee B$ *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.

Exercises

Exercise 1 One proposition is *more general* than another if we can instantiate the propositional variables in the first to obtain the second. For example, $A \supset (B \supset A)$ is more general than $A \supset (\bot \supset A)$ (with $[\bot/B]$), $(C \wedge D) \supset (B \supset (C \wedge D))$ (with $[C \wedge D/A]$, but not more general than $C \supset (D \supset E)$.

For each of the following proof terms, give the most general proposition proved by it. (We are justified in saying "*the most general*" because the most general proposition is unique up to the names of the propositional variables.)

1. $\lambda u.\, \lambda w.\, \lambda k.\, w\,(u\,k)$
2. $\lambda w.\, \langle(\lambda u.\, w\,(\ell \cdot u)), (\lambda k.\, w\,(r \cdot k))\rangle$
3. $\lambda x.\, (\mathsf{fst}\,x)\,(\mathsf{snd}\,x)\,(\mathsf{snd}\,x)$
4. $\lambda x.\, \lambda y.\, \lambda z.\, (x\,z)\,(y\,z)$

Exercise 2 Write out a proof term for each of the following propositions. As you know from this lecture, this is the same as writing a program of the translated type in our program language without the use of fixed points.

1. $(A \wedge (A \supset \bot)) \supset B$
2. $(A \vee (A \supset \bot)) \supset (((A \supset \bot) \supset \bot) \supset A)$

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 Parametricity

15-814: Types and Programming Languages
Frank Pfenning

Lecture 16
Tuesday, October 27, 2020

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 the next 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 to 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 it 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. As an example, let's consider some (unknown) function

$$\cdot \vdash f : \forall\alpha.\,\alpha \to \alpha \to \alpha$$

and assume it *parametric* in its type argument. We have

$$\begin{array}{lcl} f\,[\mathit{bool}] & : & \mathit{bool} \to \mathit{bool} \to \mathit{bool} \\ f\,[\mathit{nat}] & : & \mathit{nat} \to \mathit{nat} \to \mathit{nat} \end{array}$$

Now consider a relation R such that *false* $R\ \overline{0}$ and *true* $R\ \overline{n}$ for $n > 0$. If

$$f\ [bool]\ \mathit{false}\ \mathit{true} \mapsto^* \mathit{false}$$

then it must also be the case that, for example,

$$f\ [nat]\ \overline{0}\ \overline{17} \mapsto^* \overline{0}$$

On the other hand, from the indicated behavior and relation we cannot immediately make a statement about

$$f\ [nat]\ \overline{42}\ \overline{0}$$

But we can pick a different relation! Let *false* $S\ \overline{42}$ and *true* $S\ \overline{0}$ (and no other values are related). From the relation S and parametricity we conclude

$$f\ [nat]\ \overline{42}\ \overline{0} \mapsto^* \overline{42}$$

We can see that parametricity is quite powerful, since we can tell a lot about the behavior of f without knowing its definition

What Reynolds showed is that in a polymorphic λ-calculus with products and Booleans, all expressions are parametric in this sense.

We begin by recalling extensional equality and then a new form of equality based on the idea of parametricity called *logical equality*.

2 Extensional Equality

In Lecture 8 we defined an *extensional equality* between expressions. We repeat it here, with a few additional cases. First, expressions are simply evaluated to values that are then compared with a more specialized relation.

Expressions: $e \approx e' : \tau$ iff $e \mapsto^* v$, $e' \mapsto^* v'$ with v, v' values, and $v \sim v' : \tau$ or both e and e' diverge.

For positive types (eager pairs, sums, unit) we compare the structure of the values (which are observable), while for negative types (functions, lazy pairs) we apply the destructor and then compare the results.

Functions: $v \sim v' : \tau_1 \to \tau_2$ iff for all $v_1 : \tau_1$ we have $v\ v_1 \approx v'\ v_1 : \tau_2$.

Pairs: $v \sim v' : \tau_1 \times \tau_2$ iff $v = \langle v_1, v_2 \rangle$, $v' = \langle v_1', v_2' \rangle$ and $v_1 \sim v_1' : \tau_1$ and $v_2 \sim v_2' : \tau_2$.

Units: $v \sim v' : 1$ iff $v = \langle\,\rangle$ and $v' = \langle\,\rangle$ (which is always the case, by the canonical forms theorem).

Sums: $v \sim v' : \sum_{i \in I}(i : \tau_i)$ iff $v = k \cdot v_k$ and $v' = k \cdot v'_k$ and $v_k \sim v'_k : \tau_k$ for some $k \in I$.

Lazy Pair: $v \sim v' : \tau_1 \,\&\, \tau_2$ iff $\mathsf{fst}\ v \approx \mathsf{fst}\ v' : \tau_1$ and $\mathsf{snd}\ v \approx \mathsf{snd}\ v' : \tau_2$

We didn't state this explicitly, but these can be extended to polymorphic and recursive types, since of recursive types as positive and universal quantification as negative.

Universal Quantification: $v \sim v' : \forall\alpha.\, \tau$ iff for all closed σ we have $v\,[\sigma] \approx v'[\sigma] : [\sigma/\alpha]\tau$.

Recursion: $v \sim v' : \rho\alpha.\, \tau$ iff $v = \mathsf{fold}\ v_1$ and $v' = \mathsf{fold}\ v'_1$ and $v_1 \sim v'_1 : [\rho\alpha.\, \tau/\alpha]\tau$.

These last two cases are different from the earlier ones in that the types do not get smaller, something that will occupy us shortly. Also, it seems at least possible we may get into a chain of reasoning

$$v \sim v' : \forall\alpha.\, \tau \to \tau \quad \text{iff} \quad \ldots \quad \text{iff} \quad v \sim v' : \forall\alpha.\, \tau \to \tau$$

so the equality may somehow not be well-defined.

3 Logical Equality

The notion of extensional equality (and the underlying Kleene equality) are almost sufficient, but it is insufficient when we come to *parametricity*. 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 $\forall\alpha.\, \tau$ we cannot apply e and e' to the exact *same* type σ. Instead, we must apply them to *related* types. This in turn means that the two expressions we are comparing may not have the same type but *related* types. The notion of equality we derive from this is called *logical equality* because it is based on *logical relations* [Sta85], one of the many connections between logic and computation. We write

$$e \approx e' \in [\![\tau]\!]$$

if the expressions e and e' stand in the relation designated by τ. This is a slight abuse of notation because, as we will see, τ can be more than just a

type. Also, we no longer require that e and e' should have type τ. For the reason explained above, they may not have the same type. Furthermore, they may not even be well-typed anymore which allows a richer set of applications for logical equality. We also have a second relation, designated by $[\tau]$ that applies only to values. We write $v \sim v' \in [\tau]$ if the values v and v' are related by $[\tau]$. We define

Expressions: $e \approx e' \in [\![\tau]\!]$ iff $e \mapsto^* v$ and $e' \mapsto^* v'$ and $v \sim v' \in [\tau]$.

We assume here, to keep the development simple, that all expressions terminate. In fact, logical relations can be used to prove exactly that. The clauses for the positive types remain essentially the same as for extensional equality, where we restrict recursive types to be purely positive.

Pairs: $v \sim v' \in [\tau_1 \times \tau_2]$ iff $v = \langle v_1, v_2 \rangle$ and $v' = \langle v_1', v_2' \rangle$ for some v_1, v_2, v_1', v_2' and $v_1 = v_1' \in [\tau_1]$ and $v_2 = v_2' \in [\tau_2]$.

Unit: $v \sim v' \in [1]$ iff $v = \langle\,\rangle = v'$.

Sums: $v \sim v' \in [\sum_{i \in I}(i : \tau_i)]$ iff $v = k \cdot v_k$ and $v' = k \cdot v_k'$ for some k, v_k and v_k' with $v_k = v_k' \in [\tau_k]$.

Recursion: $v \sim v' \in [\rho\alpha^+.\, \tau^+]$ iff $v = \mathsf{fold}\ v_1$ and $v' = \mathsf{fold}\ v_1'$ and $v_1 = v_1' \in [[\rho\alpha^+.\, \tau^+/\alpha^+]\tau^+]$.

To be explicit, we define the purely positive types as

$$\tau^+ ::= \tau_1^+ \times \tau_2^+ \mid 1 \mid \sum_{i \in I}(i : \tau_i^+) \mid \rho\alpha^+.\, \tau^+ \mid \alpha^+$$

Even though the type becomes larger in the last clause, the definition is not circular because the values we are comparing get smaller. In fact, we can prove that $v \sim v' \in [\tau^+]$ iff $v = v'$. So the clauses for positive types are mostly useful if negative types are embedded in them.

The case for lazy pairs mirrors what we had before, using the destructors.

Lazy Pairs: $v \sim v' \in [\tau_1 \& \tau_2]$ iff $\mathsf{fst}\ v \approx \mathsf{fst}\ v' \in [\![\tau_1]\!]$ and $\mathsf{snd}\ v \approx \mathsf{snd}\ v' \in [\![\tau_2]\!]$

In some circumstances we can use an equivalent formulation where we require v and v' to be a lazy pairs of two related expressions.

The definition becomes different when we come to universal quantification, where we need to be careful to (a) avoid circularity in the definition, and (b) capture the idea behind parametricity. We write $R : \sigma \leftrightarrow \sigma'$ for a relation between values $v : \sigma$ and $v' : \sigma'$, and $v\ R\ v'$ if R relates v and v'. In

some situations when we would like to reason about parametricity using logical relations, we may need to put some conditions on R, but here we think of it as an arbitrary relation on values. We then define

Universal Quantification: $v \sim v' \in [\forall\alpha.\,\tau]$ iff for all closed types σ and σ' and relations $R : \sigma \leftrightarrow \sigma'$ we have $v[\sigma] \approx v'[\sigma'] \in [\![[R/\alpha]\tau]\!]$

(R) $v \sim v' \in [R]$ iff $v\ R\ v'$.

The second clause here is a new base case in the definition of $[\tau]$, in addition to the type 1. It is needed because we substitute an arbitrary relation R for the type variable α in the clause for universal quantification. So when we encounter R we just use it to compare v and v'.

We have taken a big conceptual step, because what we write as type τ actually now contains relations instead of type variables, as well as ordinary type constructors.

For functions, we apply them to *related* arguments and check that their results are again *related*.

Functions: $v \sim v' \in [\tau_1 \to \tau_2]$ iff for all $v_1 \sim v_1' \in [\tau_1]$ we have $v\,v_1 \approx v'\,v_1' \in [\![\tau_2]\!]$

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 relation between values, we have to establish properties of e and e'. It is an instructive exercise to check that

$$\Lambda\alpha.\,\lambda x.\,x \sim \Lambda\alpha.\,\lambda x.\,x \in [\forall\alpha.\,\alpha \to \alpha]$$

To check: $\Lambda\alpha.\,\lambda x.\,x \sim \Lambda\alpha.\,\lambda x.\,x \in [\forall\alpha.\,\alpha \to \alpha]$
This holds if $(\Lambda\alpha.\,\lambda x.\,x)\,[\sigma] \approx (\Lambda\alpha.\,\lambda x.\,x)\,\sigma' \in [\![R \to R]\!]$
for arbitrary σ, σ' and $R : \sigma \leftrightarrow \sigma'$
This holds if $\lambda x.\,x \sim \lambda x.\,x \in [R \to R]$
This holds if $(\lambda x.\,x)\,v_1 \approx (\lambda x.\,x)\,v_1' \in [\![R]\!]$ for arbitrary $v_1 \sim v_1' \in [R]$
This holds if $v_1 \sim v_1' \in [R]$, which is true by assumption

There is nothing wrong with this proof, but let's turn this reasoning around and present it in the "forward" direction, just to see it in a different form.

Let σ, σ', $R : \sigma \leftrightarrow \sigma'$ be arbitrary	Assumption
$v_1\ R\ v_1'$ for some arbitrary v_1 and v_1'	Assumption
$v_1 \sim v_1' \in [R]$	By defn. of $\sim$ at $[R]$
$(\lambda x.\,x)\,v_1 \approx (\lambda x.\,x)\,v_1' \in [\![R]\!]$	By defn. of $\approx$ at $[\![R]\!]$

$\lambda x.\, x \sim \lambda x.\, x \in [R \to R]$	By defn. of $\sim$ at $[R \to R]$ since v_1 and v_1' were arbitrary
$(\Lambda\alpha.\, \lambda x.\, x)\,[\sigma] \approx (\Lambda\alpha.\, \lambda x.\, x)\,[\sigma'] \in [\![R \to R]\!]$	By defn. of $\approx$ at $[\![R \to R]\!]$
$\Lambda\alpha.\, \lambda x.\, x \sim \Lambda\alpha.\, \lambda x.\, x$	By defn. of $\sim$ at $[\forall\alpha.\, \alpha \to \alpha]$ since σ, σ', and R were arbitrary

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. This property holds in a language without general recursive types and general fixed point expressions.

Theorem 1 (Parametricity [Rey83]) *If* $\cdot\,;\cdot \vdash e : \tau$ *then* $e \approx e \in [\![\tau]\!]$

We will not go into the proof of this theorem, but just explore its consequences. Besides the original paper, there are a number of proofs in the literature including in the textbook [Har16, Chapter 48] in a language and formalization that's quite similar to ours. We do not go into detail under which conditions it might be restored in the presence of recursive types and fixed point expressions (see, for example, Ahmed [Ahm06]).

4 Some Useful Properties

In a couple of places we may use the following properties, which follow directly from small-step determinism (sequentiality) and the definition of $[\![\tau]\!]$.

(Closure under Expansion) If $e \approx e' \in [\![\tau]\!]$ and $e_0 \mapsto^* e$ and $e_0' \mapsto^* e'$ then $e_0 \approx e_0' \in [\![\tau]\!]$.

(Closure under Reduction) If $e \approx e' \in [\![\tau]\!]$ and $e \mapsto^* e_0$ and $e' \mapsto^* e_0'$ then $e_0 \approx e_0' \in [\![\tau]\!]$.

Also, the call-by-value strategy entails the following properties for reasoning about logical equality.

(Closure under Application) If $e_1 \approx e_1' \in [\![\tau_2 \to \tau_1]\!]$ and $e_2 \approx e_2' \in [\![\tau_2]\!]$ then $e_1\, e_2 \approx e_1'\, e_2' \in [\![\tau_1]\!]$.

(Closure under Type Application) If $e \approx e' \in [\![\forall\alpha.\, \tau]\!]$ and $R : \sigma \leftrightarrow \sigma'$ then $e[\sigma] \approx e'[\sigma'] \in [\![[R/\alpha]\tau]\!]$.

5 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 behaves like the identity function! We express this first by proving

$$f\ [\sigma_0]\ v_0 \mapsto^* v_0 \quad \text{for all types } \sigma_0 \text{ and values } v_0 : \sigma_0$$

Later, we prove this property in a second form, namely that f is logically equivalent to the polymorphic identity.

For simplicity, assume f is a value. By the parametricity theorem, we have

$$f \approx f \in [\![\forall\alpha.\, \alpha \to \alpha]\!]$$

By definition of $[\![-]\!]$ and the fact that f is a value, we obtain

$$f \sim f \in [\forall\alpha.\, \alpha \to \alpha]$$

By definition of $[\forall\alpha.-]$, this entails that

$$f\,[\tau] \approx f\,[\tau'] \in [\![R \to R]\!]$$

for any τ, τ', and $R : \tau \leftrightarrow \tau'$. In view of the property we want to show, we pick $\tau = \tau' = \sigma_0$ and R_0 such that $v_0\ R_0\ v_0$. That is, $R_0 : \sigma_0 \leftrightarrow \sigma_0$ relates only v_0 to itself and not any other values. This mean we have

$$f\,[\sigma_0] \approx f\,[\sigma_0] \in [\![R_0 \to R_0]\!]$$

Next, by definition of $[\![-]\!]$ we find $f\,[\sigma_0] \mapsto^* f_{\sigma_0}$ for some value f_{σ_0} and

$$f_{\sigma_0} \sim f_{\sigma_0} \in [R_0 \to R_0]$$

By definition of $[_ \to _]$ this means that for any value v such that $v \sim v \in [R_0]$ we have $f_{\sigma_0}\, v \approx f_{\sigma_0}\, v \in [R_0]$. We pick $v = v_0$ because $v_0\ R_0\ v_0$ and consequently also

$$v_0 \sim v_0 \in [R_0]$$

Therefore we conclude

$$f_{\sigma_0}\, v_0 \approx f_{\sigma_0}\, v_0 \in [\![R_0]\!]$$

Again, by definition of $[\![-]\!]$ we know that $f_{\sigma_0}\, v_0 \mapsto^* w$ for some w with

$$w \sim w \in [R_0]$$

which in turn is the case if and only if

$$w \; R_0 \; w$$

by the definition of $[R_0]$. But R_0 was chosen so it relates only v_0 to v_0, so we conclude that

$$w = v_0$$

Unwinding the chain of evaluations under our call-by-value strategy we find

$$f\ [\sigma_0]\ v_0 \mapsto^* f_{\sigma_0}\ v_0 \mapsto^* w = v_0$$

and our theorem is proved.

Out next goal is to show that any value $f : \forall\alpha.\, \alpha \to \alpha$ is (logically) equivalent to the identity function

$$f \sim \Lambda\alpha.\, \lambda x.\, x \in [\forall\alpha.\, \alpha \to \alpha]$$

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 σ and σ' and relation $R : \sigma \leftrightarrow \sigma'$, we have $f\,[\sigma] \approx (\Lambda\alpha.\, \lambda x.\, x)\,[\sigma'] \in [\![R \to R]\!]$*

Now fix arbitrary σ, σ' and R. Because $(\Lambda\alpha.\, \lambda x.\, x)\,[\sigma'] \mapsto \lambda x.\, x$, our desired conclusion holds if $f\,[\sigma] \mapsto^* f_\sigma$ for some value f_σ and

$$f_\sigma \sim \lambda x.\, x \in [R \to R]$$

Applying the definition of $[_ \to _]$, this is true if

> *For all $v \sim v' \in [R]$ we have $f_\sigma\, v \approx (\lambda x.\, x)\, v' \in [\![R]\!]$*

So assume $v \sim v' \in [R]$. It remains to show that

> *$f_\sigma\, v \mapsto^* w$ for some w with $w \sim v' \in [R]$.*

By the previous argument (starting from the parametricity of f) we know that $f_\sigma\, v \mapsto^* v$, so determinism gives us $w = v$. Then $w\ R\ v'$ follows from $v\ R\ v'$ and $w = v$.

Let's summarize the reasoning.

To show:	$f \sim \Lambda\alpha.\, \lambda x.\, x \in [\forall\alpha.\, \alpha \to \alpha]$
True, if	$f\,[\sigma] \approx (\Lambda\alpha.\, \lambda x.\, x)\,[\sigma'] \in \llbracket R \to R \rrbracket$ for arbitrary σ, σ', $R : \sigma \leftrightarrow \sigma'$
True, if	$f_\sigma \sim \lambda x.\, x \in [R \to R]$ for $f\,[\sigma] \mapsto^* f_\sigma$
True, if	$f_\sigma\, v \approx (\lambda x.\, x)\, v' \in \llbracket R \to R \rrbracket$ for arbitrary $v \sim v' \in [R]$
True, if	$w \sim v' \in [R]$ for $f_\sigma\, v \mapsto^* w$
Holds, since	$f_\sigma\, v \mapsto^* v$ (by prior theorem) and determinism imply $w = v$

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. At the beginning of the next lecture we explore additional consequences of parametricity, so-called "*theorems for free*" [Wad89].

Exercises

Exercise 1 Prove that $\forall\alpha.\, \alpha \to \alpha \cong 1$. You may use the results of Section 3 and Section 5.

Exercise 2 Prove, using parametricity, that there cannot be a closed value $f : \forall\alpha.\, \alpha$.

Exercise 3 Prove, using parametricity, that if we have $f : \forall\alpha.\, \alpha \to \alpha \to \alpha$ for a value f then either $f \sim \Lambda\alpha.\, \lambda x.\, \lambda y.\, x \in [\forall\alpha.\, \alpha \to \alpha \to \alpha]$ or $f \sim \Lambda\alpha.\, \lambda x.\, \lambda y.\, y \in [\forall\alpha.\, \alpha \to \alpha \to \alpha]$.

Exercise 4 Prove, using parametricity, that $\forall\alpha.\, \alpha \to \alpha \to \alpha \cong 2$.

References

[Ahm06] Amal J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In P. Sestoft, editor, *15th European Symposium on Programming (ESOP 2006)*, pages 69–83, Vienna, Austria, March 2006. Springer LNCS 3924.

[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.

[Sta85] Richard Statman. Logical relations and the typed λ-calculus. *Information and Control*, 65:85–97, 1985.

[Wad89] Philip Wadler. Theorems 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 17
Thursday, October 29, 2020

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].

In this lecture we will first explore some more consequences of Reynolds's parametricity theorem that are used in modern compilers and then move towards questions of data abstraction and modularity.

2 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

$$f : \forall \alpha.\, \alpha \to \alpha$$

for a value f. First, parametricity tells us

$$f \sim f \in [\forall \alpha.\, \alpha \to \alpha]$$

This time, we pick types τ and τ' and a *relation* R which is in fact a *function* $R : \tau \to \tau'$. Then

$$f[\tau] \approx f[\tau'] \in [\![R \to R]\!]$$

which means that $f[\tau] \mapsto^* f_\tau$ and $f[\tau'] \mapsto^* f_{\tau'}$ with

$$f_\tau \sim f_{\tau'} \in [R \to R]$$

Now, for arbitrary values $v : \tau$ and $v' : \tau'$, $v\ R\ v'$ actually means $R\, v \mapsto^* v'$. Using the definition of $\sim$ at function type we get

$$f_\tau\, v \approx f_{\tau'}\, (R\, v) \in [\![R]\!]$$

but this in turn means

$$R\, (f_\tau\, v) \mapsto^* w \quad \text{and} \quad f_{\tau'}\, (R\, v) \mapsto^* w \quad \text{for some value } w$$

Wadler summarizes this by stating that for any function $R : \tau \to \tau'$,

$$R \circ f[\tau] = f[\tau'] \circ R$$

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

$$\begin{array}{lll} R\, (f[\tau]\, v_0) & \mapsto^* & v_0 \\ f[\tau]\, (R\, v_0) & \mapsto^* & f[\tau]\, v_0 \end{array}$$

so we find $f[\tau]\, v_0 \mapsto^* v_0$ which, since v_0 was arbitrary, is another way of saying that f behaves like the identity function.

3 Parametricity on Lists

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. Here, we think of lists defined with

$$\mathsf{list}\ \tau = \rho\alpha.\,(\mathbf{nil} : 1) + (\mathbf{cons} : \tau \times \alpha)$$

even though type constructors like list haven't been formally introduced into our language. Then we use a shorthand notation for lists, that is, elaborate the left-hand side into the right-hand side:

$$[e_1, \ldots, e_n] \triangleq \mathsf{fold}\ \mathbf{cons} \cdot \langle e_1, \ldots \mathsf{fold}\ \mathbf{cons} \cdot \langle e_n, \mathsf{fold}\ \mathbf{nil} \cdot \langle\,\rangle\rangle\rangle$$

We then extend the notion of logical equalities to values of list type *inductively* over the structure of the list, which reduces the type of the relation because each element has a smaller type.

> $e \sim e' : \mathsf{list}\ \tau$ *iff* $e \mapsto^* [v_1, \ldots, v_n]$, $e' \mapsto^* [v'_1, \ldots, v'_n]$ *and* $v_i \sim v'_i : \tau$ *for all* $1 \leq i \leq n$.

Then we have, for example, a polymorphic *map* function:

$$\begin{array}{l}
\mathit{map} : \forall\alpha.\,\forall\beta.\,(\alpha \to \beta) \to \mathsf{list}\ \alpha \to \mathsf{list}\ \beta \\
\mathit{map} = \Lambda\alpha.\,\Lambda\beta.\,\mathsf{fix}\ m.\,\lambda f.\,\lambda l. \\
\qquad\quad \mathsf{case}\ l\ \ (\ \mathsf{fold}\ \mathbf{nil} \cdot \langle\,\rangle \Rightarrow \mathsf{fold}\ \mathbf{nil} \cdot \langle\,\rangle \\
\qquad\qquad\quad\ \ \mid \mathsf{fold}\ \mathbf{cons} \cdot \langle x, l' \rangle \Rightarrow \mathsf{fold}\ \mathbf{cons} \cdot \langle f\,x, m\ f\ l'\ )
\end{array}$$

The map function then satisfies (for $f : \tau \to \tau'$):

$$\mathit{map}\ [\tau]\ [\tau']\ f\ [v_1, \ldots, v_n] = [f\,v_1, \ldots, f\,v_n]$$

where equality here is Kleene equality (both sides reduce to the same value). The example(s) are easier to understand if we isolate the special case list R for a relation $R : \tau \to \tau'$ which is actually a function. In this case we obtain

> $v \approx v' \in [\mathsf{list}\ R]$ for an $R : \tau \to \tau'$ iff $(\mathit{map}\ [\tau]\ [\tau'] R)\,v = v'$.

Returning to examples, what can the type tell us about a function

$$f : \forall\alpha.\,\mathsf{list}\ \alpha \to \alpha\ \mathsf{list}\ \alpha\ ?$$

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

$f \sim f \in [\forall \alpha.\, \mathsf{list}\ \alpha \rightarrow \mathsf{list}\ \alpha]$ *by parametricity.*

Now let $R : \tau \rightarrow \tau'$ be a function. Then $f[\tau] \mapsto^* f_\tau$, $f[\tau'] \mapsto^* f_{\tau'}$, and

$f_\tau \sim f_{\tau'} \in [\mathsf{list}\ R \rightarrow \mathsf{list}\ R]$ *by definition of* $\approx$*.*

Using the definition of $\sim$ on function types, we obtain

For any values $l : \mathsf{list}\ \tau$ *and* $l' : \mathsf{list}\ \tau'$ *with* $l\ (R\,\mathsf{list})\ l'$ *we have* $f_\tau\, l\ (R\,\mathsf{list})\ f_{\tau'}\, l'$

By the remark on the interpretation of R list when R is a function, this becomes

If $(\mathit{map}\ [\tau]\ [\tau']\ R)\, l = l'$ *then* $(\mathit{map}\ [\tau]\ [\tau']\ R)\ (f\ l) = f\ l'$

or, equivalently,

$(\mathit{map}\ [\tau]\ [\tau']\ R)\ (f\ [\tau]\ l) = f\ [\tau']\ ((\mathit{map}\ [\tau]\ [\tau']\ R)\ l)$.

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].

$$f : \forall\alpha.\, (\alpha\ \mathsf{list})\ \mathsf{list} \to \alpha\ \mathsf{list}$$
$$(\mathit{map}\ [\tau]\,[\tau']\,R)\,(f\,[\tau]\,l) = f\ [\tau']\,((\mathit{map}\ [\mathsf{list}\,\tau]\,[\mathsf{list}\,\tau']\,(\mathit{map}\ [\tau]\,[\tau']\,R))\ l)$$

$$f : \forall\alpha.\, (\alpha \to \mathit{bool}) \to \alpha\ \mathsf{list} \to \alpha\ \mathsf{list}$$
$$(\mathit{map}\ [\tau]\,[\tau']\,R)\ (f\,[\tau]\,(\lambda x.\, p\,(R\,x))\,l) = f\,[\tau']\,p\,((\mathit{map}\,[\tau]\,[\tau']\,R)\,l)$$

These theorems do not quite come "for free", but they are fairly straightforward consequences of parametricity, keeping in mind the requirement of termination.

4 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 an implementation of a counter with constant *new* and functions *inc* and *dec* to increment and decrement the counter. 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) to specify interfaces between a library and its client.

Below we name *CTR* as the *signature* that describes the interface of a module.

$\mathit{CTR} = \{$
  $\quad\mathbf{type}\ \mathit{ctr}$
  $\quad\mathit{new} : \mathit{ctr}$
  $\quad\mathit{inc} : \mathit{ctr} \to \mathit{ctr}$
  $\quad\mathit{dec} : \mathit{ctr} \to 1 + \mathit{ctr}$
$\}$

The value *new* will be a counter with initial value 0. The decrement function *dec* returns an optional counter with the new value, where we consider the predecessor of 0 to be undefined (returning $\mathbf{l} \cdot \langle\,\rangle$). This provides the only means for the client to observe the value of a counter. The implementations are straightforward so we elide them for now, and just assume we have type *nat* and *bin* and suitable functions on them.

$$
\begin{array}{l}
\mathit{NatCtr} : \mathit{CTR} = \{ \\
\quad \mathbf{type}\ \mathit{ctr} = \mathit{nat} \\
\quad \mathit{new} = \mathit{zero} \\
\quad \mathit{inc} = \mathit{succ} \\
\quad \mathit{dec} = \mathit{pred} \\
\}
\end{array}
$$

An interesting aspect of this definition is that, for example, *zero* : *nat* while the interface specifies *new* : *ctr*. But this is okay because the type *ctr* is in fact implemented by *nat* in this version. Next, we show the implementation using numbers in binary representation. This time, we define some of the functions directly in the module, assuming *e* : *bin* represents 0, and we have suitable functions *plus1* and *minus1* on binary numbers already defined.

$$
\begin{array}{l}
\mathit{BinCtr} : \mathit{CTR} = \{ \\
\quad \mathbf{type}\ \mathit{ctr} = \mathit{nat} \\
\quad \mathit{new} = \mathit{e} \\
\quad \mathit{inc} = \mathit{plus1} \\
\quad \mathit{dec} = \mathit{minus1} \\
\}
\end{array}
$$

Now what does a client look like? Assume it has an implemention *C* : *CTR*. 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 would write

$$
\begin{array}{l}
\mathbf{open}\ C : \mathit{CTR} \\
\mathit{isZero} : \mathit{ctr} \to \mathit{bool} \\
\mathit{isZero} = \lambda x.\, \mathsf{case}\ \mathit{dec}\ x\ (\, \mathbf{l} \cdot \langle\,\rangle \Rightarrow \mathit{true} \\
\qquad\qquad\qquad\qquad\quad \mid \mathbf{r} \cdot _ \Rightarrow \mathit{false}\,)
\end{array}
$$

but **not**

$$
\begin{array}{l}
\mathbf{open}\ C : \mathit{CTR} \\
\mathit{isZero} : \mathit{ctr} \to \mathit{bool} \\
\mathit{isZero} = \lambda n.\, \mathsf{case}\ (\mathsf{unfold}\ n)\ (\, \mathbf{zero} \cdot \langle\,\rangle \Rightarrow \mathit{true} \\
\qquad\qquad\qquad\qquad\qquad\quad \mid \mathbf{succ} \cdot _ \Rightarrow \mathit{false}\,)
\end{array}
$$

because the latter supposes that the library *C* : *CTR* implements the type *ctr* by *nat*, which it may not.

5 Formalizing Abstract Types

We will write a signature such as

$CTR = \{$
  $\mathbf{type}\ \mathit{ctr}$
  $\mathit{new} : \mathit{ctr}$
  $\mathit{inc} : \mathit{ctr} \to \mathit{ctr}$
  $\mathit{dec} : \mathit{ctr} \to 1 + \mathit{ctr}$
$\}$

in abstract form as

$$\exists \alpha.\ \underbrace{\alpha}_{\mathit{new}} \times \underbrace{(\alpha \to \alpha)}_{\mathit{inc}} \times \underbrace{(\alpha \to 1 + \alpha)}_{\mathit{dec}}$$

where the name annotations are just explanatory and not part of the syntax. Note that α stands for *ctr* which is bound here by the existential quantifier.

Now what should an expression

$$e : \exists \alpha.\, \alpha \times (\alpha \to \alpha) \times (\alpha \to 1 + \alpha)$$

look like? It should provide a concrete implementation type (such as *nat* or *bin*) for α, as well as an implementation of the three functions. We obtain this with the following rule

$$\frac{\Gamma \vdash \sigma\ \mathit{type} \quad \Gamma \vdash e : [\sigma/\alpha]\tau}{\Gamma \vdash \langle \sigma, e \rangle : \exists \alpha.\, \tau}\ \mathsf{tp/exists}$$

Besides checking that σ is indeed a type with respect to all the type variables declared in Γ, the crucial aspect of this rule is that the implementation e is at type $[\sigma/\alpha]\tau$.

For example, to check that *new*, *inc*, and *dec* are well-typed we substitute the implementation type for α (namely *nat* in one case and *bin* in the other case) before we proceed to check 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, and to support general pattern matching, we'll write is as a case.

Types	τ	$::=$	$\ldots \mid \exists \alpha.\, \tau$
Expressions	e	$::=$	$\ldots \mid \langle \sigma, e \rangle \mid \mathsf{case}\ e\ (\langle \alpha, x \rangle \Rightarrow e')$

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.

$$\frac{\Gamma \vdash e : \exists\alpha.\,\tau \qquad \begin{array}{c}(\alpha \notin \mathrm{dom}(\Gamma) \cup \mathrm{FTV}(\Gamma) \cup \mathrm{FTV}(\tau'))\\ \Gamma, \alpha\ \mathit{type}, x : \tau \vdash e' : \tau'\end{array}}{\Gamma \vdash \mathsf{case}\ e\ (\langle \alpha, x\rangle \Rightarrow e') : \tau'}\ \mathsf{tp/casee}$$

The fact that the type α must be *new* is explicit here in the conditions that it does not already appear in Γ or τ'. Such a condition is often left implicit, relying on the well-formedness presuppositions of the judgments. For example, the presupposition that Γ may not contain any repeated variables means that 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 are 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 $\Gamma \vdash e : \tau$ we make the following *presuppositions*:

1. All the variables and type variables in Γ are distinct.
2. Γ *ctx*
3. $\Gamma \vdash \tau$ *type*

where the validity of context is defined by the following rules:

$$\frac{}{(\cdot)\ \mathit{ctx}}\ \mathsf{ctx/emp} \qquad \frac{\Gamma\ \mathit{ctx}}{(\Gamma, \alpha\ \mathit{type})\ \mathit{ctx}}\ \mathsf{ctx/tpvar} \qquad \frac{\Gamma\ \mathit{ctx} \quad \Gamma \vdash \tau\ \mathit{type}}{(\Gamma, x : \tau)\ \mathit{ctx}}\ \mathsf{ctx/var}$$

With these presuppositions the condition on α in the tp/casee rule is automatically satisfied. 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 case/exists again in this light.

1. We assume all variables in Γ 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 $\Gamma, \alpha\ \mathit{type}, x : \tau$, which may include an implicit renaming of the type variable α or the variable x bound in the the expression $\langle \alpha, x\rangle \Rightarrow e'$.

2. By presupposition (from the conclusion), Γ *ctx*, which means that there are no *free* type variables in it, but variables declared in it can occur to their right. But what about τ? Actually, it is okay (and in fact mostly needed) for α to appear in τ.

3. By presupposition (from the conclusion), $\Gamma \vdash \tau'$ *type*. This covers the second premise. Often, this rule is given with an explicit premise $\Gamma \vdash \tau'$ *type* to emphasize τ' must be independent of α. Indeed, the scope of α is the type of x and the expression 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 dynamics is straightforward and not very interesting.

$$\frac{v\ \mathit{value}}{\langle \sigma, v\rangle\ \mathit{value}}\ \mathsf{val/exists} \qquad \frac{e \mapsto e'}{\langle \sigma, e\rangle \mapsto \langle \sigma, e'\rangle}\ \mathsf{step/pack}_1$$

$$\frac{e_0 \mapsto e_0'}{\mathsf{case}\ e_0\ (\langle \alpha, x\rangle \Rightarrow e_1) \mapsto \mathsf{case}\ e_0'\ (\langle \alpha, x\rangle \Rightarrow e_1)}\ \mathsf{step/casee}_0$$

$$\frac{}{\mathsf{case}\ \langle \sigma, v\rangle\ (\langle \alpha, x\rangle \Rightarrow e) \mapsto [\sigma/\alpha, v/x]e}\ \mathsf{step/casee/pack}$$

In a language with variadic sums and pattern matching, we would extend the language of patterns.

$$\text{Patterns}\quad p \quad ::= \quad x \mid \langle p_1, p_2\rangle \mid \langle\,\rangle \mid i \cdot p \mid \mathsf{fold}\ p \mid \langle \alpha, p\rangle$$

The hypothetical open construct now corresponds a pattern match, with the scope of the openend module extending to the end of the case expression. For example, we can test an implementation of *CTR* by creating a fresh counter and then verifying that incrementing it followed by a decrement has a well-defined answer. In the definition of *test* we exploit general pattern matching so an exception is raised if a decrement of zero was attempted.

$$\begin{array}{l} \mathit{test} : \mathit{CTR} \rightarrow 1 \\ \mathit{test} = \lambda c.\ \mathsf{case}\ c\ (\ \langle \alpha, \langle \mathit{new}, \langle \mathit{inc}, \mathit{dec}\rangle\rangle\rangle \Rightarrow \\ \qquad\qquad \mathsf{case}\ \mathit{dec}\ (\mathit{inc}\ \mathit{new})\ (\ \mathbf{r} \cdot _ \Rightarrow \langle\,\rangle\ )\ ) \end{array}$$

Note that the case opens the package (representing the module) and matches against its components so it can refer to them in the body of the function. The following two expressions will evaluate to unit (instead of raising an exception) if the implementation is correct (to the very limited extent that is tested here).

test NatCtr
test BinCtr

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.

[Wad89] Philip Wadler. Theorems 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 Representation Independence

15-814: Types and Programming Languages
Frank Pfenning

Lecture 18
Tuesday, November 3, 2020

1 Introduction

In this lecture we prove that we can replace the unary implementation of counters with the binary one without breaking any clients (or vice versa). This is a consequence of parametricity, and the definition of logical equality we developed in the previous two lectures, extended to existential types.

2 Existential Types and Parametricity

We have said that the client of a module (expressed as having an existential type) is parametric in the implementation type. Let's recall the crucial rules.

$$\frac{\Gamma \vdash \sigma\ \mathit{type} \quad \Gamma \vdash e : [\sigma/\alpha]\tau}{\Gamma \vdash \langle \sigma, e \rangle : \exists \alpha.\, \tau}\ \mathsf{tp/exists}$$

$$\frac{\Gamma \vdash e : \exists \alpha.\, \tau \quad \Gamma, \alpha\ \mathit{type}, x : \tau \vdash e' : \tau'}{\Gamma \vdash \mathsf{case}\ e\ (\langle \alpha, x \rangle \Rightarrow e') : \tau'}\ \mathsf{tp/casee}$$

The client here is e' in the $\mathsf{tp/casee}$ rule. From typing judgment for e' in the second premise we can infer

$$\frac{\dfrac{\Gamma, \alpha\ \mathit{type}, x : \tau \vdash e' : \tau'}{\Gamma, \alpha\ \mathit{type} \vdash \lambda x.\, e' : \tau \to \tau'}\ \mathsf{tp/lam}}{\Gamma \vdash \Lambda \alpha.\, \lambda x.\, e' : \forall \alpha.\, \tau \to \tau'}\ \mathsf{tp/tplam}$$

to see that, indeed, $\lambda x.\, e'$ is parametric in α and therefore also e'.

3 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 between the two given implementation types.

Recall from Lecture 16:

($\forall$) $v \sim v' \in [\forall\alpha.\,\tau]$ iff for all closed types σ and σ' and relations $R : \sigma \leftrightarrow \sigma'$ we have $v\,[\sigma] \approx v'\,[\sigma'] \in [\![[R/\alpha]\tau]\!]$

(R) $v \sim v' \in [R]$ iff $v \; R \; v'$.

We add

($\exists$) $v \sim v' \in [\exists\alpha.\,\tau]$ iff $v = \langle \sigma, v_1 \rangle$ and $v' = \langle \sigma', v_1' \rangle$ for some closed types σ, σ' and values v_1, v_1', and there is a relation $R : \sigma \leftrightarrow \sigma'$ such that $v_1 \sim v_1' \in [[R/\alpha]\tau]$.

In our example, we ask if

$$\mathit{NatCtr} \sim \mathit{BinCtr} \in [\mathit{CTR}]$$

which unfolds into demonstrating that there is a relation $R : \mathit{nat} \leftrightarrow \mathit{bin}$ such that

$$\langle \mathit{zero}, \langle \mathit{succ}, \mathit{pred} \rangle \rangle \sim \langle e, \langle \mathit{inc}, \mathit{dec} \rangle \rangle \in [R \times (R \to R) \times (R \to 1 + R)]$$

Since logical equality at type $\tau_1 \times \tau_2$ 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.

For reference, the complete implementation can be found in `exists.cbv`. In Listing 1 we show the implementation *NatCtr* and *BinCtr* in LAMBDA. The concrete syntax for an existential type $\exists\alpha.\,\tau$ is `?a.tau`, and a package $\langle \sigma, e \rangle$ is written as `([sigma],e)`. This notation means that, uniformly, types occurring in expressions are enclosed in square brackets.

4 Defining a Relation Between Implementations

The relation $R : \mathit{nat} \leftrightarrow \mathit{bin}$ we seek needs to relate natural numbers in two different representations. It is convenient and general to define such relations by using inference rules. In particular, this will allow us to prove

```
1  decl pred : nat -> 1 + nat
2  defn pred = \n. case n of ( fold 'zero () => 'l ()
3                            | fold 'succ m => 'r m )
4
5  decl dec : bin -> 1 + bin
6  defn dec = $dec. \x.
7    case x
8      of ( fold 'b0 y => case dec y  % 2y-1 = 2(y-1)+1
9                         of ( 'l () => 'l ()
10                           | 'r z => 'r (b1 z) )
11        | fold 'b1 y => 'r (b0 y)   % (2y+1)-1 = 2y
12        | fold 'e () => 'l () )
13
14 type CTR = ?a. a * (a -> a) * (a -> 1 + a)
15
16 decl NatCtr : CTR
17 defn NatCtr = ([nat], zero, succ, pred)
18
19 decl BinCtr : CTR
20 defn BinCtr = ([bin], e, inc, dec)
```

Listing 1: Binary counters as an abstract type

properties by *rule induction*. An alternative approach would be to define such relations as functions, but because representations are often not unique this is not quite as general.

Once we have made this decision, the relation could be based on the structure of $x : \mathit{bin}$ or on the structure of $n : \mathit{nat}$. The latter 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 concisely, we use a representation function for (mathematical) natural numbers k into our language of values defined by

$$
\begin{array}{lcl}
\overline{0} & = & \mathsf{fold}\ \mathbf{zero} \cdot \langle\,\rangle \\
\overline{n+1} & = & \mathsf{fold}\ \mathbf{succ} \cdot \overline{n}
\end{array}
$$

We also write binary number representations in compressed form with the

least significant bit first:[1]

$$\begin{array}{rcl} \mathsf{0}x & = & \mathsf{fold}\ \mathbf{b0} \cdot x \\ \mathsf{1}x & = & \mathsf{fold}\ \mathbf{b1} \cdot x \\ \mathsf{e} & = & \mathsf{fold}\ \mathbf{e} \cdot \langle\, \rangle \end{array}$$

Recall the ambiguity that e, $\mathsf{0e}$, $\mathsf{00e}$ etc. all represent the natural number 0.

We then define:

$$\frac{}{\overline{0}\ R\ \mathsf{e}}\ R_e \qquad \frac{\overline{k}\ R\ x}{\overline{2k}\ R\ \mathsf{0}x}\ R_0 \qquad \frac{\overline{k}\ R\ x}{\overline{2k+1}\ R\ \mathsf{1}x}\ R_1$$

As usual, we consider $n\ R\ x$ to hold if and only if we can derive it using these rules.

5 Verifying the Relation

Because our signature exposes three constants, we now have to check three properties:

$$\begin{array}{l} \mathit{zero} \sim e \in [R] \\ \mathit{succ} \sim \mathit{inc} \in [R \to R] \\ \mathit{pred} \sim \mathit{dec} \in [R \to 1 + R] \end{array}$$

We already have by definition that $v \sim v' \in [R]$ iff $v\ R\ v'$. For convenience, we also define the notation $e\ \mathbb{R}\ e'$ to stand for $e \approx e' \in [\![R]\!]$, which means that $e \mapsto^* v$ and $e' \mapsto^* v'$ with $v\ R\ v'$

Lemma 1 *$\mathit{zero} \sim e \in [R]$.*

Proof: Since $\overline{0} = \mathit{zero}$ and $\mathsf{e} = e$ this is just the contents of rule R_e. □

Lemma 2 *$\mathit{succ} \sim \mathit{inc} \in [R \to R]$.*

Proof: By definition of logical equality, this is equivalent to showing

> *For all values $n : \mathit{nat}$ and $x : \mathit{bin}$ with $n\ R\ x$ we have $(\mathit{succ}\ n)\ \mathbb{R}\ (\mathit{inc}\ x)$.*

[1] In lecture, we used the notation $\mathit{b0}\,x$, $\mathit{b1}\,x$ and e to stand for the corresponding *values*, but that is somewhat ambiguous since *b0* and *b1* were previously defined as function in our language rather than as functions at the metalevel as we need here.

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

$$\frac{}{\overline{0}\ R\ \mathsf{e}}\ R_e$$

with $n = \overline{0}$ and $x = \mathsf{e}$. We have to show that $(\mathit{succ}\ \overline{0})\ \mathbb{R}\ (\mathit{inc}\ \mathsf{e})$

$\mathit{succ}\ \overline{0} \mapsto^* \overline{1}$	By defn. of *succ*
$\mathit{inc}\ \mathsf{e} \mapsto^* \mathsf{1e}$	By defn. of *inc*
$\overline{1}\ R\ \mathsf{1e}$	By rules R_1 and R_e

Case: Rule

$$\frac{\overline{k}\ R\ y}{\overline{2k}\ R\ \mathsf{0}y}\ R_0$$

where $x = \mathsf{0}y$ and $n = \overline{2k}$. To prove is $(\mathit{succ}\ \overline{2k})\ \mathbb{R}\ (\mathit{inc}\ \mathsf{0}y)$.

$\mathit{succ}\ \overline{2k} \mapsto^* \overline{2k+1}$	By defn of *succ*
$\mathit{inc}\ \mathsf{0}y \mapsto^* \mathsf{1}y$	By defn. of *inc*
$\overline{k}\ R\ y$	Premise in this case
$\overline{2k+1}\ R\ \mathsf{1}y$	By rule R_1

Case: Rule

$$\frac{\overline{k}\ R\ y}{\overline{2k+1}\ R\ \mathsf{1}y}\ R_1$$

where $n = \overline{2k+1}$ and $x = \mathsf{1}y$. To show: $(\mathit{succ}\ \overline{2k+1})\ \mathbb{R}\ (\mathit{inc}\ \mathsf{1}y)$.

$\mathit{succ}\ \overline{2k+1} \mapsto^* \overline{2k+2}$	By defn. of *succ*
$\mathit{inc}\ \mathsf{1}y \mapsto^* \mathit{b0}\ (\mathit{inc}\ y) \mapsto^* \mathsf{0}z$ where $\mathit{inc}\ y \mapsto^* z$	By defn. of *inc* and *b0*
Remains to show: $\overline{2k+2}\ R\ \mathsf{0}z$	
$\overline{k}\ R\ y$	Premise in this case
$(\mathit{succ}\ \overline{k})\ \mathbb{R}\ (\mathit{inc}\ y)$	By ind. hyp.
$\overline{k+1}\ R\ z$	By defn. of $\mathbb{R}$ and *succ*
$\overline{2(k+1)}\ R\ \mathsf{0}z$	By rule R_0
$\overline{2k+2}\ R\ \mathsf{0}z$	By arithmetic

□

In order to prove the relation between the implementation of the predecessor function we write out the interpretation of the type $1 + R$.

$v \sim v' \in [1 + R]$ iff ($v = \mathbf{l} \cdot \langle\,\rangle$ and $v' = \mathbf{l} \cdot \langle\,\rangle$)
or ($v = \mathbf{r} \cdot v_1$ and $v' = \mathbf{r} \cdot v_1'$ and $v_1 \; R \; v_1'$.

Lemma 3 *pred* $\sim$ *dec* $\in [R \to 1 + R]$

Proof: By[2] definition of logical equality, this is equivalent to showing

> *For all values n : nat and x : bin with $n \; R \; x$ we have pred $n \approx$ dec $x \in [\![1 + R]\!]$.*

We break this down into two properties, based on n.

(i) For all $\overline{0} \; R \; x$ we have *pred* $\overline{0} \approx$ *dec* $x \in [\![(\mathbf{l} : 1)]\!]$.

(ii) For all $\overline{k+1} \; R \; x$ we have *pred* $\overline{k+1} \approx$ *dec* $x \in [\![(\mathbf{r} : R)]\!]$.

For part (i), we note that *pred* $\overline{0} \mapsto^* \mathbf{l} \cdot \langle\,\rangle$, so all that remains to show is that *dec* $x \mapsto^* \mathbf{l} \cdot \langle\,\rangle$ for all $\overline{0} \; R \; x$. We prove this by rule induction on the derivation of $\overline{0} \; R \; x$.

Case(i):

$$\frac{}{\overline{0} \; R \; \mathsf{e}} \; R_e$$

where $x = \mathsf{e}$. Then *dec* x = *dec* $\mathsf{e} \mapsto^* \mathbf{l} \cdot \langle\,\rangle$.

Case(i):

$$\frac{\overline{k} \; R \; y}{\overline{2k} \; R \; \mathsf{0}y} \; R_0$$

where $x = \mathsf{0}y$ and $2k = 0$ and therefore also $k = 0$. Then

dec $\mathsf{0}y \mapsto^*$ case (*dec* y) $(\mathbf{l} \cdot \langle\,\rangle \Rightarrow \mathbf{l} \cdot \langle\,\rangle \mid \mathbf{r} \cdot z \Rightarrow \mathbf{r} \cdot (b1 \; z))$ By defn. of *dec*
dec $y \mapsto^* \mathbf{l} \cdot \langle\,\rangle$ By ind. hyp.
case (*dec* y) $(\mathbf{l} \cdot \langle\,\rangle \Rightarrow \mathbf{l} \cdot \langle\,\rangle \mid \mathbf{r} \cdot z \Rightarrow \mathbf{r} \cdot (b1 \; z)) \mapsto^* \mathbf{l} \cdot \langle\,\rangle$

[2] We skipped this part of the proof in lecture.

Case(i):

$$\frac{\overline{k}\ R\ y}{\overline{2k+1}\ R\ 1y}\ R_1$$

This case is impossible since $2k+1 \neq 0$.

Now we come to Part (ii). We note that *pred* $\overline{k+1} \mapsto^* \mathbf{r} \cdot \overline{k}$ so what we have to show is that

(ii)′ For all $\overline{k+1}\ R\ x$ we have *dec* $x \mapsto^* \mathbf{r} \cdot y$ with $\overline{k}\ R\ y$.

We prove this by rule induction on the derivation of $\overline{k+1}\ R\ x$.

Case(ii):

$$\frac{}{\overline{0}\ R\ \mathsf{e}}\ R_e$$

is impossible since $\overline{0} \neq \overline{k+1}$.

Case(ii):

$$\frac{\overline{j}\ R\ y}{\overline{2j}\ R\ 0y}\ R_0$$

where $k+1 = 2j$ and $x = 0y$.

$j = j' + 1$ for some j'	Since $j > 0$ by arithmetic
dec $y \mapsto^* \mathbf{r} \cdot z$ with $\overline{j'}\ R\ z$	By ind. hyp.
dec $0y \mapsto^* \mathbf{r} \cdot 1z$	By defn. of *dec*
$\overline{2j'+1}\ R\ 1z$	By rule R_1
$\overline{k}\ R\ 1z$	By arithmetic

Case(ii):

$$\frac{\overline{j}\ R\ y}{\overline{2j+1}\ R\ 1y}\ R_1$$

for $k+1 = 2j+1$ and $x = 1y$. Then

dec $1y \mapsto^* \mathbf{r} \cdot 0y$	By defn. of *dec*
$\overline{j}\ R\ y$	Premise in this case
$\overline{2j}\ R\ 0y$	By rule R_0
$\overline{k}\ R\ 0y$	By arithmetic

□

6 Concrete Types vs. Abstract Types

An interesting observation about the logical equivalence of the two implementation of counters is that, had we omitted the decrement operation from the interface, then universal relation ($n\ U\ x$ for all values $n : \mathit{nat}$ and $x : \mathit{bin}$) also allows us to prove equivalence. This is because without the decrement we can create a counter and increment it, but can never observe any of its properties.

This raises the question how we should more generally observe properties of elements of abstract type. There is no universal answer: different applications or libraries require different choices. A particularly frequent and useful technique is to endow abstract types with a *view*, realized by a function called *expose* or *out*.

As an example, let's reconsider the (concrete) type of binary numbers:

$$\mathit{bin} = (\mathbf{b0} : \mathit{bin}) + (\mathbf{b1} : \mathit{bin}) + (\mathbf{e} : 1)$$

This concrete type allows clients to construct numbers with leading zeros, which may be undesirable because it complicates certain algorithms (e.g., equality of binary numbers). In this case, one solution would be to split the type *bin* into positive numbers *pos* and numbers in standard form *std* (with no leading zeros), which we did in Lecture 11, Exercise 1. However, now all client code has to be aware of these two types and use them appropriately. Alternatively, we can create an abstract type providing the constructors in the interface. to start with, we would have

$$\begin{array}{lll} \mathit{BIN} = \exists \alpha.\ (\alpha \to \alpha) & & \%\ \mathit{b0} \\ \qquad\quad \times (\alpha \to \alpha) & & \%\ \mathit{b1} \\ \qquad\quad \times \alpha & & \%\ \mathit{e} \\ \qquad\quad \times \ldots & & \end{array}$$

The implementation of these constructors can make sure that only numbers with no leading zeros are ever created. But how do we *observe* a value of the abstract type? The technique is to provide a function $\mathit{out} : \alpha \to \tau$ where τ is usually a sum that the client can pattern match against. Here we would have

$$\begin{array}{lll} \mathit{BIN} = \exists \alpha.\ (\alpha \to \alpha) & & \%\ \mathit{b0} \\ \qquad\quad \times (\alpha \to \alpha) & & \%\ \mathit{b1} \\ \qquad\quad \times \alpha & & \%\ \mathit{e} \\ \qquad\quad \times (\alpha \to (\mathbf{b0} : \alpha) + (\mathbf{b1} : \alpha) + (\mathbf{e} : 1)) & & \%\ \mathit{out} \end{array}$$

The result *out* v where v is a value of the abstract type allows one level of pattern matching. The value tagged by **b0** or **b1** is again of abstract type and we must apply *out* again. If we want to allow multiple levels of pattern matching we would need some special syntax to designate *out* as a view with a corresponding pattern constructor, say, out^{-1}. Then matching the value $v : \alpha$ against the pattern $out^{-1}\ p$ will evaluate $out\ v \mapsto^* w$ and match w against p.

We show the implementation of this abstract type in LAMBDA. In this example, the *out* function just has to unfold the recursive type to expose the sum underneath.

```
type BIN = ?a. (a -> a)    % b0 = \n. 2n
             * (a -> a)    % b1 = \n. 2n+1
             * a           % e = 0
             * (a -> (('b0 : a) + ('b1 : a) + ('e : 1))) % out

decl Bin : BIN
defn Bin = ([bin], \x. case x of ( fold 'e () => e
                                 | _  => b0 x ),
                   \x. b1 x, e, \x. unfold x)
```

The only other interesting part of this is the constructor corresponding to the tag **b0** ensures that it never constructs 0e but returns e instead, thereby making the representation unique.

7 Polymorphic Lists

In functional languages lists are usually represented by a so-called *type constructor list* : *type* $\rightarrow$ *type*. That is, for any type τ, we would have

$$list\ \tau = \rho\beta.\,(\mathbf{nil} : 1) + (\mathbf{cons} : \tau \times \beta)$$

We have not introduced type constructors into our language, so we cannot express this directly. But we can formulate it as an abstract type. Essentially, the implementation is a *function* which takes an element type τ as an argument and returns in instance of an existential type for this particular τ.

```
type LIST = !a. ?b. b              % nil
                  * (a * b -> b)   % cons x l
                  * (b -> ('nil : 1) + ('cons : a * b)) % out l
```

There is, however, a quirk with the implementation that often comes up with abstract types. If we have an implementation of lists, for example

```
decl List : LIST
defn List = /\a. ([$list. ('nil : 1) + ('cons : a * list)],
                   fold 'nil (),
                   \p. fold 'cons p,
                   \l. unfold l)
```

then two different uses of this, e.g., *List* [*nat*] and *List* [*nat*] are incompatible because there is no way the type checker can know that the different abstract types are actually equal. We summarize this sometimes by saying that abstract types are *generative* because every time an implementation of an abstract type is opened, a fresh type variable is generated to stand for the implementation type.

This implementation of lists, by the way, is called a *functor* in languages in the ML family, because it is a module-level function. We think of it this way because it is a function that returns an abstract type when given a type.

8 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 tp/casee rule:

$$\frac{\Gamma \vdash e : \exists \alpha.\, \tau \quad \Gamma, \alpha\ \mathit{type}, x : \tau \vdash e' : \tau'}{\Gamma \vdash \mathbf{case}\ e\ (\langle \alpha, x\rangle \Rightarrow e') : \tau'}\ \mathsf{tp/casee}$$

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

$$\Lambda\alpha.\, \lambda x.\, e' : \forall \alpha.\, \tau \to \tau'$$

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 $v_0 \sim v_0' \in [[R/\alpha]\tau]$ then $(\Lambda\alpha\lambda x.\, e')[\sigma]\, v_0 \approx (\Lambda\alpha.\, \lambda x.\, e')[\sigma']\ v_0' \in [\![[R/\alpha]\tau']\!]$. But α does not occur in τ', so this is just saying that $[\sigma/\alpha, v_0/x]e' \approx [\sigma'/\alpha, v_0'/x]e' \in [\![\tau']\!]$. So the result of substituting the two different implementations is equivalent.

Exercises

Exercise 1 We can represent integers a as pairs $\langle x, y\rangle$ of natural numbers where $a = x - y$. We call this the *difference representation* and call the representation type *diff*.

$$\begin{array}{lcl} \mathit{nat} & = & \rho\alpha.\,(\mathbf{zero} : 1) + (\mathbf{succ} : \mathit{alpha}) \\ \mathit{diff} & = & \mathit{nat} \times \mathit{nat} \end{array}$$

In your answers below you may use *constructors* $\mathit{zero} : \mathit{nat}$ and $\mathit{succ} : \mathit{nat} \to \mathit{nat}$ to construct terms of type *nat*. If you need auxiliary functions on natural numbers, you should define them.

1. Define a function $\mathit{nat2diff} : \mathit{nat} \to \mathit{diff}$ that, when given a representation of the natural number n returns an integer representing n.

2. Define a constant $\mathit{d_zero} : \mathit{diff}$ representing the integer 0 as well as functions $\mathit{dplus} : \mathit{diff} \to \mathit{diff} \to \mathit{diff}$ and $\mathit{dminus} : \mathit{diff} \to \mathit{diff} \to \mathit{diff}$ representing addition and subtraction on integers, respectively.

3. Consider the type

   $$\mathit{ord} = (\mathbf{lt} : 1) + (\mathbf{eq} : 1) + (\mathbf{gt} : 1)$$

   that represents the outcome of a comparison (**lt** = "less than", **eq** = "equal", **gt** = "greater than"). Define a function $\mathit{dcompare} : \mathit{diff} \to \mathit{diff} \to \mathit{ord}$ to compare the two integer arguments. Again, you may use *lt*, *eq* and *gt* as constructors.

Exercise 2 We consider an alternative *signed representation* of integers where

$$\mathit{sign} = (\mathbf{pos} : \mathit{nat}) + (\mathbf{neg} : \mathit{nat})$$

where $\mathbf{pos} \cdot x$ represents the integer x and $\mathbf{neg} \cdot x$ represents the integer $-x$. In your answers below you may use *pos* and *neg* as data constructors, to construct elements of type *sign*. Define the following functions in analogy with Exercise 1:

1. $\mathit{nat2sign} : \mathit{nat} \to \mathit{sign}$

2. $\mathit{s_zero} : \mathit{sign}$

3. $\mathit{s_plus} : \mathit{sign} \to \mathit{sign} \to \mathit{sign}$

4. $s_minus : sign \to sign \to sign$

5. $s_compare : sign \to sign \to ord$

Exercise 3 In this exercise we pursue two different implementations of an integer counter, which can become negative (unlike the natural number counter in this lecture). The functions are simpler than the ones in Exercise 1 and Exercise 2 so that the logical equality argument is more manageable. We specify a signature

$INTCTR = \{$
  type $ictr$
  $new : ictr$
  $inc : ictr \to ictr$
  $dec : ictr \to ictr$
  $is0 : ictr \to bool$
$\}$

where *new*, *inc*, *dec* and *is0* have their obvious specification with respect to integers, generalizing the *CTR* type defined in the last lecture and used in this one.

1. Write out the definition of *INTCTR* as an existential type.

2. Define the constants and functions d_zero, d_inc, d_dec and d_is0 for the implementation where type $ictr = diff$ from Exercise 1.

3. Define the constants and functions $szero$, s_inc, s_dec and s_is0 for the implementation where type $ictr = sign$ from Exercise 2.

Now consider the two definitions

$$\begin{array}{l} DiffCtr : INTCTR = \langle diff, \langle d_zero, d_inc, d_dec, d_is0 \rangle\rangle \\ SignCtr : INTCTR = \langle sign, \langle s_zero, s_inc, s_dec, s_is0 \rangle\rangle \end{array}$$

4. Prove that $DiffCtr \sim SignCtr \in [INTCTR]$ by defining a suitable relation $R : diff \leftrightarrow sign$ and proving that

$$\begin{array}{c} \langle d_zero, d_inc, d_dec, d_is0 \rangle \sim \langle s_zero, s_inc, s_dec, s_is0 \rangle \\ \in [R \times (R \to R) \times (R \to R) \times (R \to bool)] \end{array}$$

Lecture Notes on
Shared Memory Concurrency

15-814: Types and Programming Languages
Frank Pfenning

Lecture 19
Thursday, November 5, 2020

1 Introduction

The main objective of this lecture is to start making the role of memory explicit in a description of the dynamics of our programming language. Towards that goal, we take several steps at the same time:

1. We introduce a translation from our source language of *expressions* to an intermediate language of concurrent *processes* that act on (shared) memory. The sequential semantics of our original language can be recovered as a particular *scheduling policy* for concurrent processes.
2. We introduce a new collection of semantic objects that represent the state of processes and the shared memory they operate on. The presentation is as a *substructural operational semantics* [Pfe04, PS09, CS09]
3. We introduce *destination-passing style* [CPWW02] as a particular style of specification for the dynamics of programming languages that seems to be particularly suitable for an explicit store.

We now start to develop the ideas in a piecemeal fashion. This lecture is based on very recent work, at present under submission [PP20].

2 Representing the Store

Our typing judgment for expressions is

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

By the time we actually evaluate e, all the variables declared in Γ will have been replaced by values v (values, because we are in a call-by-value language, with variables for fixed point expressions representing an exception to that rule). Evaluation of *closed* expressions e proceeds as

$$e \mapsto e_1 \mapsto e_2 \mapsto \cdots \mapsto v$$

where v (if the computation is finite) represents the final outcome of the evaluation. A nice property of this formulation of the dynamics is that it does not require any semantic artifacts: we stay entirely within the language of expressions (which include values). The K Machine from Lecture 12 introduced continuations as a first dynamic artifact.

The main dynamic artifact we care about in this lecture is a representation of the *store* or *memory*, terms we use interchangeably. In our formulation, cells can hold only *small values* W (yet to be defined) and we write

$$\mathsf{cell}\ c_0\ W_0, \mathsf{cell}\ c_1\ W_1, \ldots, \mathsf{cell}\ c_n\ W_n$$

where all c_i are distinct. We read $\mathsf{cell}\ c\ W$ as "*cell c contains W*" or "*the memory at address c holds W*". We will shortly generalize this further.

As an example, before we actually see how these arise, let's consider the representation of a list. We define

$$\mathit{list}\ \alpha \cong (\mathbf{nil} : 1) + (\mathbf{cons} : \alpha \times \mathit{list}\ \alpha)$$

Then a list with two values $v_1 : \tau$ and $v_2 : \tau$ would be written as an *expression*

$$\mathsf{fold}\ (\mathbf{cons} \cdot \langle v_1, \mathsf{fold}\ (\mathbf{cons} \cdot \langle v_2, \mathsf{fold}\ (\mathbf{nil} \cdot \langle\, \rangle)\rangle)\rangle) : \mathit{list}\ \tau$$

Our representation of this in memory at some initial address c_0 would be

$$\begin{array}{l} \mathsf{cell}\ c_8\ \langle\, \rangle \\ \mathsf{cell}\ c_7\ (\mathbf{nil} \cdot c_8), \\ \mathsf{cell}\ c_6\ (\mathsf{fold}\ c_7), \\ \mathsf{cell}\ c_5\ \langle a_2, c_6 \rangle, \\ \mathsf{cell}\ c_4\ (\mathbf{cons} \cdot c_5), \\ \mathsf{cell}\ c_3\ (\mathsf{fold}\ c_4), \\ \mathsf{cell}\ c_2\ \langle a_1, c_3 \rangle, \\ \mathsf{cell}\ c_1\ (\mathbf{cons} \cdot c_2), \\ \mathsf{cell}\ c_0\ (\mathsf{fold}\ c_1) \end{array}$$

Here, we assume a_1 is the *address* of v_1 in memory, and a_2 the address of v_2. You can see a list of length n requires $3n + 3$ cells. In a lower-level representation this could presumably be optimized by compressing the information.

3 From Expressions to Processes

We translate expressions e to processes P. Instead of returning a value v, a process P executes and writes the result of computation to a *destination* d which is the address of a cell in memory. So we write the translation as

$$[\![e]\!]\, d = P$$

which means that expression e translates to a process P that computes with destination d. Given an expression

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

its translation $P = [\![e]\!]\, d$ will be typed as

$$\Gamma \vdash P :: (d : \tau)$$

In this typing judgment we have made the destination d of the computation explicit. But the reinterpretation does not end there: we also no longer substitute values for the variables in Γ. Instead, we substitute *addresses*, so the process P can *read* from memory at the addresses in Γ and must *write* to the destination d (unless it does not terminate). We will also arrange that after writing to destination d the process P will immediately terminate. Explicitly:

$$\underbrace{c_1 : \tau_1, \ldots, c_n : \tau_n}_{\text{read from}} \vdash P :: \underbrace{(d : \tau)}_{\text{write to}}$$

Because at the moment we are only interested in modeling our pure functional language and *not* arbitrary mutation of memory, we require that all the c_i and d are distinct.

For each process P that is executing we have a semantic object

$$\mathsf{proc}\ d\ P$$

which means that P is executing with destination d. We do not make the cells that P may read from explicit because it would introduce unnecessary clutter.

4 Allocation and Spawn

Given the logic explained in the preceding sections, there is a single construct in our language of processes that accomplishes two things: (a) it allocates a

new cell in memory, and (b) it spawns a process whose job it is to write to this cell. We may also have a single initial cell c_0 to hold the outcome of the overall computation. We write this as

$$\text{Process} \quad P \quad ::= \quad x \leftarrow P \; ; \; Q \mid \ldots$$

where the scope of x includes both P and Q. More specifically, a new destination c is created, P is spawned with destination c, and Q can read from c (once its value has been written. We formalize this as

$$\mathcal{C}, \mathsf{proc}\; d\; (x \leftarrow P \; ; \; Q) \mapsto \mathcal{C}, \mathsf{proc}\; c\; ([c/x]P), \mathsf{proc}\; d\; ([c/x]Q) \quad (c \text{ fresh})$$

Here $\mathcal{C}$ represents the remaining *configuration*, which includes the representation of memory and other processes that may be executing. The freshly allocated cell at address c is uninitialized to start with. It represents a point of synchronization between P and Q, because Q can only read from it after P has written to it. Except for this synchronization point, P and Q can now evolve independently.

From a typing perspective, we can see that the type of two occurrences of the cell x must match.

$$\frac{\Gamma \vdash P :: (x : \tau) \quad \Gamma, x : \tau \vdash Q :: (d : \sigma)}{\Gamma \vdash x \leftarrow P \; ; \; Q :: (d : \sigma)} \; \mathsf{cut}$$

This rule is called *cut* because of this name for the corresponding logical rule in the *sequent calculus*

$$\frac{\Gamma \vdash A \quad \Gamma, A \vdash C}{\Gamma \vdash C} \; \mathsf{cut}$$

where A acts as a lemma in the proof of C from Γ.

The configuration is not intrinsically ordered, so the process with destination d can occur anywhere in a configuration. Nevertheless, we follow a convention writing a configuration (or part of a configuration) so that a cell c precedes all the processes that may read from c or other cells that contain c. Because we do not have arbitrary mutation of store there cannot be any cycles (although we have to carefully reconsider this point when we consider fixed point expressions).

Since all of our rules only operate locally on a small part of the configuration, we generally omit $\mathcal{C}$ to stand for the remainder of the configuration. But we always have to remember that we *remove* the part of the configuration matching the left-hand side of a transition rule and then we *add in* the right-hand side.

5 Copying

Before we get into the constructors and destructors for specific types in our source language of expressions, let's consider the translation of variables. We write

$$[\![x]\!]\, d = d \leftarrow x$$

The intuitive meaning of the process expression $d \leftarrow x$ is that it copies the contents of the cell at address x to address d. Thereby, this process has written to its destination d and terminates.

$$\mathsf{cell}\ c\ W, \mathsf{proc}\ d\ (d \leftarrow c), \mapsto \mathsf{cell}\ c\ W, \mathsf{cell}\ d\ W$$

In this rule the cell c should have been written to already, and we just copy its value (which is small) to d.

The typing rule just requires that c and d have the same type (otherwise copying would violate type preservation).

$$\frac{}{\Gamma, c : \tau \vdash (d \leftarrow c) :: (d : \tau)}\ \mathsf{id}$$

From a logical perspective, it explains that the antecedent A entails the succedent A in the sequent calculus, usually called the identity rule.

$$\frac{}{\Gamma, A \vdash A}\ \mathsf{id}$$

6 The Unit Type

Recall the constructor and destructor for the unit type 1.

$$\text{Expressions} \quad e \quad ::= \quad \langle\,\rangle \mid \mathsf{case}\ e\ (\langle\,\rangle \Rightarrow e') \mid \ldots$$

The unit element is already a small value, so it can be written directly to memory. Our notation for this is $d.\langle\,\rangle$.

$$\begin{array}{l} [\![\langle\,\rangle]\!]\, d = d.\langle\,\rangle \\ \mathsf{proc}\ d\ (d.\langle\,\rangle) \mapsto \mathsf{cell}\ d\ \langle\,\rangle \end{array}$$

$$\frac{}{\Gamma \vdash d.\langle\,\rangle :: (d : 1)}\ 1R$$

The way we evaluate $\mathsf{case}\ e\ (\langle\,\rangle \Rightarrow e')$ is to first evaluate e and then match the resulting value against the pattern $\langle\,\rangle$. Actually, we know by typing this will be the only possibility.

$$\begin{aligned}[\![\mathsf{case}\ e\ (\langle\,\rangle \Rightarrow e')]\!]\ d = {} & x \leftarrow [\![e]\!]\ x\ ; \\ & \mathsf{case}\ x\ (\langle\,\rangle \Rightarrow [\![e']\!]\ d)\end{aligned}$$

Note here how the process executing $[\![e]\!]\ x$ will write to a fresh destination c (substituted for x) and the $\mathsf{case}\ c$ destructor will read the value of c from memory when it becomes available. We then continue with the evaluation of e' to fill the original destination d.

$$\mathsf{cell}\ c\ \langle\,\rangle, \mathsf{proc}\ d\ (\mathsf{case}\ c\ (\langle\,\rangle \Rightarrow P)) \mapsto \mathsf{cell}\ c\ \langle\,\rangle, \mathsf{proc}\ d\ P$$

We see here that we need to replicate the cell c that we read on the right-hand side of the rule because there may be other processes that may want to read c. Because this is a frequent pattern, we mark cells that have a value as *persistent* by writing $!\mathsf{cell}\ c\ W$. It means this object, once created, persists from then on. In particular, if it occurs on the left-hand side of a transition rule it is *not* removed from the configuration. We now rewrite our rules with this notation:

$$\begin{aligned}&\mathsf{proc}\ d\ (d.\langle\,\rangle) \mapsto\ !\mathsf{cell}\ d\ \langle\,\rangle \\ &!\mathsf{cell}\ c\ \langle\,\rangle, \mathsf{proc}\ d\ (\mathsf{case}\ c\ (\langle\,\rangle \Rightarrow P)) \mapsto \mathsf{proc}\ d\ P\end{aligned}$$

The typing rule for this case construct is straightforward.

$$\frac{c : 1 \in \Gamma \quad \Gamma \vdash P :: (d : \tau)}{\Gamma \vdash \mathsf{case}\ c\ (\langle\,\rangle \Rightarrow P) :: (d : \tau)}\ 1L$$

We name these rules $1R$ (the type 1 occurring in the succedent) and $1L$ (the type 1 occurring among the antecedents) according to the traditions of the sequent calculus.

7 Eager Pairs

Eager pairs are another positive type and therefore quite analogous to the unit type. To evaluate an eager pair $\langle e_1, e_2\rangle$ we have to evaluate e_1 and e_2 and then form the pair of their values. The corresponding process $[\![\langle e_1, e_2\rangle]\!]\ d$ allocates two new destinations, d_1 and d_2 and launches two new processes, one to compute and write the value of e_1 to d_1 and the other to write the value of e_2 to d_2. Without waiting for these two finish, we already can form the pair $\langle d_1, d_2\rangle$ and write it to the original destination d.

$$\begin{aligned}[\![\langle e_1, e_2\rangle]\!]\, d = {} & x_1 \leftarrow [\![e_1]\!]\, d_1 \ ; \\ & x_2 \leftarrow [\![e_2]\!]\, d_2 \ ; \\ & d.\langle x_1, x_2\rangle\end{aligned}$$

There is a lot of parallelism in this translation: not only can the translations of e_1 and e_2 can proceed in parallel (without possibility of interference), but any process waiting for a value in the cell d will be able to proceed immediately, before either of these two finish. In the previously introduced parallel pairs on the midterm the synchronization point is earlier, namely when the pair of the values of e_1 and e_2 is formed.

$$\begin{aligned}[\![\mathsf{case}\ e\ (\langle x_1, x_2\rangle \Rightarrow e')]\!]\, d = {} & x \leftarrow [\![e]\!]\, x \ ; \\ & \mathsf{case}\ x\ (\langle x_1, x_2\rangle \Rightarrow [\![e']\!]\, d)\end{aligned}$$

In the rule just above we note that the occurrences of x_1 and x_2 in e' will be translated using the rule for variables.

The new process construct $d.\langle c_1, c_2\rangle$ simply writes the pair $\langle c_1, c_2\rangle$ to destination d and case reads the pair from memory and matches it against the pattern $\langle x_1, x_2\rangle$.

$$\begin{array}{l}\mathsf{proc}\ c\ (c.\langle c_1, c_2\rangle) \mapsto \mathsf{!cell}\ c\ \langle c_1, c_2\rangle \\ \mathsf{!cell}\ c\ \langle c_1, c_2\rangle, \mathsf{proc}\ d\ (\mathsf{case}\ c\ (\langle x_1, x_2\rangle \Rightarrow P)) \mapsto \mathsf{proc}\ d\ ([c_1/x_1, c_2/x_2]P)\end{array}$$

Typing rules generalize the unit types in interesting ways. We start with $d.\langle d_1, d_2\rangle$. This writes to d, which must therefore have type $\tau_1 \times \tau_2$. It must be able to read destinations d_1 and d_2 which must have types τ_1 and τ_2, respectively.

$$\frac{c_1 : \tau_1 \in \Gamma \quad c_2 : \tau_2 \in \Gamma}{\Gamma \vdash d.\langle c_1, c_2\rangle :: (d : \tau_1 \times \tau_2)}\ \times R^0$$

We use the superscript 0 because this is a nonstandard rule—the usual rule of the sequent calculus has 2 premises, while this rule only checks membership in the typing context. Note that c_1 and c_2 could be equal if $\tau_1 = \tau_2$.

The rule for the new case construct mirrors the usual rule for expressions, but using destinations.

$$\frac{c : \tau_1 \times \tau_2 \in \Gamma \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash P :: (d : \sigma)}{\Gamma \vdash \mathsf{case}\ c\ (\langle x_1, x_2\rangle \Rightarrow P) :: (d : \sigma)}\ \times L$$

We close this section with the corresponding logical rules.

$$\frac{}{\Gamma \vdash 1}\ 1R^0 \qquad \frac{1 \in \Gamma \quad \Gamma \vdash C}{\Gamma \vdash C}\ 1L$$

$$\frac{A, B \in \Gamma}{\Gamma \vdash A \times B}\ {\times}R^0 \qquad \frac{A \times B \in \Gamma \quad \Gamma, A, B \vdash C}{\Gamma \vdash C}\ {\times}L$$

All the types considered in this lecture are *positive types*, so they are "eager" in the sense that a value only contains other values and that the destructors are case constructs.

8 Summary

Since we have changed our notation a few times, we summarize the translation and the transition rules.

$$[\![x]\!]\ d = d \leftarrow x$$

$$[\![\langle\,\rangle]\!]\ d = d.\langle\,\rangle$$
$$\begin{aligned}[\![\mathsf{case}\ e\ (\langle\,\rangle \Rightarrow e')]\!]\ d = {} & d_1 \leftarrow [\![e]\!]\ d_1\ ; \\ & \mathsf{case}\ d_1\ (\langle\,\rangle \Rightarrow [\![e']\!]\ d)\end{aligned}$$

$$\begin{aligned}[\![\langle e_1, e_2\rangle]\!]\ d = {} & d_1 \leftarrow [\![e_1]\!]\ d_1\ ; \\ & d_2 \leftarrow [\![e_2]\!]\ d_2\ ; \\ & d.\langle d_1, d_2\rangle\end{aligned}$$
$$\begin{aligned}[\![\mathsf{case}\ e_0\ (\langle x_1, x_2\rangle \Rightarrow e')]\!]\ d = {} & d_0 \leftarrow [\![e_0]\!]\ d_0\ ; \\ & \mathsf{case}\ d_0\ (\langle x_1, x_2\rangle \Rightarrow [\![e']\!]\ d)\end{aligned}$$

$\mathsf{proc}\ d'\ (x \leftarrow P\ ;\ Q) \mapsto \mathsf{proc}\ d\ ([d/x]P), \mathsf{cell}\ d\ _, \mathsf{proc}\ d'\ ([d/x]Q)$ (d fresh) (alloc/spawn)

$!\mathsf{cell}\ c\ W, \mathsf{proc}\ d\ (d \leftarrow c), \mathsf{cell}\ d\ _ \mapsto \mathsf{cell}\ d\ W$ (copy)

$\mathsf{proc}\ d\ (d.\langle\,\rangle), \mathsf{cell}\ d\ _ \mapsto\ !\mathsf{cell}\ d\ \langle\,\rangle$ $(1R^0)$

$!\mathsf{cell}\ c\ \langle\,\rangle, \mathsf{proc}\ d\ (\mathsf{case}\ c\ (\langle\,\rangle \Rightarrow P)) \mapsto \mathsf{proc}\ d\ P$ $(1L)$

$\mathsf{proc}\ d\ (d.\langle c_1, c_2\rangle), \mathsf{cell}\ d\ _ \mapsto\ !\mathsf{cell}\ d\ \langle c_1, c_2\rangle$ $({\times}R^0)$

$!\mathsf{cell}\ c\ \langle c_1, c_2\rangle, \mathsf{proc}\ d\ (\mathsf{case}\ c\ (\langle x_1, x_2\rangle \Rightarrow P)) \mapsto \mathsf{proc}\ d\ ([c_1/x_1, c_2/x_2]P)$ $({\times}L)$

9 Streamlining the Positive Types

In the presentation of this lecture we notice commonality between the cases and we can refactor it so all positive (eager) types are treated uniformly. We define (omitting $\exists\alpha.\,\tau$ for simplicity):

$$
\begin{array}{llcll}
\text{Positive types} & \tau & ::= & 1 \mid \tau_1 \times \tau_2 \mid \sum_{i\in I}(i : \tau_i) \mid \rho\alpha.\,\tau & \\
\text{Small values} & V & ::= & \langle\,\rangle \mid \langle a_1, a_2\rangle \mid i \cdot a \mid \mathsf{fold}\ a & \\
\text{Continuations} & K & ::= & (\langle\,\rangle \Rightarrow P) \mid (\langle x_1, x_2\rangle \Rightarrow P) \mid (i \cdot x_i \Rightarrow P_i)_{i\in I} \mid (\mathsf{fold}\ x \Rightarrow P) & \\
\text{Processes} & P & ::= & x \leftarrow P \,;\, Q & (allocate/spawn) \\
 & & \mid & c \leftarrow d & (copy) \\
 & & \mid & d.V & (write) \\
 & & \mid & \mathsf{case}\ c\ K & (read/match) \\
\text{Configurations} & \mathcal{C} & ::= & \mathsf{proc}\ d\ P \mid \mathsf{!cell}\ c\ V \mid \cdot \mid \mathcal{C}_1, \mathcal{C}_2 &
\end{array}
$$

We only have four transition rules for configurations, in addition to explaining how values are matched against continuations.

$$
\begin{array}{rlcl}
 & \mathsf{proc}\ d\ (x \leftarrow P \,;\, Q) & \mapsto & \mathsf{proc}\ c\ ([c/x]P), \mathsf{proc}\ d\ ([c/x]Q) \\
\mathsf{!cell}\ c\ V, & \mathsf{proc}\ d\ (d \leftarrow c) & \mapsto & \mathsf{!cell}\ d\ V \\
 & \mathsf{proc}\ d\ (d.V) & \mapsto & \mathsf{!cell}\ d\ V \\
\mathsf{!cell}\ c\ V, & \mathsf{proc}\ d\ (\mathsf{case}\ c\ K) & \mapsto & \mathsf{proc}\ d\ (V \triangleright K)
\end{array}
$$

$$
\begin{array}{rclcl}
\langle\,\rangle & \triangleright & (\langle\,\rangle \Rightarrow P) & = & P \\
\langle c_1, c_2\rangle & \triangleright & (\langle x_1, x_2\rangle \Rightarrow P) & = & [c_1/x_1, c_2/x_2]P \\
k \cdot c & \triangleright & (i \cdot x_i \Rightarrow P_i)_{i\in I} & = & [c/x_k]P_k \\
\mathsf{fold}\ c & \triangleright & (\mathsf{fold}\ x \Rightarrow P) & = & [c/x]P
\end{array}
$$

10 Example: Writing a Value

A closed value in the our language of expressions is translated to a *program* that will create a representation of this value in memory. As such, memory and the contents of the its cells is observable since it represents the outcome of the computation. As an example, consider

$bin = \rho bin.\,(\mathbf{b0} : bin) + (\mathbf{b1} : bin) + (\mathbf{e} : bin)$
$one = \mathsf{fold}\ \mathbf{b1} \cdot \mathsf{fold}\ \mathbf{e} \cdot \langle\,\rangle$

We work out the translation of $[\![one]\!]$ in stages.

$$
\begin{aligned}
[\![one]\!]\, c_0 &= x_1 \leftarrow [\![\mathbf{b1} \cdot \mathsf{fold}\ \mathbf{e} \cdot \langle\,\rangle]\!]\, x_1 \ ; \\
&\quad\ x_0.(\mathsf{fold}\ b1) \\
&= x_1 \leftarrow (\ x_2 \leftarrow [\![\mathsf{fold}\ \mathbf{e} \cdot \langle\,\rangle]\!]\, x_2 \ ; \\
&\qquad\qquad\quad x_1.(\mathbf{b_1} \cdot x_2)\ )\ ; \\
&\quad\ c_0.(\mathsf{fold}\ x_1) \\
&= x_1 \leftarrow (\ x_2 \leftarrow (\ x_3 \leftarrow [\![\mathbf{e} \cdot \langle\,\rangle]\!]\, x_3 \ ; \\
&\qquad\qquad\qquad\qquad x_2.(\mathsf{fold}\ x_3)\ )\ ; \\
&\qquad\qquad\quad x_1.(\mathbf{b_1} \cdot x_2)\ )\ ; \\
&\quad\ c_0.(\mathsf{fold}\ x_1) \\
&= x_1 \leftarrow (\ x_2 \leftarrow (\ x_3 \leftarrow (\ x_4 \leftarrow [\![\langle\,\rangle]\!]\, x_4 \ ; \\
&\qquad\qquad\qquad\qquad\qquad\quad x_3.(\mathbf{e} \cdot x_4)\ )\ ; \\
&\qquad\qquad\qquad\qquad x_2.(\mathsf{fold}\ x_3)\ )\ ; \\
&\qquad\qquad\quad x_1.(\mathbf{b_1} \cdot x_2)\ )\ ; \\
&\quad\ c_0.(\mathsf{fold}\ x_1) \\
&= x_1 \leftarrow (\ x_2 \leftarrow (\ x_3 \leftarrow (\ x_4 \leftarrow x_4.\langle\,\rangle \ ; \\
&\qquad\qquad\qquad\qquad\qquad\quad x_3.(\mathbf{e} \cdot x_4)\ )\ ; \\
&\qquad\qquad\qquad\qquad x_2.(\mathsf{fold}\ x_3)\ )\ ; \\
&\qquad\qquad\quad x_1.(\mathbf{b_1} \cdot x_2)\ )\ ; \\
&\quad\ c_0.(\mathsf{fold}\ x_1)
\end{aligned}
$$

This program will allocate four fresh cells, say, $c_1, \ldots, c_4$, for $x_1, \ldots, x_4$ and fill them with the indicated small values, in no particular order. The resulting final configuration will be

$$
\begin{aligned}
&\mathsf{proc}\ c_0\ ([\![one]\!]\, c_0) \\
&\mapsto^* \mathsf{!cell}\ c_4\ \langle\,\rangle, \mathsf{!cell}\ c_3\ (\mathbf{e} \cdot c_4), \mathsf{!cell}\ c_2\ (\mathsf{fold} \cdot c_3), \mathsf{!cell}\ c_1\ (\mathbf{b1} \cdot\ c_2), \mathsf{!cell}\ c_0\ (\mathsf{fold} \cdot c_1)
\end{aligned}
$$

The following law of *associativity* (not justified here, because we do not have a simple theory of process equivalence) allows us to rewrite this process into a more readable form. In order to apply the equivalence, some restrictions need to be placed on variable occurrences, so we indicate permissible references to variables for each process in parentheses.

$$x \leftarrow (y \leftarrow P(y)\ ;\ Q(x,y))\ ;\ R(x) \equiv y \leftarrow P(y)\ ;\ (x \leftarrow Q(x,y)\ ;\ R(x))$$

Applying this multiple times to re-associate the cuts to the left we get

$$
\begin{aligned}
[\![one]\!]\, c_0 &= x_4 \leftarrow x_4.\langle\,\rangle\ ; \\
&\quad\ x_3 \leftarrow x_3.(\mathbf{e} \cdot x_4)\ ; \\
&\quad\ x_2 \leftarrow x_2.(\mathsf{fold}\ x_3)\ ; \\
&\quad\ x_1 \leftarrow x_1.(\mathbf{b_1} \cdot x_2)\ ; \\
&\quad\ c_0.(\mathsf{fold}\ x_1)
\end{aligned}
$$

11 Preservation and Progress

In order to understand preservation and progress, we should understand the typing of configurations. Recall that we have the following judgments:

Small values $\Gamma \vdash V : \tau$

Processes $\Gamma \vdash P :: (d : \tau)$

where the context Γ contains destinations or addresses of cells even at runtime. That is, values V are not necessarily closed as they were in our expression language, but may reference other cells.

When we type a configuration, we write

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

where both Γ and Δ contain types for addresses. The ones in Γ can be *used* in $\mathcal{C}$, which means they can be read by processes or references in cells. The addresses in Δ are *provided* by the configuration, which means they may be written by a process in $\mathcal{C}$, defined by a cell in $\mathcal{C}$, or they already occur in Γ. With this understanding we obtain the following rules.

$$\frac{\Gamma \vdash P :: (c : \tau)}{\Gamma \vdash \mathsf{proc}\ c\ P :: (\Gamma, c : \tau)}\ \mathsf{tp/proc} \qquad \frac{\Gamma \vdash V : \tau}{\Gamma \vdash \mathsf{cell}\ c\ V :: (\Gamma, c : \tau)}\ \mathsf{tp/cell}$$

$$\frac{}{\Gamma \vdash (\cdot) :: \Gamma}\ \mathsf{tp/empty} \qquad \frac{\Gamma \vdash \mathcal{C}_1 :: \Gamma_1 \quad \Gamma_1 \vdash \mathcal{C}_2 :: \Gamma_2}{\Gamma \vdash (\mathcal{C}_1, \mathcal{C}_2) :: \Gamma_2}\ \mathsf{tp/join}$$

We see the rules are arranged so that $\Gamma \vdash \mathcal{C} :: \Delta$ implies that $\Gamma \subseteq \Delta$. In the preservation theorem we need to account for the possibility that a new cell is allocated which would then appear in Δ' with its type but not in Δ.

While configurations are not explicitly ordered, a typing derivation imposes some ordering constraints. In particular, a cell (or the writer of a cell), always precedes a reader of a cell in the left-to-right order of the typing derivation.x

In this lecture we only state progress and preservation; we may come back later to prove them when our language is complete.

Theorem 1 (Preservation) *If* $\Gamma \vdash \mathcal{C} :: \Delta$ *and* $\mathcal{C} \mapsto \mathcal{C}'$ *then* $\Gamma \vdash \mathcal{C} :: \Delta'$ *for some* $\Delta' \supseteq \Delta$.

To state progress, we should reflect on what plays the role of a *value* in our usual formulation of progress. But it turns out to be easy: it is a configuration consisting entirely of cells and no processes. We call such a configuration *final*. Clearly, such a configuration cannot take a step. The usual notion of a *closed expression* that we start with is replace by a configuration that does not rely (that is, may read from) and external addresses.

Theorem 2 (Progress) *If* $\cdot \vdash \mathcal{C} :: \Delta$ *then either* $\mathcal{C} \mapsto \mathcal{C}'$ *or* $\mathcal{C}$ *is final.*

References

[CPWW02] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

[CS09] Iliano Cervesato and Andre Scedrov. Relating state-based and process-based concurrency through linear logic. *Information and Computation*, 207(10):1044–1077, October 2009.

[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.

[PP20] Klaas Pruiksma and Frank Pfenning. Back to futures. *CoRR*, abs/2002.04607, February 2020.

[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.

Lecture Notes on Negative Types

15-814: Types and Programming Languages
Frank Pfenning

Lecture 20
Tuesday, November 10, 2020

1 Introduction

We continue the investigation of shared memory concurrency by adding *negative types*. In our language so far they are functions $\tau \to \sigma$, lazy pairs $\tau \mathbin{\&} \sigma$, and universal types $\forall \alpha.\, \tau$.

2 Review of Positives

We review the types so far, with a twist: we annotate every address that we write to with a superscript^W and every address we read from with a superscript^R.

$$
\begin{array}{llllll}
\text{Processes} & P & ::= & x \leftarrow P \,;\, Q & & \text{allocate/spawn} \\
 & & \mid & x^W \leftarrow y^R & & \text{copy} \\
 & & \mid & x^W.\langle\,\rangle & \mid \mathsf{case}\ x^R\ (\langle\,\rangle \Rightarrow P) & (1) \\
 & & \mid & x^W.\langle y, z\rangle & \mid \mathsf{case}\ x^R\ (\langle y, z\rangle \Rightarrow P) & (\times) \\
 & & \mid & x^W.(j \cdot y) & \mid \mathsf{case}\ x^R\ (i \cdot y \Rightarrow P_i)_{i \in I} & (+) \\
 & & \mid & x^W.\mathsf{fold}\ y & \mid \mathsf{case}\ x^R\ (\mathsf{fold}\ y \Rightarrow P) & (\rho) \\
\text{Small Values} & V & ::= & \langle\,\rangle \mid \langle a_1, a_2\rangle \mid j \cdot a \mid \mathsf{fold}\ a & & \\
\text{Configurations} & \mathcal{C} & ::= & \cdot \mid \mathcal{C}_1, \mathcal{C}_2 \mid \mathsf{proc}\ d\ P \mid \mathsf{!cell}\ c\ V & &
\end{array}
$$

The configurations are unordered and we think of "," as an associative and commutative operator with unit ".". Since we have changed our notation a few times, we summarize the translation and the transition rules.

$$[\![x]\!]\, d = d^W \leftarrow x^R$$

$$[\![\langle\,\rangle]\!]\, d = d^W.\langle\,\rangle$$
$$[\![\mathsf{case}\ e\ (\langle\,\rangle \Rightarrow e')]\!]\, d = x \leftarrow [\![e]\!]\, x\ ;\ \mathsf{case}\ x^R\ (\langle\,\rangle \Rightarrow [\![e']\!]\, d)$$

$$[\![\langle e_1, e_2\rangle]\!]\, d = x_1 \leftarrow [\![e_1]\!]\, x_1\ ;\ x_2 \leftarrow [\![e_2]\!]\, x_2\ ;\ d^W.\langle x_1, x_2\rangle$$

$$[\![\mathsf{case}\ e\ (\langle x_1, x_2\rangle \Rightarrow e')]\!]\, d = x \leftarrow [\![e]\!]\, x\ ;\ \mathsf{case}\ x^R\ (\langle x_1, x_2\rangle \Rightarrow [\![e']\!]\, d)$$

$$[\![j \cdot e]\!]\, d = x \leftarrow [\![e]\!]\, x\ ;\ d^W.(j \cdot x)$$

$$[\![\mathsf{case}\ e\ (i \cdot x \Rightarrow e_i)_{i\in I}]\!]\, d = x \leftarrow [\![e]\!]\, x\ ;\ \mathsf{case}\ x^R\ (i \cdot x \Rightarrow [\![e_i]\!]\, d)_{i\in I}$$

$$[\![\mathsf{fold}\ e]\!]\, d = x \leftarrow [\![e]\!]\, x\ ;\ d^W.(\mathsf{fold}\ x)$$

$$[\![\mathsf{case}\ e\ (\mathsf{fold}\ y \Rightarrow e')]\!]\, d = x \leftarrow [\![e]\!]\, x\ ;\ \mathsf{case}\ x^R\ (\mathsf{fold}\ y \Rightarrow [\![e']\!]\, d)$$

To show the computation rules for *configurations* we refactor the specifications, separating out *continuations* K.

$$\begin{array}{llcll}
\text{Continuations} & K & ::= & (\langle\,\rangle \Rightarrow P) \mid (\langle x_1, x_2\rangle \Rightarrow P) \mid (i \cdot x_i \Rightarrow P_i)_{i\in I} \mid (\mathsf{fold}\ x \Rightarrow P) & \\
\text{Processes} & P & ::= & x \leftarrow P\ ;\ Q & (allocate/spawn) \\
 & & \mid & c \leftarrow d & (copy) \\
 & & \mid & d^W.V & (write) \\
 & & \mid & \mathsf{case}\ c^R\ K & (read/match)
\end{array}$$

We only have four transition rules for configurations, in addition to explaining how values are matched against continuations.

$$\begin{array}{rlcl}
 & \mathsf{proc}\ d\ (x \leftarrow P\ ;\ Q) & \mapsto & \mathsf{proc}\ c\ ([c/x]P), \mathsf{proc}\ d\ ([c/x]Q) \quad (c\ \text{fresh}) \\
!\mathsf{cell}\ c\ V, & \mathsf{proc}\ d\ (d \leftarrow c) & \mapsto & !\mathsf{cell}\ d\ V \\
 & \mathsf{proc}\ d\ (d.V) & \mapsto & !\mathsf{cell}\ d\ V \\
!\mathsf{cell}\ c\ V, & \mathsf{proc}\ d\ (\mathsf{case}\ c\ K) & \mapsto & \mathsf{proc}\ d\ (V \triangleright K)
\end{array}$$

$$\begin{array}{rcrcl} \langle\,\rangle & \triangleright & (\langle\,\rangle \Rightarrow P) & = & P \\ \langle c_1, c_2\rangle & \triangleright & (\langle x_1, x_2\rangle \Rightarrow P) & = & [c_1/x_1, c_2/x_2]P \\ k \cdot c & \triangleright & (i \cdot x_i \Rightarrow P_i)_{i\in I} & = & [c/x_k]P_k \\ \mathsf{fold}\ c & \triangleright & (\mathsf{fold}\ x \Rightarrow P) & = & [c/x]P \end{array}$$

3 Functions

As the first negative type we consider function $\tau \to \sigma$. How do we translate an abstraction $\lambda x.\, e$? The translation must actually take *two* arguments: one is the original argument x, the other is the destination where the result of the functional call should be written to. And the process $[\![\lambda x.\, e]\!]\, d$ must write the translation of the function to destination d.

Before we settle on the syntax for this, consider how to translate function application.

$$\begin{array}{rl} [\![e_1\, e_2]\!]\, d = & x_1 \leftarrow [\![e_1]\!]\, x_1 \ ; \\ & x_2 \leftarrow [\![e_2]\!]\, x_2 \ ; \\ & \boxed{} \end{array}$$

How should we complete this translation?

We know that after $[\![e_1]\!]\, x_1$ has completed the cell x_1 will contain *a function of two arguments*. The *first* argument is the original argument, which we find in x_2 after $[\![e_2]\!]\, x_2$ has completed. The *second* argument is the destination for the result of the function application, which is d. So we get:

$$\begin{array}{rl} [\![e_1\, e_2]\!]\, d = & x_1 \leftarrow [\![e_1]\!]\, x_1 \ ; \\ & x_2 \leftarrow [\![e_2]\!]\, x_2 \ ; \\ & x_1^R.\langle x_2, d\rangle \end{array}$$

This looks just like eager pairs, except that we *read* from x_1 instead of writing to it. To retain the analogy, we write the translation of a function using case, but *writing* the (single) branch of the case expression to memory.

$$[\![\lambda x.\, e]\!]\, d = \mathsf{case}\ d^W\ (\langle x, y\rangle \Rightarrow [\![e]\!]\, y)$$

The transition rules for these new constructs just formalize the explanation.

$$\begin{array}{lr} \mathsf{proc}\ d\ (\mathsf{case}\ d^W\ (\langle x, y\rangle \Rightarrow P)) \mapsto \mathsf{!cell}\ d\ (\langle x, y\rangle \Rightarrow P) & ({\to}R) \\ \mathsf{!cell}\ c\ (\langle x, y\rangle \Rightarrow P), \mathsf{proc}\ d\ (c^R.\langle c_1, d\rangle) \mapsto \mathsf{proc}\ d\ ([c_1/x, d/y]P) & ({\to}L^0) \end{array}$$

As an example, we consider the expression $(\lambda x.\, x)\, \langle\,\rangle$.

$$
\begin{aligned}
[\![(\lambda x.\, x)\,\langle\,\rangle]\!]\, d_0 &= x_1 \leftarrow [\![\lambda x.\, x]\!]\, x_1 \;; \\
&\quad\; x_2 \leftarrow [\![\langle\,\rangle]\!]\, x_2 \;; \\
&\quad\; x_1^R.\langle x_2, d_0\rangle \\
&= x_1 \leftarrow \mathsf{case}\; x_1^W\; (\langle x, y\rangle \Rightarrow [\![x]\!]\, y) \;; \\
&\quad\; x_2 \leftarrow x_2^W.\langle\,\rangle \;; \\
&\quad\; x_1^R.\langle x_2, d_0\rangle \\
&= x_1 \leftarrow \mathsf{case}\; x_1^W\; (\langle x, y\rangle \Rightarrow y^W \leftarrow x^R) \;; \\
&\quad\; x_2 \leftarrow x_2^W.\langle\,\rangle \;; \\
&\quad\; x_1^R.\langle x_2, d_0\rangle
\end{aligned}
$$

Let's execute the final process from with the initial destination d_0.

$$
\begin{array}{ll}
& \mathsf{proc}\; d_0\; (x_1 \leftarrow \mathsf{case}\; x_1^W\; (\ldots) \;;\; x_2 \leftarrow x_2^W.\langle\,\rangle \;;\; \ldots) \\
\mapsto & \mathsf{proc}\; d_1\; (\mathsf{case}\; d_1^W\; (\langle x, y\rangle \Rightarrow y^W \leftarrow x^R)), \\
& \mathsf{proc}\; d_0\; (x_2 \leftarrow x_2^W.\langle\,\rangle \;;\; d_1^R.\langle x_2, d_0\rangle) \\
\mapsto^2 & !\mathsf{cell}\; d_1\; (\langle x, y\rangle \Rightarrow y^W \leftarrow x^R), \\
& \mathsf{proc}\; d_2\; (d_2^W.\langle\,\rangle), \\
& \mathsf{proc}\; d_0\; (d_1^R.\langle d_2, d_0\rangle) \\
\mapsto^2 & !\mathsf{cell}\; d_1\; (\langle x, y\rangle \Rightarrow y^W \leftarrow x^R), \\
& !\mathsf{cell}\; d_2\; \langle\,\rangle, \\
& \mathsf{proc}\; d_0\; (d_0^W \leftarrow d_2^R) \qquad\qquad \text{(from } [d_2/x, d_0/y](y^W \leftarrow x^R)\text{)} \\
\mapsto & !\mathsf{cell}\; d_1\; (\langle x, y\rangle \Rightarrow y^W \leftarrow x^R), \\
& !\mathsf{cell}\; d_2\; \langle\,\rangle \\
& !\mathsf{cell}\; d_0\; \langle\,\rangle
\end{array}
$$

In the final configuration we have cell d_0 holding the final result $\langle\,\rangle$, which is indeed the result of evaluating $(\lambda x.\, x)\ \langle\,\rangle$. We also have some newly allocated intermediate destinations d_1 and d_2 that are preserved, but could be garbage collected if we only retain the cells that are reachable from the initial destination d_0 which now holds the final value.

4 Store Revisited

In our table of process expression, two things stand out. One is that functions are exactly like pairs, except that the role of reads and writes are reversed.

The other is that a cell may now contain something of the form $(\langle y, z\rangle \Rightarrow P)$.

$$
\begin{array}{llllll}
\text{Processes} & P & ::= & x \leftarrow P \; ; Q & & \text{allocate/spawn} \\
& & \mid & x^W \leftarrow y^R & & \text{copy} \\
& & \mid & x^W.\langle\,\rangle & \mid \mathsf{case}\ x^R\ (\langle\,\rangle \Rightarrow P) & (1) \\
& & \mid & x^W.\langle y, z\rangle & \mid \mathsf{case}\ x^R\ (\langle y, z\rangle \Rightarrow P) & (\times) \\
& & \mid & x^W.(j \cdot y) & \mid \mathsf{case}\ x^R\ (i \cdot y \Rightarrow P_i)_{i \in I} & (+) \\
& & \mid & x^W.\mathsf{fold}(y) & \mid \mathsf{case}\ x^R\ (\mathsf{fold}(y) \Rightarrow P) & (\rho) \\
\hline
& & \mid & x^R.\langle y, z\rangle & \mid \mathsf{case}\ x^W\ (\langle y, z\rangle \Rightarrow P) & (\rightarrow)
\end{array}
$$

We can refactor this into a more uniform presentation, even though not all of the syntactically legal forms have corresponding types in the current language.

$$
\begin{array}{llllll}
\text{Processes} & P & ::= & x \leftarrow P \; ; Q & & \text{allocate/spawn} \\
& & \mid & x^w \leftarrow y^R & & \text{copy} \\
& & \mid & x^W.V & \mid \mathsf{case}\ x^R\ K & (1, \times, +, \rho) \\
& & \mid & x^R.V & \mid \mathsf{case}\ x^W\ K & (\rightarrow)
\end{array}
$$

$$
\begin{array}{llll}
\text{Small values} & V & ::= & \langle\,\rangle \mid \langle a_1, a_2\rangle \mid i \cdot a \mid \mathsf{fold}\ a \\
\text{Continuations} & K & ::= & (\langle\,\rangle \Rightarrow P) \mid (\langle x_1, x_2\rangle \Rightarrow P) \mid (i \cdot x_i \Rightarrow P_i)_{i \in I} \mid (\mathsf{fold}\ x \Rightarrow P) \\
\\
\text{Cell contents} & W & ::= & V \mid K \\
\\
\text{Configurations} & \mathcal{C} & ::= & \cdot \mid \mathcal{C}_1, \mathcal{C}_2 \mid \mathsf{proc}\ d\ P \mid !\mathsf{cell}\ c\ W
\end{array}
$$

There is now a legitimate concern that the contents of cells in memory is no longer "small", because a program P could be of arbitrary size. At a lower level of abstraction, continuations would probably be implemented as *closures*, that is, a pairs consisting of an environment and the address of code to be executed. The translation to get us to this form is called *closure conversion*, which we might discuss in a future lecture. For now, we are content with the observation that, yes, we are violating a basic principle of fixed-size storage and that it can be mitigated (but is not completely solved) through the introduction of closures.

In our example of $(\lambda x.\, x)\ \langle\,\rangle$ the continuation has the form $(\langle x, y\rangle \Rightarrow y^W \leftarrow x^R)$ which is a closed process. This can be directly compiled to a function that takes two addresses x and y and writes the contents of x into y. So at least in this special case the contents of the cell d_1 could simply be the address of this piece of code.

The symmetry between eager pairs (positive) and functions (negative) stems from the property that in logic we have $A \vdash B \supset C$ if and only if $A \times B \vdash C$ (where $\times$ is a particular form of conjunction). Or, we can chalk it up to the isomorphism $\tau \to (\sigma \to \rho) \cong (\tau \times \sigma) \to \rho$: an arrow on the right behaves like a product on the left.

One can ask if similarly symmetric constructors exists for 1, $+$, and ρ and the answer is yes. It turns out that lazy records are symmetric to sums and there is a type $\bot$ that is symmetric to 1 (see Exercises 1 and 2). There may even be a lazy analogue of recursive types that exhibits the same kind of symmetry and maybe useful to model so-called corecursive types (see Exercise 3).

We postpone discussion on the typing of process expression, cells, and configurations until the next lecture when we consider analogues of the progress and preservation theorems.

5 Typing

Before writing an example, it may be helpful to revisit the typing in its factored form. We separate out the positives, since the typing for the negatives is not quite as uniform as one might expect.

To type the contents of cells directly, we have the judgment $\Gamma \vdash V : \tau$ for positive τ.

$$\frac{}{\Gamma \vdash \langle\,\rangle : 1}\ \mathsf{val/unit} \qquad \frac{y : \tau \in \Gamma \qquad z : \sigma \in \Gamma}{\Gamma \vdash \langle y, z \rangle : \tau \times \sigma}\ \mathsf{val/prod}$$

$$\frac{(j \in I) \quad y : \tau_j \in \Gamma}{\Gamma \vdash j \cdot y : \sum_{i \in I}(i : \tau_i)}\ \mathsf{val/sum} \qquad \frac{y : [\rho\alpha.\, \tau/\alpha]\tau \in \Gamma}{\Gamma \vdash \mathsf{fold}\ y : \rho\alpha.\, \tau}\ \mathsf{val/fold}$$

Process typing for the positives is now unified, but we still separate out the negative with some special-purpose rules. For positive types τ we also have a judgment to verify that a value $V : \tau$ is matched against a suitable continuation, $\Gamma \vdash \tau \triangleright K :: (z : \sigma)$.

$$\frac{\Gamma \vdash V : \tau}{\Gamma \vdash x^W.V :: (x : \tau)}\ \mathsf{write/pos} \qquad \frac{x : \tau \in \Gamma \qquad \Gamma \vdash \tau \triangleright K :: (z : \sigma)}{\Gamma \vdash \mathsf{case}\ x^R\ K :: (z : \sigma)}\ \mathsf{read/pos}$$

$$\frac{\Gamma \vdash P :: (z : \sigma)}{\Gamma \vdash 1 \triangleright (\langle\,\rangle \Rightarrow P) :: (z : \sigma)}\ \mathsf{m/unit} \qquad \frac{\Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash P :: (z : \sigma)}{\Gamma \vdash \tau_1 \times \tau_2 \triangleright (\langle x_1, x_2 \rangle \Rightarrow P) : (z : \sigma)}\ \mathsf{m/prod}$$

$$\frac{(\text{for all } i \in I) \quad \Gamma, y : \tau_i \vdash P_i :: (z : \sigma)}{\Gamma \vdash \sum_{i \in I}(i : \tau_i) \triangleright (i(y) \Rightarrow P_i) :: (z : \sigma)}\ \mathsf{m/sum} \qquad \frac{\Gamma, y : [\rho\alpha.\, \tau/\alpha]\tau \vdash P :: (z : \sigma)}{\Gamma \vdash \rho\alpha.\, \tau \triangleright (\mathsf{fold}(y) \Rightarrow P) :: (z : \sigma)}\ \mathsf{m/rho}$$

For the negative types (here only functions), we have somewhat more specific rules. They arise, because for the type $\tau \to \sigma$ the types τ and σ are on different sides of the turnstile.

$$\frac{x : \tau \to \sigma \in \Gamma \quad y : \tau \in \Gamma}{\Gamma \vdash x^R.\langle y, z\rangle :: (z : \sigma)}\ \mathsf{read/arrow} \qquad \frac{\Gamma, y : \tau \vdash P :: (z : \sigma)}{\Gamma \vdash \mathsf{case}\ x^W\ (\langle y, z \rangle \Rightarrow P) :: (x : \tau \to \sigma)}\ \mathsf{write/arrow}$$

6 Example: A Pipeline

As a simple example for concurrency in this language we consider setting up a (very small) pipeline. We consider a sequence of bits

$$\mathit{bits} = \rho\alpha.\, (\mathbf{b0} : \alpha) + (\mathbf{b1} : \alpha) + (\mathbf{e} : 1)$$

(which also happen to be isomorphic to binary numbers). We assume there is a process $\mathit{flip} : \mathit{bits} \to \mathit{bits}$ that just flips every bit. We will write this during the next lecture; for this lecture the goal is to compose two such processes in a pipeline.

Assume there is a cell

$$!\mathsf{cell}\ \mathit{flip}\ K_{\mathit{flip}} : \mathit{bits} \to \mathit{bits}$$

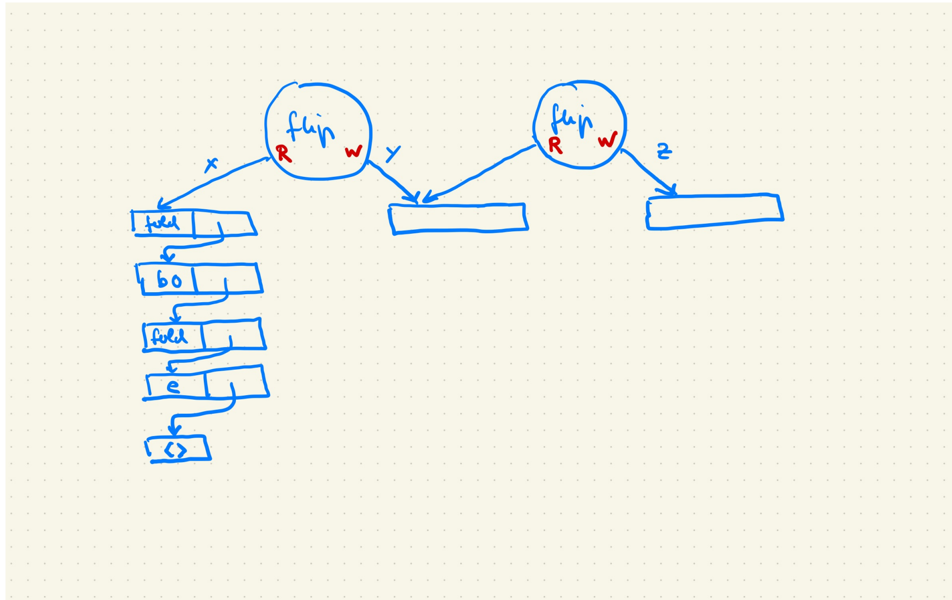
This means that $K_{\mathit{flip}} = (\langle x, y\rangle \Rightarrow P)$ where $x : \mathit{bits}$ is address of the argument and $y : \mathit{bits}$ is the destination for the result.

Then we can compose two of these as

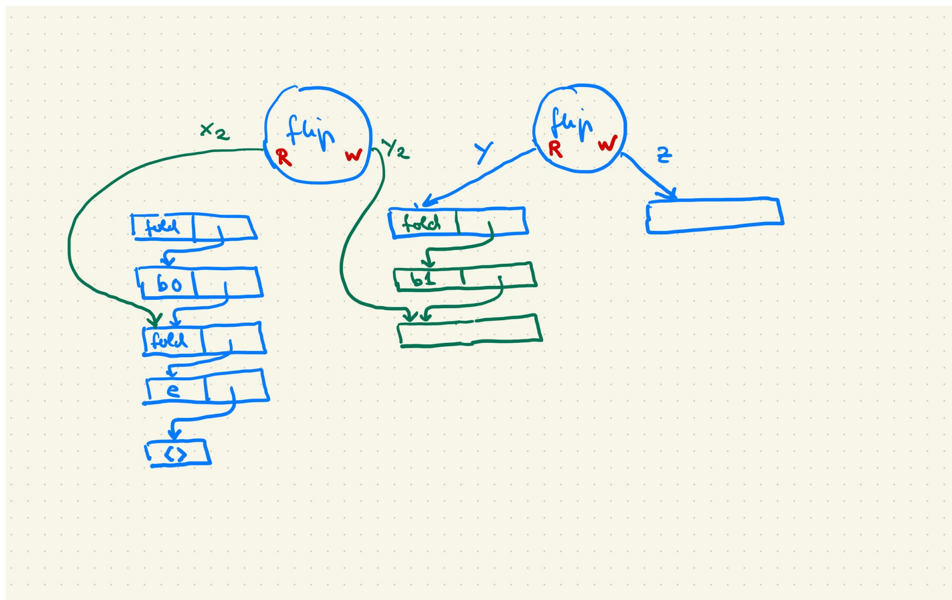
$$\begin{aligned} K_{\mathit{flip2}} = \langle x, z \rangle \Rightarrow & \\ & y \leftarrow \mathit{flip}^R.\langle x, y\rangle \\ & \mathit{flip}^R.\langle y, z\rangle \end{aligned}$$

In the picture below we see the two *flip* processes running, after the code for *kflip2* has executed but neither of these has taken any action yet. The process on the left reads from x and writes to y while the process on the right reads from y and writes to z. The destinations y and z have been allocated but

have not yet been written to. Cell x contains the sample input, which is the memory representation of fold $(\mathbf{b0} \cdot \text{fold } (\mathbf{e} \ \cdot \langle\,\rangle))$.

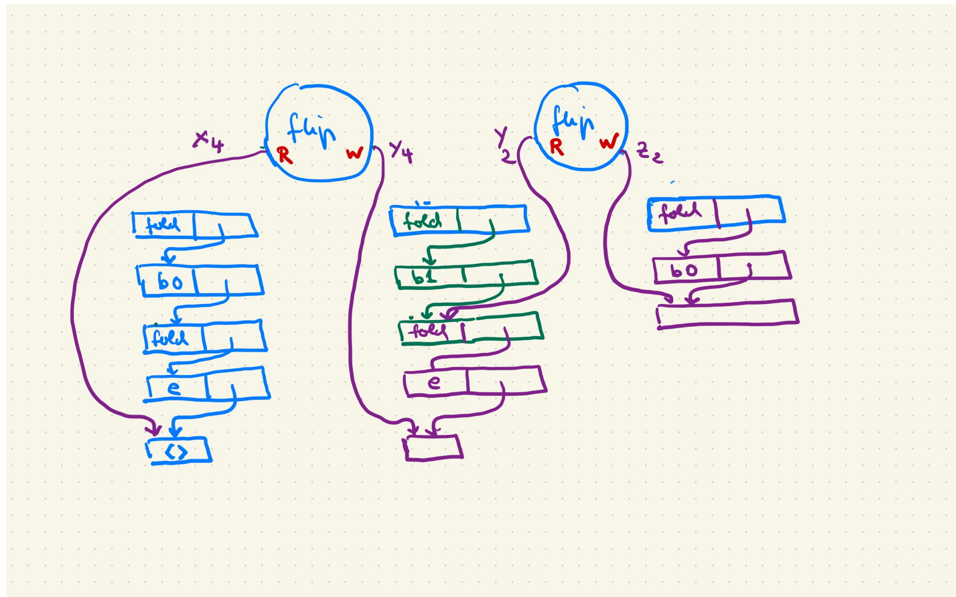


The left process now reads along x and allocates and writes along y. After it runs for a few steps, we might reach the following situation:



The green part here is the new part compared to the previous configuration. It should be clear how each of the two processes translates into a proc object, while each filled cell corresponds to a cell object. The empty cells are addresses that have been allocated, but not yet written to, so they are not explicit in the configuration.

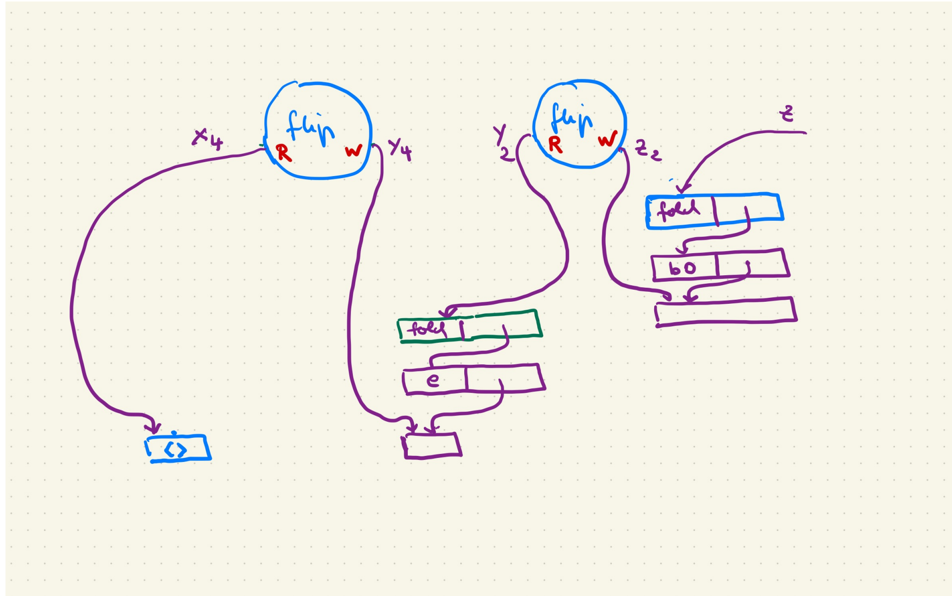
The two processes also run in parallel, which is how they form a pipeline. For example, after a few more steps we might reach the configuration (with the purple part being new):



The right process here lags behind the left one, which is possible since the semantics here is not synchronous. A cell can be read as soon as it is filled, but it may not be read immediately while other computations take place.

If we knew that the left process was the only reader along x (and any cells reachable from it) we could "garbage-collect" the cells that are no longer accessible and the situation would look as follows (assuming here

some process not shown could read the output z).



In the next lecture we will write the code for *flip* that can exhibit the shown behavior. In a future lecture we will consider a type system that tracks if cells have unique readers which will allow the eager deallocation of cells that have been read.

Exercises

Exercise 1 For lazy records (as a generalization of lazy pairs) we introduce the following syntax in our language of expressions:

$$\begin{array}{lcl} \text{Types} & ::= & \ldots \mid \&_{i \in I}(i : \tau_i) \\ \text{Expressions} & ::= & \ldots \mid \langle\!| i \Rightarrow e_i |\!\rangle_{i \in I} \mid e \cdot j \end{array}$$

1. Give the typing rules and the dynamics (stepping rules) for the new constructs.
2. Extend the translation $[\![e]\!]\, d$ to encompass the new constructs. Your process syntax should expose the duality between eager sums and lazy records.
3. Extend the transition rules of the store-based dynamics to the new constructs. The translated form may permit more parallelism than the

original expression evaluation, but when scheduled sequentially they should have the same behavior (which you do not need to prove).

4. Show the typing rules for the new process constructs.

Exercise 2 Explore what the rules and meaning of $\bot$ as the formal dual of 1 in the process language should be, including whichever of the following you find make sense. If something does not make sense somehow, please explain.

1. Write out the new forms of process expressions.
2. Provide the store-based dynamics for the new process expressions.
3. Show the typing rules for the new process expressions.
4. Reverse-engineer new functional expressions in our original language so they translate to your new process expression. Show the rules for typing and stepping the new constructs.
5. Summarize and discuss what you found.

Exercise 3 In our expression language the fold e constructor for elements of recursive type is eager. Explore a new *lazy* ravel e constructor which has type $\delta\alpha.\,\tau$, providing:

1. Typing rules for ravel and a corresponding destructor (presumably an unravel or case construct).
2. Stepping rules for the new forms of expressions.
3. A translation from the new forms of expressions to processes, extending the language of processes as needed
4. Typing rules for the new forms of processes.
5. Transition rules for the new forms of processes.

These reference rules depart from the lecture notes in two minor points: (1) typing of cells is simplified by reducing it to the typing of processes that would write such a cell, and (2) in the dynamics we do not use persistent cells, but explicitly carry ephemeral cells from the left-hand side to the right-hand side of each rule that reads from memory.

Abstract Syntax

Types	τ	$::=$	$\tau_1 \to \tau_2 \mid \&_{i\in I}(i : \tau_i) \mid \tau_1 \times \tau_2 \mid 1 \mid \sum_{i\in I}(i : \tau_i) \mid \rho\alpha.\,\tau$	
Contexts	Γ	$::=$	$\cdot \mid \Gamma, x : \tau$	(all variables distinct)
Processes	P	$::=$	$x \leftarrow P \;;\; Q$	allocate/spawn
		$\mid$	$x^w \leftarrow y^R$	copy
		$\mid$	$x^W.V \mid \mathsf{case}\ x^R\ K$	$(1, \times, +, \rho)$
		$\mid$	$x^R.V \mid \mathsf{case}\ x^W\ K$	$(\to, \&)$

Small values	V	$::=$	$\langle\,\rangle \mid \langle a_1, a_2\rangle \mid i \cdot a \mid \mathsf{fold}\ a$
Continuations	K	$::=$	$(\langle\,\rangle \Rightarrow P) \mid (\langle x_1, x_2\rangle \Rightarrow P) \mid (i \cdot x_i \Rightarrow P_i)_{i\in I} \mid (\mathsf{fold}\ x \Rightarrow P)$
Cell contents	W	$::=$	$V \mid K$
Configurations	$\mathcal{C}$	$::=$	$\cdot \mid \mathcal{C}_1, \mathcal{C}_2 \mid \mathsf{proc}\ d\ P \mid \mathsf{cell}\ c\ W$

Judgments

$\Gamma \vdash P :: (z : \sigma)$	process P reads from Γ and writes to $z : \sigma$
$\Gamma \vdash \mathcal{C} :: \Delta$	configuration $\mathcal{C}$ reads from Γ and writes to Δ
$\Gamma \vdash \mathcal{C}$ *final*	configuration $\mathcal{C}$ is final (consists only of cells)

Theorems

Preservation. If $\Gamma \vdash \mathcal{C} :: \Delta$ and $\mathcal{C} \mapsto \mathcal{C}'$ then $\Gamma \vdash \mathcal{C}' :: \Delta'$ for some $\Delta' \supseteq \Delta$.

Progress. If $\cdot \vdash \mathcal{C} :: \Delta$ then either $\mathcal{C} \mapsto \mathcal{C}'$ for some $\mathcal{C}'$ or $\mathcal{C}$ *final*.

Statics, Allocate and Copy

$$\frac{\Gamma \vdash P :: (x : \tau) \quad \Gamma, x : \tau \vdash Q :: (z : \sigma)}{\Gamma \vdash (x \leftarrow P \;;\; Q) :: (z : \sigma)}\ \mathsf{tp/alloc} \qquad \frac{y : \tau \in \Gamma}{\Gamma \vdash x \leftarrow y :: (x : \tau)}\ \mathsf{tp/copy}$$

Statics, Positive Types

$$\frac{}{\Gamma \vdash x^W.\langle\,\rangle :: (x : 1)}\ \text{w/unit} \qquad \frac{x : 1 \in \Gamma \quad \Gamma \vdash P :: (z : \sigma)}{\Gamma \vdash \mathsf{case}\ x^R\ (\langle\,\rangle \Rightarrow P) :: (z : \sigma)}\ \text{r/unit}$$

$$\frac{y : \tau \in \Gamma \qquad z : \sigma \in \Gamma}{\Gamma \vdash x^W.\langle y, z\rangle :: (x : \tau \times \sigma)}\ \text{w/pair} \qquad \frac{x : \tau_1 \times \tau_2 \in \Gamma \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash P :: (z : \sigma)}{\Gamma \vdash \mathsf{case}\ x^R\ (\langle x_1, x_2\rangle \Rightarrow P) : (z : \sigma)}\ \text{r/pair}$$

$$\frac{(j \in I) \quad y : \tau_j \in \Gamma}{\Gamma \vdash x^W.(j \cdot y) :: (x : \sum_{i \in I}(i : \tau_i))}\ \text{w/tag} \qquad \frac{x : \sum_{i \in I}(i : \tau_i) \in \Gamma \quad \Gamma, y : \tau_i \vdash P_i :: (z : \sigma)\ (\forall i \in I)}{\Gamma \vdash \mathsf{case}\ x^R\ (i \cdot y \Rightarrow P_i)_{i \in I} :: (z : \sigma)}\ \text{r/tag}$$

$$\frac{y : [\rho\alpha.\,\tau/\alpha]\tau \in \Gamma}{\Gamma \vdash x^W.(\mathsf{fold}\ y) :: (x : \rho\alpha.\,\tau)}\ \text{w/fold} \qquad \frac{x : \rho\alpha.\,\tau \in \Gamma \quad \Gamma, y : [\rho\alpha.\,\tau/\alpha]\tau \vdash P :: (z : \sigma)}{\Gamma \vdash \mathsf{case}\ x^R\ (\mathsf{fold}\ y \Rightarrow P) :: (z : \sigma)}\ \text{r/fold}$$

Statics, Negative Types

$$\frac{\Gamma, y : \tau \vdash P :: (z : \sigma)}{\Gamma \vdash \mathsf{case}\ x^W\ (\langle y, z\rangle \Rightarrow P) :: (x : \tau \to \sigma)}\ \text{w/fun} \qquad \frac{x : \tau \to \sigma \in \Gamma \quad y : \tau \in \Gamma}{\Gamma \vdash x^R.\langle y, z\rangle :: (z : \sigma)}\ \text{r/fun}$$

$$\frac{\Gamma \vdash P_i :: (z_i : \tau_i) \quad (\text{for all } i \in I)}{\Gamma \vdash \mathsf{case}\ x^W\ (i \cdot z_i \Rightarrow P_i)_{i \in I} :: (x : \&_{i \in I}(i : \tau_i))}\ \text{w/record} \qquad \frac{x : \&_{i \in I}(i : \tau_i) \in \Gamma \quad j \in I}{\Gamma \vdash x^R.(j \cdot z) :: (z : \tau_j)}\ \text{r/record}$$

Statics, Configurations

$$\frac{\Gamma \vdash P :: (c : \tau)}{\Gamma \vdash \mathsf{proc}\ c\ P :: (\Gamma, c : \tau)}\ \text{tp/proc}$$

$$\frac{\Gamma \vdash c^W.V :: (c : \tau)}{\Gamma \vdash \mathsf{cell}\ c\ V :: (\Gamma, c : \tau)}\ \text{tp/cell/val} \qquad \frac{\Gamma \vdash \mathsf{case}\ c^W\ K :: (c : \tau)}{\Gamma \vdash \mathsf{cell}\ c\ K :: (\Gamma, c : \tau)}\ \text{tp/cell/cont}$$

$$\frac{}{\Gamma \vdash (\cdot) :: \Gamma}\ \text{tp/empty} \qquad \frac{\Gamma \vdash \mathcal{C}_1 :: \Gamma_1 \quad \Gamma_1 \vdash \mathcal{C}_2 :: \Gamma_2}{\Gamma \vdash (\mathcal{C}_1, \mathcal{C}_2) :: \Gamma_2}\ \text{tp/join}$$

$$\frac{}{\Gamma \vdash (\cdot)\ \mathit{final}}\ \text{fin/empty} \qquad \frac{\Gamma \vdash \mathcal{C}\ \mathit{final}}{\Gamma \vdash (\mathcal{C}, \mathsf{cell}\ c\ W)\ \mathit{final}}\ \text{fin/cell}$$

Dynamics

$$
\begin{array}{llcll}
 & \mathsf{proc}\ d\ (x \leftarrow P \ ;\ Q) & \mapsto & \mathsf{proc}\ c\ ([c/x]P), \mathsf{proc}\ d\ ([c/x]Q) & (\text{alloc/spawn, } c \text{ fresh}) \\
\mathsf{cell}\ c\ W, & \mathsf{proc}\ d\ (d^W \leftarrow c^R) & \mapsto & \mathsf{cell}\ c\ W, \mathsf{cell}\ d\ W & (\text{copy}) \\
 & & & & \\
 & \mathsf{proc}\ d\ (d^W.V) & \mapsto & \mathsf{cell}\ d\ V & (\text{write: } \times, 1, +, \rho) \\
\mathsf{cell}\ c\ V, & \mathsf{proc}\ d\ (\mathsf{case}\ c^R\ K) & \mapsto & \mathsf{cell}\ c\ V, \mathsf{proc}\ d\ (V \triangleright K) & (\text{read: } \times, 1, +, \rho) \\
 & & & & \\
 & \mathsf{proc}\ d\ (\mathsf{case}\ d^W\ K) & \mapsto & \mathsf{cell}\ d\ K & (\text{write: } \rightarrow, \&) \\
\mathsf{cell}\ c\ K, & \mathsf{proc}\ d\ (c^R.V) & \mapsto & \mathsf{cell}\ c\ K, \mathsf{proc}\ d\ (V \triangleright K) & (\text{read: } \rightarrow, \&)
\end{array}
$$

$$
\begin{array}{rcllcl}
\langle\,\rangle & \triangleright & (\langle\,\rangle \Rightarrow P) & = & P \\
\langle c_1, c_2\rangle & \triangleright & (\langle x_1, x_2\rangle \Rightarrow P) & = & [c_1/x_1, c_2/x_2]P \\
j \cdot c & \triangleright & (i \cdot x_i \Rightarrow P_i)_{i \in I} & = & [c/x_j]P_j \\
\mathsf{fold}\ c & \triangleright & (\mathsf{fold}\ x \Rightarrow P) & = & [c/x]P
\end{array}
$$

Lecture Notes on Concurrent Programming

15-814: Types and Programming Languages
Frank Pfenning

Lecture 21
Thursday, November 12, 2020

1 Introduction

In this lecture we explore concurrent programming in our language of processes through two different examples: a *pipeline* (as started in the last lecture) and *fork/join* parallelism with *map/reduce*. Before we get to these, a small example to get used to the representation of functions in the concurrent language.

2 Simple Functions

We want to define a process

$$\mathit{curry} : ((\tau \times \sigma) \to \rho) \to (\tau \to (\sigma \to \rho))$$

Its implementation will immediately write a continuation to memory.

$$[\![\mathit{curry}]\!]\, d = \mathsf{case}\ d^W\ (\langle f, g\rangle \Rightarrow \boxed{})$$

So the real essence of this function is in the continuation

$K_{\mathit{curry}} = (\langle f, g\rangle \Rightarrow P)$

where P reads from $f : (\tau \times \sigma) \to \rho$ and writes to $g : \tau \to (\sigma \to \rho)$. The result is immediately a λ-expression, which means that as a process we write another continuation to memory.

$K_{\mathit{curry}} = (\langle f, g\rangle \Rightarrow \mathsf{case}\ g^W\ (\langle x, h\rangle \Rightarrow \boxed{}))$

Here $x : \tau$, the argument to g. Again, we write a function, this time one that takes $y : \sigma$ and a destination $r : \rho$ for the final result.

$$K_{\mathit{curry}} = (\langle f, g\rangle \Rightarrow \mathsf{case}\ g^W\ (\langle x, h\rangle \Rightarrow \mathsf{case}\ h^W\ (\langle y, r\rangle \Rightarrow \boxed{})))$$

At this point we have x and y in hand, so we can pair them up and pass the pair to f. But, wait! We cannot actually construct a pair and pass it. Instead, we need to allocate a cell to hold the pair $\langle x, y\rangle$ and pass its address p to g. In addition, we also have to pass an address as the destination of f, but that is just r. That is:

$$\begin{aligned} K_{\mathit{curry}} = \langle f, g\rangle \Rightarrow \mathsf{case}\ g^W\ (\langle x, h\rangle \Rightarrow \mathsf{case}\ h^W\ (\langle y, r\rangle \Rightarrow \\ p \leftarrow p^W.\langle x, y\rangle\ ; \\ f^R.\langle p, r\rangle)) \end{aligned}$$

Similarly, we start for a function in the other direction:

$$K_{\mathit{uncurry}} : (\tau \to (\sigma \to \rho)) \to ((\tau \times \sigma) \to \rho)$$

$$K_{\mathit{uncurry}} = \langle g, f\rangle \Rightarrow \mathsf{case}\ f^w\ (\langle p, r\rangle \Rightarrow \boxed{})$$

Now we have $p : \tau \times \sigma$ and the destination $r : \rho$. We read out the component from the cell at address p.

$$K_{\mathit{uncurry}} = \langle g, f\rangle \Rightarrow \mathsf{case}\ f^w\ (\langle p, r\rangle \Rightarrow \mathsf{case}\ p^R\ (\langle x, y\rangle \Rightarrow \boxed{}))$$

Now we need to pass $x : \tau$ to g, but we also need a destination. The one we have ($r : \rho$) does not work, so we need to allocate a new one, call it h.

$$\begin{aligned} K_{\mathit{uncurry}} = \langle g, f\rangle \Rightarrow \mathsf{case}\ f^w\ (\langle p, r\rangle \Rightarrow \mathsf{case}\ p^R\ (\langle x, y\rangle \Rightarrow \\ h \leftarrow g^R.\langle x, h\rangle\ ; \\ \boxed{})) \end{aligned}$$

At this point we can just read the function at $h : \sigma \to \rho$ and pass it $y : \sigma$ and the destination $r : \rho$.

$$\begin{aligned} K_{\mathit{uncurry}} = \langle g, f\rangle \Rightarrow \mathsf{case}\ f^w\ (\langle p, r\rangle \Rightarrow \mathsf{case}\ p^R\ (\langle x, y\rangle \Rightarrow \\ h \leftarrow g^R.\langle x, h\rangle\ ; \\ h^R.\langle y, r\rangle)) \end{aligned}$$

Neither of these processes has much intrinsic concurrency, but the arguments, for example, to f and g are addresses, and the value to be stored at these addresses may not yet have been written. We can see that neither x nor y are read by these functions, just passed through. As mentioned previously, this is the characteristic of *futures*.

3 A Bit-Flipping Pipeline

More interesting from the concurrency point of view is the bit-flipping pipeline. Recall from the last lecture the type of sequences of bits

$$\textit{bits} = \rho\alpha.\,(\mathbf{b0} : \alpha) + (\mathbf{b1} : \alpha) + (\mathbf{e} : 1)$$

We start by writing an ordinary function $\textit{flip} : \textit{bits} \to \textit{bits}$ that flips the input bits from 0 to 1 and vice versa.

$$\begin{array}{l}
\textit{flip} : \textit{bits} \to \textit{bits} \\
\textit{flip} = \mathsf{fix}\,\textit{flip}.\,\lambda x.\,\mathsf{case}\ x\ (\,\mathsf{fold}\ (\mathbf{b0} \cdot x') \Rightarrow \mathsf{fold}\ (\mathbf{b1} \cdot (\textit{flip}\ x')) \\
\qquad\qquad\qquad\qquad\quad \mid \mathsf{fold}\ (\mathbf{b1} \cdot x') \Rightarrow \mathsf{fold}\ (\mathbf{b0} \cdot (\textit{flip}\ x')) \\
\qquad\qquad\qquad\qquad\quad \mid \mathsf{fold}\ (\mathbf{e} \cdot u) \Rightarrow \mathsf{fold}\ (\mathbf{b1} \cdot \mathsf{fold}\ (\mathbf{e} \cdot u))\,)
\end{array}$$

In the remainder of this section we make a small syntactic simplification which makes the code much shorter without loss of information content: we skip reading and writing the *fold* messages. We can think of the types as "silently unfolded", or postulate an elaboration pass over the program that inserts suitable fold constructors and fold patterns. The *flip* function then would look like

$$\begin{array}{l}
\textit{flip} : \textit{bits} \to \textit{bits} \\
\textit{flip} = \mathsf{fix}\,\textit{flip}.\,\lambda x.\,\mathsf{case}\ x\ (\,\mathbf{b0} \cdot x' \Rightarrow \mathbf{b1} \cdot (\textit{flip}\ x') \\
\qquad\qquad\qquad\qquad\quad \mid \mathbf{b1} \cdot x' \Rightarrow \mathbf{b0} \cdot (\textit{flip}\ x') \\
\qquad\qquad\qquad\qquad\quad \mid \mathbf{e} \cdot u \Rightarrow \mathbf{b1} \cdot (\mathbf{e} \cdot u)\,)
\end{array}$$

Instead of translating this function we write it directly to a process. For this purpose we have to decide how to handle recursion. There seem to be two solutions:

1. We add a process $\mathsf{fix}\ f.\,P$ which transitions to $[\mathsf{fix}\ f.\,P/f]P$. This is entirely straightforward but requires process substitution in the dynamics.

2. We allow recursively defined processes

   $!\mathsf{cell}\ \textit{flip}\ K_{\textit{flip}}$

   where $K_{\textit{flip}}$ refers back to its own cell with address *flip* to encode a recursive call.

We choose the latter option, for variety, even though it would require more complicated typing rules for configurations.

It remains to define K_{flip}.

$$\begin{array}{lll} K_{\mathit{flip}} = \langle x, y \rangle \Rightarrow \mathsf{case}\ x\ (& \mathbf{b0} \cdot x' \Rightarrow & \boxed{} \\ & \mid \mathbf{b1} \cdot x' \Rightarrow & \boxed{} \\ & \mid \mathbf{e} \cdot u \Rightarrow & \boxed{}\,) \end{array}$$

In the first branch, we have to allocate a fresh cell y' for the output and make a recursive call to fill it. We can also write $\mathbf{b1} \cdot y'$ to y.

$$\begin{array}{lll} K_{\mathit{flip}} = \langle x, y \rangle \Rightarrow \mathsf{case}\ x\ (& \mathbf{b0} \cdot x' \Rightarrow & y' \leftarrow \mathit{flip}^R.\langle x', y' \rangle \ ; \\ & & y^W.(\mathbf{b1} \cdot y') \\ & \mid \mathbf{b1} \cdot x' \Rightarrow & \boxed{} \\ & \mid \mathbf{e} \cdot u \Rightarrow & \boxed{}\,) \end{array}$$

The branch for $\mathbf{b1}$ is symmetric to the first one.

$$\begin{array}{lll} K_{\mathit{flip}} = \langle x, y \rangle \Rightarrow \mathsf{case}\ x\ (& \mathbf{b0} \cdot x' \Rightarrow & y' \leftarrow \mathit{flip}^R.\langle x', y' \rangle \ ; \\ & & y^W.(\mathbf{b1} \cdot y') \\ & \mid \mathbf{b1} \cdot x' \Rightarrow & y' \leftarrow \mathit{flip}^R.\langle x', y' \rangle \ ; \\ & & y^W.(\mathbf{b0} \cdot y') \\ & \mid \mathbf{e} \cdot u \Rightarrow & \boxed{}\,) \end{array}$$

In the last case, we allocate a new cell to hold $\mathbf{e} \cdot u$ and share $u : 1$ between the input and the output. Alternatively, we could avoid the inner allocation and just share x itself, or we could copy u also.

$$\begin{array}{lll} K_{\mathit{flip}} = \langle x, y \rangle \Rightarrow \mathsf{case}\ x\ (& \mathbf{b0} \cdot x' \Rightarrow & y' \leftarrow \mathit{flip}^R.\langle x', y' \rangle \ ; \\ & & y^W.(\mathbf{b1} \cdot y') \\ & \mid \mathbf{b1} \cdot x' \Rightarrow & y' \leftarrow \mathit{flip}^R.\langle x', y' \rangle \ ; \\ & & y^W.(\mathbf{b0} \cdot y') \\ & \mid \mathbf{e} \cdot u \Rightarrow & y' \leftarrow y'^W.(\mathbf{e} \cdot u) \\ & & y^W.(\mathbf{b1} \cdot y')\,) \end{array}$$

As shown in the last lecture, we can compose two *flip* processes into a pipeline as follows:

$$\begin{array}{l} K_{\mathit{flip2}} = \langle x, z \rangle \Rightarrow \\ \qquad y \leftarrow \mathit{flip}^R.\langle x, y \rangle \\ \qquad \mathit{flip}^R.\langle y, z \rangle \end{array}$$

You may look back at the diagrams to visualize how the two processes work together, effectively communicating via the shared location y, which then becomes y', y'', etc. as the computation progresses and recursive calls are mad in both of them.

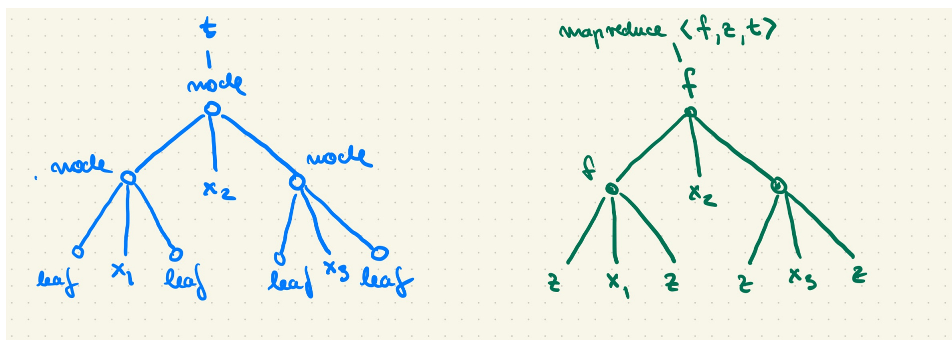
Under a sequential interpretation, where $x \leftarrow P \; ; \; Q$ waits until P has written to destination x before Q starts executing, all recursive calls in *flip* would have to be finished before the first bit of output is written. When we compose two, the inner one has to finish entirely, writing out the whole sequence of bits before the outer one can start. This is the behavior of the functional $\lambda x. \mathit{flip}\,(\mathit{flip}\,x)$ where the intermediate destination y and the final destination z remain unnamed.

4 Map/Reduce

As a second example with significant concurrency we consider the popular *mapreduce*. We use a function f to *map* over a tree, reducing it to a value. In many applications the tree may not be explicit, but emerge dynamically from the way the data are distributed. As a consequence we require our function f to be associative and have a unit z, which may stand in for the absence of data. See Exercise 3 for a version where trees are represented differently. We define tree as a family of types, indexed by the type of element, even though we have not formally introduced this into our language.

$$\mathit{tree}\ \alpha = \rho t.\, (\mathbf{node} : t \times \alpha \times t) + (\mathbf{leaf} : 1)$$

We can picture the action of *mapreduce* as an *iteration* over this kind of tree. We supply a function f to "replace" every node and constant z to stand in for every leaf, as pictured in green in the image below.



We can read off the type

$\mathit{mapreduce} : [\forall\alpha.\,\forall\beta.]\,(\beta \times \alpha \times \beta \to \beta) \times \beta \times \mathit{tree}\ \alpha \to \beta$

As before, we imagine a cell !cell *mapreduce* $K_{\mathit{mapreduce}}$ and define $K_{\mathit{mapreduce}}$. We have put the type quantifiers on α and β in brackets because we haven't explicitly considered how to handle these in our concurrent language. Instead, we think of *mapreduce* as a family of functions indexed by α and β.

$$K_{\mathit{mapreduce}} = \langle\langle f, z, t\rangle, y\rangle \Rightarrow \boxed{}$$

Here we have $f : \beta \times \alpha \times \beta \to \beta$, $z : \beta$, and $t : \mathit{tree}\ \alpha$, with the destination $y : \beta$. We have taken a small shortcut here by using pattern matching: in fully official syntax, the right-hand side would start as

$$K_{\mathit{mapreduce}} = \langle p, y\rangle \Rightarrow \mathsf{case}\ p\ (\langle f, q\rangle \Rightarrow \mathsf{case}\ q\ (\langle z, t\rangle \Rightarrow \boxed{}))$$

but this is more verbose and more difficult to read. Back to the previous version. We start with a case analysis over t: is it a leaf or a node? If it is a leaf, we just copy z to the destination y.

$$\begin{array}{l} K_{\mathit{mapreduce}} = \langle\langle f, z, t\rangle, y\rangle \Rightarrow \\ \qquad \mathsf{case}\ t^R\ (\,\mathbf{leaf}\cdot\langle\,\rangle \Rightarrow y^W \leftarrow z^R \\ \qquad\qquad \mid \mathbf{node}\cdot\langle l, x, r\rangle \Rightarrow \boxed{}\,) \end{array}$$

Here, l is the address of the left subtree, x is the element at the node, and r is the address of the right subtree. Now we need to make two recursive calls, on the left and right subtrees. In order to make these calls we need to allocate two new cells y_1 and y_2 to receive the values of these calls and pass them as destinations.

$$\begin{array}{l} K_{\mathit{mapreduce}} = \langle\langle f, z, t\rangle, y\rangle \Rightarrow \\ \qquad \mathsf{case}\ t^R\ (\,\mathbf{leaf}\cdot\langle\,\rangle \Rightarrow y^W \leftarrow z^R \\ \qquad\qquad \mid \mathbf{node}\cdot\langle l, x, r\rangle \Rightarrow \\ \qquad\qquad\quad y_1 \leftarrow \mathit{mapreduce}^R.\langle\langle f, z, l\rangle, y_1\rangle\ ; \\ \qquad\qquad\quad y_2 \leftarrow \mathit{mapreduce}^R.\langle\langle f, z, r\rangle, y_2\rangle\ ; \\ \qquad\qquad\quad \boxed{}\,) \end{array}$$

Note that these two recursive calls proceed concurrently. Finally, we have to invoke the function f on the results from these recursive calls and x, and pass the result to y.

$$
\begin{array}{l}
K_{\textit{mapreduce}} = \langle\langle f, z, t\rangle, y\rangle \Rightarrow \\
\qquad \mathsf{case}\ t^R\ (\ \mathbf{leaf} \cdot \langle\,\rangle \Rightarrow y^W \leftarrow z^R \\
\qquad\qquad \mid \mathbf{node} \cdot \langle l, x, r\rangle \Rightarrow \\
\qquad\qquad\quad y_1 \leftarrow \textit{mapreduce}^R.\langle\langle f, z, l\rangle, y_1\rangle \ ; \\
\qquad\qquad\quad y_2 \leftarrow \textit{mapreduce}^R.\langle\langle f, z, r\rangle, y_2\rangle \ ; \\
\qquad\qquad\quad f^R.\langle\langle y_1, x, y_2\rangle, y\rangle\ )
\end{array}
$$

Again, we have used a short-hand here. In official syntax we have to allocate pairs to hold the first argument to f, so the last line would expand to:

$$
\begin{array}{l}
p_1 \leftarrow p_1^W.\langle x, y_2\rangle \ ; \\
p_2 \leftarrow p_2^W.\langle y_1, p_1\rangle \ ; \\
f^R.\langle p_2, y\rangle
\end{array}
$$

In any case, we can see that no synchronization on y_1 or y_2 occurs until the function f actually needs their values.

5 Recovering Sequentiality

Originally, we thought of our concurrent process language as the result of translating our expression language LAMBDA. However, the result of the translation behaves significantly differently from the source due the pervasive concurrency.

We could just say that we schedule the different processes for taking step in a way that exactly mimics left-to-right sequential execution. Or we can manipulate the translation to enforce sequentiality. Since only cut (an allocate followed by a spawn) creates a new thread of control, this is our main lever to work with. For example, we could have a *sequential cut* $x \Leftarrow P; Q$ which runs P to completion before starting Q. Its semantics might be:

$$\mathsf{proc}\ d\ (x \Leftarrow P; Q) \mapsto \mathsf{proc}\ c\ ([c/x]P), \mathsf{susp}\ c\ d\ ([c/x]Q)$$

with a new semantics object wait with the rule

$$!\mathsf{cell}\ c\ W, \mathsf{susp}\ c\ d\ Q \mapsto \mathsf{proc}\ d\ Q$$

where the new semantic object $\mathsf{susp}\ c\ d\ Q$ represents a suspected process, waiting for the cell c to be written to. Since writing to c is the last action of a process with destination c, this will prevent Q from computing until P has finished. Moreover, Q will never have to synchronize on c because it is guaranteed to have already been written to.

It is then easy to prove, by induction on transition sequences, that there is at most one (unsuspended) process in a configuration if we start with just one process. Also, the concurrent semantics can *simulate* the sequential one by always making particular choices, but not the other way around.

In a language with both sequential and concurrent cut we can work mostly sequentially and occasionally spawn a process to run concurrently. This is the idea behind *futures* where the expression $\mathsf{future}\ e$ immediately returns a destination d that the evaluation of e eventually writes to. Attempts to read the future will block until the value has been written.

Exercises

Exercise 1 Consider the translation

$$[\![\mathsf{fix}\, f.\, e]\!]\, d = \mathsf{case}\ d^W\ (\langle x, y\rangle \Rightarrow [d/f][\![e]\!]\, y)$$

in which d is written to but also (potentially) read from in the translation of $[d/f][\![e]\!]\, y$. Execution of this process may therefore create circular references in the configuration.

(i) Give an example where the translation behaves *incorrectly* with respect to the dynamics of the expression $\mathsf{fix}\, f.\, e$ in LAMBDA.

(ii) Give an example where circular references arise but behave *correctly* with respect to the dynamics in the source.

(iii) From your examples, conjecture a restriction of the general translation so the result behaves correctly.

(iv) Devise new typing rules for processes and configurations such that (a) the translation above is well-typed, as a process, and (b) the typing of configurations is preserved by transitions, and (c) the progress theorem continues to be true. You do not need to prove these properties, but it may be helpful to sketch the proof to yourself to make sure your rules are correct.

Exercise 2 When translating functional fixed point expression to recursively defined processes, we need to account for the fact that processes may be invoked in multiple places with different destinations. We there introduce the notation $x.\, P$ for a process with variable destination x and $(x.\, P)(d)$ for its instantiation to a particular destination. We then translate:

$[\![\mathsf{fix}\, f.\, e]\!]\, d = (x.\, \mathsf{rec}\, f.\, [\![e]\!]\, x)(d)$
where $[\![f]\!]\, c = f(c)$ for every occurrence of f in e.

We also extend the dynamics with the rule

$$\mathsf{proc}\ d\ ((x.\, \mathsf{rec}\, f.\, P)(d)) \mapsto \mathsf{proc}\ d\ ([(x.\, \mathsf{rec}\, f.\, P)/f][d/x]P)$$

(i) Give typing rules for the new forms of processes.

(ii) Provide an implementation of the *flip* process using this representation of recursion.

(iii) Illustrate the key transition steps in the computation of *flip*, showing the plausibility of this translation.

Exercise 3 Consider the type of tree where the information is kept only in the leaves:

$$\mathit{shrub}\ \alpha = \rho t.\, (\mathbf{branch} : t \times t) + (\mathbf{bud} : \alpha)$$

(i) Write a version of *mapreduce* that operates on shrubs and exhibits analogous concurrent behavior. You may use similar shortcuts to the ones we used in our implementation.

(ii) Write processes *forth* and *back* to translate between trees and shrubs while preserving the elements. Do they form an isomorphism? If not, do you see a simple modification to restore an isomorphism?

Exercise 4 The sequential execution in Section 5 is *eager* in the sense that in $x \Leftarrow P\ ;\ Q$, P completes by writing to x before Q starts.

A lazy version, $x \leftharpoondown P\ ;\ Q$ would immediately start Q and suspend P until Q (or some process spawned by it) would try to read from x. We would still like it to be sequential in the sense that at most one process can take a step at any time.

Devise a semantics for $x \leftharpoondown P\ ;\ Q$ that exhibits the desired lazy behavior while remaining sequential. You may introduce new semantic objects or apply some transformation to P and/or Q, but you should strive for the simplest, most elegant solution to keep the dynamics simple.

Lecture Notes on
Lazy Records and Mutable Store

15-814: Types and Programming Languages
Frank Pfenning

Lecture 22
Tuesday, November 17, 2020

1 Introduction

We have moved from a semantics directly on expressions to one that makes memory explicit and supports concurrency (at the discretion of the scheduler or the language designer). Memory is allocated and then written to at most once; after that it may be read many times.

In imperative languages we can also mutate the contents of a memory cell by writing a different value to it. In a functional language, this is typically segregated, either in a monad (as in Haskell) or via a new type of mutable references (as in ML). We pursue here the latter approach because there is a slightly lower conceptual overhead.

Before that, we consider some examples of stream programming, which is a good example of lazy functional programming which is available to use most easily via lazy pairs and (more generally) lazy records.

2 Lazy Records and Streams

A *lazy record* is a generalization of a lazy pair where each alternative has a different label i. They were introduced in Exercise L20.1 with the syntax

$$
\begin{array}{lcl}
\text{Types} & ::= & \ldots \mid \&_{i \in I}(i : \tau_i) \\
\text{Expressions} & ::= & \ldots \mid \langle\!| i \Rightarrow e_i |\!\rangle_{i \in I} \mid e \cdot j
\end{array}
$$

where $\langle\!| i \Rightarrow e_i |\!\rangle_{i \in I}$ has type $\&_{i \in I}(i : \tau_i)$ if each e_i has type τ_i. Similarly, for an e of type $\&_{i \in I}(i : \tau_i)$ the projection onto j (written $e \cdot j$) has type τ_j.

As an example, consider potentially infinite streams *stream* α of elements of some type α may be defined as

$$\textit{stream}\ \alpha = \rho s.\, (\mathbf{hd} : \alpha) \mathbin{\&} (\mathbf{tl} : s)$$

which then satisfies

$$\textit{stream}\ \alpha \cong (\mathbf{hd} : \alpha) \mathbin{\&} (\mathbf{tl} : \textit{stream}\ \alpha)$$

The concrete LAMBDA syntax is quite similar. Since we don't have type constructors in the implementation, we just consider streams of natural numbers which are defined with

```
1 type stream = $stream. ('hd : nat) & ('tl : stream)
```

Next we would like to produce an infinite stream of increasing numbers. The specification is that

$$\textit{up}\ n = n, n+1, n+2, \ldots$$

which we write as

```
1 decl up : nat -> stream
2 defn up = $up. \n. fold (| 'hd => n | 'tl => up (succ n) |)
```

Here we see the concrete syntax `(| 'i1 => e1 | ... | 'in => en |)` for a lazy record with fields `'i1` through `'in`. Laziness of the records is crucial here because otherwise the function $\textit{up}\,\overline{n}$ would never terminate. Indeed, evaluating

```
1 eval s0 = up zero
```

just produces (after 2 evaluation steps) a stream we cannot observe:

```
1 % eval s0 = up zero
2 % 2 evaluation steps
3 decl s0 : stream
4 defn s0 = fold ---
```

Fortunately, we can write a function to observe the first n elements of a stream as a list. For this purpose we define lists, restricting ourselves to the special case of lists of natural numbers.

```
1 type list = $list. ('nil : 1) + ('cons : nat * list)
2 decl nil : list
3 decl cons : nat -> list -> list
4 defn nil = fold 'nil ()
5 defn cons = \x. \l. fold 'cons (x, l)
```

Our specification is now that

$$take\ n\ s = [s_1, \ldots, s_n]$$

where s_i is the ith element of the stream. We start by cases over n, and returning the empty list if n is zero.

```
1 decl take : nat -> stream -> list
2 defn take = $take. \n. \s.
3    case n
4      of ( fold 'zero () => nil
5         | fold 'succ m => ... )
```

In the case where $n = m + 1$ we would like to create a list of length with the first element begin the head of the stream (obtained with `(unfold s).'hd`). The remainder of the list is the result of a recursive call to take m elements from the tail of the stream.

```
1 decl take : nat -> stream -> list
2 defn take = $take. \n. \s.
3    case n
4      of ( fold 'zero () => nil
5       | fold 'succ m => cons ((unfold s).'hd)
6                              (take m ((unfold s).'tl)) )
```

Then, taking the first 5 elements from the stream $0, 1, 2, \ldots$ eis achieved with

```
1 eval l5 = take _5 (up _0)
```

which indeed yields the list $[\overline{0}, \ldots, \overline{4}]$.

As the next programming puzzle we would like to compute the stream of Fibonacci numbers, $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$. The key insight is that we need to remember *two* numbers to generate the next one, generalizing the idea behind *up*. We specify

$$fib\ n\ k = n, k, n + k, k + (n + k), \ldots$$

after which the actual Fibonacci sequence is *fib* $0\ 1$. We implement this function by shifting the second argument k to become the first argument in the recursive call, and the sum $n + k$ to become the new second argument.

```
1 decl fib : nat -> nat -> stream
2 defn fib = $fib. \n. \k.
3    fold (| 'hd => n | 'tl => fib k (plus n k) |)
4
5 eval fib_stream = fib _0 _1
```

Records here are lazy so, as before, `fib_stream` is not observable. But we can test our function by taking the first 10 elements with

```
eval f10 = take (plus _5 _5) fib_stream
```

There are more examples of stream programming in Exercise 2.

3 Object-Oriented Programming

We can also use lazy records to model some idioms from object-oriented programming. Consider an object of type *stack* which can receive two messages: pushing another element onto the stack, or popping an element from the stack. In the latter case, the response is either **none** (the stack is empty) or **some** and the element.

```
type stack = $stack. ('push : nat -> stack)
                   & ('pop :    ('none : stack)
                              + ('some : nat * stack))
```

The implementation of the stack maintains a list in the local state: it adds a new element to the front of the list to implement **push** and deconstructs the list to implement **pop**. Note that the methods of the objects are the components of a lazy record.

```
decl stack_list : list -> stack
defn stack_list = $stack_list. \l. fold
    (| 'push => \x. stack_list (cons x l)
     | 'pop => case l
                 of ( fold 'nil () => 'none (stack_list l)
                    | fold 'cons (x,l') => 'some (x, stack_list l') )
     |)

decl stack_new : stack
defn stack_new = stack_list nil
```

See the file `lazy.cbv` for a illustrative sequence of push and pop operations.

4 Streams and Functions

An excellent question was raised in lecture, namely if any stream can be represented by a function of type $nat \to nat$ and vice versa. Assuming totality of the functions involved, we were able to conjecture

$$nat \to nat \cong stream$$

using the following functions

```
1  decl forth : (nat -> nat) -> stream
2  decl back : stream -> (nat -> nat)
3
4  % forth f = f 0, f 1, f 2, ...
5  % forth' f n = f n, f (n+1), f (n+1), ...
6  decl forth' : (nat -> nat) -> nat -> stream
7  defn forth' = $forth'. \f. \n.
8     fold (| 'hd => f n | 'tl => forth' f (succ n) |)
9  defn forth = \f. forth' f zero
10
11 defn back = $back. \s. \n.
12   case n
13     of ( fold 'zero () => (unfold s).'hd
14        | fold 'succ m => back ((unfold s).'tl) m )
```

Some small examples in `lazy.cbv` seemed to confirm the correctness of these definitions.

5 The Type of Mutable References

Returning to our original goal of this lecture, we now consider mutable references. We will have to depart from the strong logical basis of our language, but the notation and concepts we have developed to describe typing are sufficient to easily capture the statics and dynamics of the new constructs.

We introduce one new type constructor and three new forms of expression into our functional language:

$$\begin{array}{lrcl} \text{Types} & \tau & ::= & \ldots \mid \mathsf{ref}\ \tau \\ \text{Expressions} & e & ::= & \ldots \mid \mathsf{ref}\ e \mid e_1 := e_2 \mid !e \end{array}$$

Operationally, $\mathsf{ref}\ e$ evaluates e to a value v, then creates a new mutable reference m and initializes its value to v. An assignment $e_1 := e_2$ evaluates e_1 to a mutable reference m, then e_2 to a value v_2 and stores v_2 in m. It returns just the unit element, since its principal task is the effect on m. Finally, $!e$ (which has nothing to do with ! to denote persistent semantic objects) evaluates e to a reference m and returns the current value of m. Based on

this description, we type these new expressions as follows

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{ref}\ e : \mathsf{ref}\ \tau}\ \mathsf{tp/ref} \qquad \frac{\Gamma \vdash e_1 : \mathsf{ref}\ \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : 1}\ \mathsf{tp/assign}$$

$$\frac{\Gamma \vdash e : \mathsf{ref}\ \tau}{\Gamma \vdash\ !e : \tau}\ \mathsf{tp/deref}$$

These rules do not fit the previous patterns of constructor and destructors because of the rule for mutation tp/assign.

It seems difficult, if not impossible, to specify the semantics of mutable references directly on expressions in the style we have done before. Fortunately, we already have a semantics with an explicit store so we can update that. The textbook instead generalizes the small-step semantics for expression by adding a single store μ and now stepping $\mu \parallel e \mapsto \mu' \parallel e'$ [Har16, Chapters 34 & 35].

6 Translation to Our Concurrent Language

We exploit the fact we already have a representation of memory in this translation, and only two small twists are necessary. Warning: some of what is below we will later find out is not quite right. We write m for the address of a mutable cell.

$$\begin{aligned}[\![\mathsf{ref}\ e]\!]\, d = {} & m \leftarrow [\![e]\!]\, m\ ; \\ & d^W.\mathsf{addr}(m)\end{aligned}$$

Here, we introduce a new form of value, $\mathsf{addr}(m)$ which denotes the address of a mutable cell, here m. This value is deposited in destination d as required.

Reading from a mutable destination is simple.

$$\begin{aligned}[\![!e]\!]\, d = {} & x \leftarrow [\![e]\!]\, x\ ; \\ & \mathsf{case}\ x^R\ (\mathsf{addr}(m) \Rightarrow d^W \leftarrow m^R)\end{aligned}$$

Finally, mutating a cell. At first we might try

$$\begin{aligned}[\![e_1 := e_2]\!]\, d = {} & x_1 \leftarrow [\![e_1]\!]\, x_1\ ; \\ & x_2 \leftarrow [\![e_2]\!]\, x_2\ ; \\ & \mathsf{case}\ x_1^R\ (\mathsf{addr}(m) \Rightarrow m^W \leftarrow x_2^R) \quad \%\ \textit{bug here!}\end{aligned}$$

The problem here is that the translation of $e_1 := e_2$ is supposed to write to destination d, but does not do so. Recall that we decreed that the assignment should return the unit element, so we might write

$$
\begin{aligned}
[\![e_1 := e_2]\!]\, d = {} & x_1 \leftarrow [\![e_1]\!]\, x_1 \;; \\
& x_2 \leftarrow [\![e_2]\!]\, x_2 \;; \\
& \mathsf{case}\ x_1^R\ (\mathsf{addr}(m) \Rightarrow m^W \leftarrow x_2^R \;; \\
& \qquad\qquad\qquad\qquad d.\langle\,\rangle)
\end{aligned}
$$

However, this requires a version of the copy process that allows a continuation. Let's write this as $m^W \Leftarrow d_2^R$, and we get

$$
\begin{aligned}
[\![e_1 := e_2]\!]\, d = {} & d_1 \leftarrow [\![e_1]\!]\, d_1 \;; \\
& d_2 \leftarrow [\![e_2]\!]\, d_2 \;; \\
& \mathsf{case}\ d_1^R\ (\mathsf{addr}(m) \Rightarrow m^W \Leftarrow d_2^R \;; \\
& \qquad\qquad\qquad\qquad d.\langle\,\rangle)
\end{aligned}
$$

The new process expression has the dynamics

$$
!\mathsf{cell}\ m\ W, !\mathsf{cell}\ c\ W', \mathsf{proc}\ d\ (m^W \Leftarrow c^R \;; P) \mapsto\ !\mathsf{cell}\, m\ W', \mathsf{proc}\ d\ P \qquad \%\ \textit{bug!}
$$

Writing this out, however, we notice a second problem: the cell m has to be ephemeral. If it were persistent, then after this transition m would have two values: W and W'.

We can fix this in two ways. Either we make all cells (mutable or not) ephemeral. This means we have to revisit all the rules so far and make sure cell are not consumed when they are read but carried over. Alternatively, we can make only mutable cells ephemeral and keep all others persistent. Let's use the first approach. We modify the rules at the end of Section L21.4 by dropping the ! everywhere. Where we match against !cell $c\ W$ on the left-hand side, we just replace it by cell $c\ W$ and repeat it on the right-hand side. The rule for the new "write" construct becomes

$$
\mathsf{cell}\ m\ W, \mathsf{cell}\ c\ W', \mathsf{proc}\ d\ (m^W \Leftarrow c^R \;; P) \mapsto \mathsf{cell}\ m\ W', \mathsf{cell}\ c\ W', \mathsf{proc}\ d\ P
$$

For the other approach, see Exercise 1.

7 Race Conditions

In the presence of mutable references, sequential computation proceeds as before, scheduling such that in $x \leftarrow P \;; Q$ the process P completes (and therefore writes to x) before Q starts. This also means that the read and write operations on mutable cells have a well-defined order.

Under the concurrent semantics, however, the picture is more complicated. Consider the following expression:

$$
(\lambda x.\, \langle x := \textit{succ}\ x, \langle x := \textit{succ}\ x, !x\rangle\rangle)\ (\mathsf{ref}\ \textit{zero})
$$

The value of this expression will be

$\langle\langle\,\rangle, \langle\langle\,\rangle, n\rangle\rangle$

where $n \in \{\overline{0}, \overline{1}, \overline{2}\}$. For example, we obtain $\overline{0}$ if $!x$ executes before the increments. Note also that either increment or dereference of the value might have to wait until the initialization of the mutable cell with $\overline{0}$ completes because the body of the function can execute in parallel with the argument.

Despite these difficulties, progress and preservation theorems continue to hold, but it becomes much more difficult to reason about the correctness of programs. Similarly, we don't lose all of parametricity, but logical equality (and, more generally, logical relations) now require *step-indexing* [AM01, TTA+13].

Exercises

Exercise 1 Provide an alternative dynamics for our language with mutable cells, where regular cells become persistent once written, while mutable cells are ephemeral. You may have to introduce some new kinds of semantic objects or some new forms of process expression, or both.

Exercise 2 Write functions on streams as in Section 2 satisfying the specifications below.

(i) *alt* : $\forall\alpha.\, \mathit{stream}\ \alpha \rightarrow \mathit{stream}\ \alpha \rightarrow \mathit{stream}\ \alpha$ which alternates the elements from the two streams, starting with the first element of the first stream.

(ii) *filter* : $\forall\alpha.\, (\alpha \rightarrow \mathit{bool}) \rightarrow \mathit{stream}\ \alpha \rightarrow \mathit{stream}\ \alpha$ which returns the stream with just those elements of the input stream that satisfy the given predicate.

(iii) *map* : $\forall\alpha.\, \forall\beta.\, (\alpha \rightarrow \beta) \rightarrow (\mathit{stream}\ \alpha \rightarrow \mathit{stream}\ \beta)$ which returns a stream with the result of applying the given function to every element of the input stream.

(iv) *diag* : $\forall\alpha.\, \mathit{stream}\ (\mathit{stream}\ \alpha) \rightarrow \mathit{stream}\ \alpha$ which returns a stream consisting of the first element of the first stream, the second element of the second stream, the third element of the third stream, etc.

You may use earlier functions in the definition of later ones and write auxiliary functions as needed.

In the LAMBDA implementation, you may choose the special case that $\alpha = \beta = \mathit{nat}$. In the absence of type constructors in LAMBDA, define types

$$\begin{array}{l} \mathit{stream} = \rho s.\, (\mathbf{hd} : \mathit{nat}) \,\&\, (\mathbf{tl} : s) \\ \mathit{sstream} = \rho ss.\, (\mathbf{hd} : \mathit{stream}) \,\&\, (\mathbf{tl} : ss) \end{array}$$

where the first was already present in Section 2) and the second is needed for part (iv).

Your functions should be such that only as much of the output stream is computed as necessary to obtain a *value* of type $\mathit{stream}\ \alpha$ but not the components contained in the lazy record. For example, among the three definitions below of a stream transducer that adds 1 to every element, only the first definition would be lazy enough. The second definition (*succs*′) would be still terminating, but slighty too eager (for example, we may never access the element at the head of the resulting stream which would have been computed unnecessarily), while the third (succs″) would not even be terminating any more.

$$\begin{array}{l} \mathit{succs} : \mathit{stream\ nat} \to \mathit{stream\ nat} \\ \mathit{succs} = \lambda s.\, \langle\!| \mathtt{hd} \Rightarrow \mathit{succ}\, ((\mathsf{unfold}\ s) \cdot \mathtt{hd}), \mathtt{tl} \Rightarrow \mathit{succs}\, ((\mathsf{unfold}\ s) \cdot \mathtt{tl}) |\!\rangle \\ \mathit{succs}' = \lambda s.\ \mathsf{let}\ x = \mathit{succ}\, ((\mathsf{unfold}\ s) \cdot \mathtt{hd}) \\ \qquad\qquad \mathsf{in}\ \langle\!| \mathtt{hd} \Rightarrow x, \mathtt{tl} \Rightarrow \mathit{succs}'\, ((\mathsf{unfold}\ s) \cdot \mathtt{tl}) \\ \\ \mathit{succs}'' = \lambda s.\ \mathsf{let}\ s' = \mathit{succs}''\, ((\mathsf{unfold}\ s) \cdot \mathtt{tl}) \\ \qquad\qquad \mathsf{in}\ \langle\!| \mathtt{hd} \Rightarrow \mathit{succ}\, ((\mathsf{unfold}\ s) \cdot \mathtt{hd}), \mathtt{tl} \Rightarrow s' |\!\rangle \end{array}$$

Here we have used $\mathsf{let}\ x = e\ \mathsf{in}\ e'$ as syntactic sugar for $(\lambda x.\, e)\, e'$.

Exercise 3 Following the style of object-oriented programming in Section 3 consider the types of queue

$$\begin{array}{rl} \mathit{queue}\ \alpha = \rho s. & (\mathbf{enq} : \alpha \to \mathit{queue}\ \alpha) \\ & \&\, (\mathbf{deq} : \quad (\mathbf{none} : \mathit{queue}\ \alpha) \\ & \qquad\qquad + (\mathbf{some} : \alpha \times \mathit{queue}\ \alpha)) \end{array}$$

(i) Write a function

$$\mathit{reverse} : \forall \alpha.\, \mathit{stack}\ \alpha \to \mathit{stack}\ \alpha$$

that reverses the elements of the given stack.

(ii) Provide an implementation of queues

$$\mathit{queue_stacks} : \forall \alpha.\, \mathit{stack}\ \alpha \to \mathit{stack}\ \alpha \to \mathit{queue}\ \alpha$$

where a queue is represented by a pair of stacks (see below).

(iii) Provide an empty queue

$$queue_new : \forall \alpha.\, queue\ \alpha$$

One of your stacks should be the *input stack*. Elements to be enqueued should be pushed on this input stack. The second stack should be the *output stack*. Elements to be dequeued should be taken from the output stack. If the output stack happens to be empty but some elements remain on the input stack, reverse the input stack to become the new output stack. This technique is sometimes called *functional queues*.

As in Section 3, in the absence of type constructors in LAMBDA you may specialize the types of stacks and queues to $\alpha = nat$.

Exercise 4 Streams, as we have defined them in this lecture, do not memoize their results, which means that if we repeatedly access the tail or head of a stream it may be recalculated each time. Using mutable references, define memoizing streams that avoids this recomputation and illustrate it through some examples.

References

[AM01] Andrew W. Appel and David A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *Transactions on Programming Languages and Systems*, 23(5):657–683, 2001.

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

[TTA+13] Aaron Joseph Turon, Jacob Thamsbord, Amal Ahmed, Lars Birkedal, and Derek Dreyer. Logical relations for fine-grained concurrency. In R. Giacobazzi and R. Cousot, editors, *Symposium on Principles of Programming Languages (POPL'13)*, pages 343–356, Rome, Italy, January 2013. ACM.

Lecture Notes on Linear Types

15-814: Types and Programming Languages
Frank Pfenning

Lecture 23
Thursday, November 19, 2020

1 Introduction

When we make memory explicit we have to face the problem of garbage collection, that is, freeing memory when it is no longer needed. Ideal would be to deallocate memory at the time we read from it (the last time). One particular interesting class of memory cells are those that have exactly one reader or a *unique reference*. In that case, we can deallocate when it is read. The usual rule (written here without persistent objects)

$$\mathsf{cell}\ c\ V, \mathsf{proc}\ d\ (\mathsf{case}\ c^R\ K) \mapsto \mathsf{cell}\ c\ V, \mathsf{proc}\ d\ (V \triangleright K)$$

would then become

$$\mathsf{cell}\ c\ V, \mathsf{proc}\ d\ (\mathsf{case}\ c^R\ K) \mapsto \mathsf{proc}\ d\ (V \triangleright K)$$

where we model deallocation of the cell c by not repeating it on the right-hand side of the rule.

This is not as infrequent as it might seem at first. For example, in our bit negation pipeline from Lecture 20 all the intermediate cells have a single reader, namely the second process in the pipeline.

Another class of examples comes from temporary cells in the translation of functional expressions.

$$\begin{aligned} [\![e_1\, e_2]\!]\, d = {} & x_1 \leftarrow [\![e_1]\!]\, x_1 \ ; \\ & x_2 \leftarrow [\![e_2]\!]\, x_2 \ ; \\ & x_1^R.\langle x_2, d\rangle \end{aligned}$$

The destination x_1 will be written by the translation $[\![e_1]\!]\,x_1$ and is then read by the last line. But it could not be used beyond that because it can not occur elsewhere in the program since x_1 is fresh and not passed to anywhere.

The situation is different for x_2. Even though it is freshly allocated here it is passed on to the function stored in x_1 so it "escapes its lexical scope" and we cannot deallocate it here.

Methodologically, we might now examine various constructs to see which destinations we may be able to "deallocate" by not copying them from the left-hand sides of transition rule to the right. But this is complicated, so first we examine what would be required so that we would *never* have to copy cells that are being read from (excluding mutable cells from consideration for the moment, for simplicity). Essentially, can we delineate a subsect of the language so that every cell will not only be *written to* once, but also *read from* once. Of course, as you might expect in this course after all we have been through together, this is expressed as a type system! Every memory cell will have not only a unique provider (to write it) but also a unique client (to read from it). We call a type system that enforces this property *linear*, after Girard's *linear logic* [Gir87].

2 Linear Expressions

Even though our ultimate goal is in the runtime system, we start with functional expressions. We say a *function* is linear in one of its arguments if it uses that argument exactly once. The notion of "usage" here is a dynamic one; it doesn't mean that the variable *occurs* exactly once, as we will see.

$$\lambda x.\, x \qquad \text{(linear)}$$

This is linear in x and therefore the whole expression is linear.

$$\lambda x.\, \lambda y.\, x \qquad \text{(not linear)}$$

This expression is linear in x but not linear in y and therefore not linear. It's not linear in y because y is not used, but linearity requires a single use. Related to linearity is the is the notion of *affine*. A function is *affine* in a variable if it is used *at most once*. So the function above is *affine* but not *linear*. The notion of affine has recently received a lot of attention because the Rust programming language treats memory references as affine.

$$\lambda x.\, \langle x, x\rangle \qquad \text{(not linear)}$$

This expression is not linear because x is used twice and hence more than once. Functions that use their argument *at least once* are called *strict*. The notion of strictness is important because it is useful in the optimization of call-by-need languages such as Haskell. If we have a function application $e_1\, e_2$ and we can tell that e_1 denotes a *strict* function we can safely evaluate e_2 rather than waiting until e_1 might need its argument.

$$\lambda x.\, \mathsf{if}\ x\ \mathit{false}\ x \qquad \text{(not linear)}$$

This function is not linear in x. It uses x the first time to decide the condition, and then again when x is false. However, if x is false this returns x which is *false*, so extensionally equal would be

$$\lambda x.\, \mathsf{if}\ x\ \mathit{false}\ \mathit{false} \qquad \text{(linear)}$$

which is linear. These two examples show that linearity is an intensional property of expressions (how do they compute) and not an extensional property (what do they compute).

$$\lambda x.\, \lambda y.\, \mathsf{if}\ x\ y\ (\mathit{not}\ y) \qquad \text{(linear)}$$

This function is linear: x is used once as subject of the conditional. The variable y occurs twice, but whenever this expression is executed it is used exactly once: if y is true then in the first branch, and if y is false then in the second branch.

$$\lambda x.\, (\lambda y. \langle\,\rangle)\, x \qquad \text{(not linear)}$$

It shouldn't be suprising by now that this is not linear, since y is not linear in $\langle\,\rangle$. But, moreover, the whole expression is not linear in x, even though x occurs exactly once. That's because x occurs in a position where it will be dropped. On the other hand:

$$\lambda x.\, (\lambda y.\, \langle y, \langle\,\rangle\rangle)\, x \qquad \text{(linear)}$$

3 Linear Typing of Expressions

With these examples, we now work through the inference rules for expressions and classify those that are linear. We use a different notation for functions, eager pairs, sums, etc. since the connectives are subtly different from the regular ones. Our judgment has the form

$$\Delta \Vdash e : \tau$$

where Δ is a context of variables, each of which must be used once in e. We have seen the $\Vdash$ notation once before, in Lecture 12 where we used it to type patterns in which no variables could be repeated. The use of Δ instead of Γ is just stylistic, to help remind ourselves that all the variables should be linear. We use here the names of the inference rules derived from *linear logic* where introductions rules (for constructors) use I while elimination rules (for destructors) use E.

Linear Functions $\tau \multimap \sigma$. A function is linear just if its parameter is used linearly in its body.

$$\frac{\Delta, x : \tau \Vdash e : \sigma}{\Delta \Vdash \lambda x.\, e : \tau \multimap \sigma} \multimap I$$

When applying a function we have to divide up the variables among those that occur in the function (Δ_1) and those that occur in the argument (Δ_2).

$$\frac{\Delta_1 \Vdash e_1 : \tau_2 \multimap \tau_1 \quad \Delta_2 \Vdash e_2 : \tau_2}{\Delta_1, \Delta_2 \Vdash e_1\, e_2 : \tau_1} \multimap E$$

Our usual presupposition regarding contexts kicks in and we implicit require the $\mathrm{dom}(\Delta_1) \cap \mathrm{dom}(\Delta_2) = \emptyset$. The ordering of the variables in Δ is irrelevant here. If we wanted to maintain them (say, because there are type variables present) then we would use a *merge* operator between the two contexts instead. Nevertheless, in the direction we usually read the rules it would be a *split* operator.

When we look up variables, there cannot be other variables in the context because they would not be used and therefore not be linear.

$$\frac{}{x : \tau \Vdash x : \tau} \mathsf{hyp}$$

Eager Linear Pairs $\tau \otimes \sigma$. Eager linear pairs are written as $\tau \otimes \sigma$. The rules are straightforwardly patterned after previous rules, keeping in mind that for the destructor (case), the variables standing for the components of the pair must be linear.

$$\frac{\Delta_1 \Vdash e_1 : \tau_1 \quad \Delta_2 \Vdash e_2 : \tau_2}{\Delta_1, \Delta_2 \Vdash \langle e_1, e_2 \rangle : \tau_1 \otimes \tau_2} \otimes I \qquad \frac{\Delta \Vdash e : \tau_1 \otimes \tau_2 \quad \Delta', x_1 : \tau_1, x_2 : \tau_2 \Vdash e' : \tau'}{\Delta, \Delta' \Vdash \mathsf{case}\ e\ (\langle x_1, x_2 \rangle \Rightarrow e') : \tau'} \otimes E$$

The nullary version of pairs, the unit is written as 1 and the rules are the nullary version of the binary rules above (see Section 4).

Linear Sums $\tau \oplus \sigma$. Actually, we will show the labeled, variadic version $\oplus_{i\in I}(i : \tau_i)$. In the constructor rule $\oplus I$, there is not much to consider.

$$\frac{\Delta \Vdash e : \tau_j}{\Delta \Vdash j \cdot e : \oplus_{i\in I}(i : \tau_i)}\ \oplus I$$

For the destructor (case) we need to consider the same as for the conditional in the last section: only one branch of the case will be taken, so all branches must be checked with the same linear context.

$$\frac{\Delta \Vdash e : \oplus_{i\in I}(i : \tau_i) \quad \text{(for all } i \in I) \quad \Delta', x_i : \tau_i \Vdash e'_i : \tau'}{\Delta, \Delta' \Vdash \mathsf{case}\ e\ (i \cdot x_i \Rightarrow e'_i)_{i\in I} : \tau'}\ \oplus E$$

Recursion. The remaining type constructors follow similar patterns so we omit the details (see Section 4 for a listing). Recursion, however, is interesting. The computation rule for fixed points is

$$\mathsf{fix}\ f.\, e \mapsto [\mathsf{fix}\ f.\, e/f]e$$

This already departed from the pattern of the other rules. For one, we substitute an expression ($\mathsf{fix}\ f.\, e$) for a variable f in an expression e, while all the other rules just substitute values for variables. For another, it is not attached to a particular type constructor and can always be applied.

There are several sources of operational "nonlinearity" in this rule. First, even if f occurs only once in e, it is replaced by another expression ($\mathsf{fix}\ f.\, e$) containing e, thereby duplicating e. Also, when we define a recursive function we would like to make multiple recursive calls and still consider the function linear.

For example, the function that takes a bit string (usually considered just a binary number) and flips every bit should be linear: each bit of the input string is read and a corresponding bit written to the output.

$$\mathit{bits} = \rho \mathit{bits}.\, (\mathtt{b0} : \mathit{bits}) \oplus (\mathtt{b1} : \mathit{bits}) \oplus (\mathtt{e} : 1)$$

$$\mathit{flip} : \mathit{bits} \multimap \mathit{bit}$$

$$\begin{aligned} \mathit{flip} = \lambda x.\, \mathsf{case}\ (\mathsf{unfold}\ x)\ (\ & \mathtt{b0} \cdot y \Rightarrow \mathsf{fold}\ (\mathtt{b1} \cdot \mathit{flip}\ y) \\ \mid\ & \mathtt{b1} \cdot y \Rightarrow \mathsf{fold}\ (\mathtt{b0} \cdot \mathit{flip}\ y) \\ \mid\ & \mathtt{e} \cdot y \Rightarrow \mathsf{fold}\ (\mathtt{e} \cdot y)\ ) \end{aligned}$$

Note that there is no recursive call to *flip* in the third branch and yet we should consider the function linear. In order to formally represent this,

we have to nonlinear variables to the context, which can be propagated to multiple premises of a rule and may be left over in rules with no premises. Moreover, since the body of the recursively defined expression is duplicated when it is unwound, it may not depend on any linear variables.

$$\frac{\Gamma_{\mathsf{U}}, f_{\mathsf{U}} : \tau \Vdash e : \tau}{\Gamma_{\mathsf{U}} \Vdash \mathsf{fix}\ f.\, e : \tau}\ \mathsf{rec}$$

Here, the subscript U means the variable is *unrestricted* (that is, non necessarily linear), and Γ_{U} stands for a context where all variables are unrestricted. The rules for variables, for example, then would become

$$\frac{}{\Gamma_{\mathsf{U}}, x : \tau \Vdash x : \tau}\ \mathsf{hyp} \qquad \frac{}{\Gamma_{\mathsf{U}}, x_{\mathsf{U}} : \tau \Vdash x : \tau}\ \mathsf{hyp}_{\mathsf{U}}$$

With these rules (and the straightforward ones for *fold* and *unfold*) the *flip* function can indeed be checked as linear.

This example is also remarkable because a tiny change in the last branch of the conditional

$$\begin{array}{ll} \mathit{flip} : \mathit{bits} \multimap \mathit{bit} & \\ \mathit{flip} = \lambda x.\, \mathsf{case}\ (\mathsf{unfold}\ x)\ (\, \mathtt{b0} \cdot y \Rightarrow \mathsf{fold}\ (\mathtt{b1} \cdot \mathit{flip}\ y) & \\ \qquad\qquad\qquad\qquad\quad \mid \mathtt{b1} \cdot y \Rightarrow \mathsf{fold}\ (\mathtt{b0} \cdot \mathit{flip}\ y) & \\ \qquad\qquad\qquad\qquad\quad \mid \mathsf{e} \cdot y \Rightarrow \mathsf{fold}\ (\mathsf{e} \cdot \langle\,\rangle)\,) & \text{\% bug here!} \end{array}$$

makes this function now nonlinear: y is not used. Besides the code shown earlier, we can also fix the problem by *using* $y : 1$.

$$\begin{array}{l} \mathit{flip} : \mathit{bits} \multimap \mathit{bit} \\ \mathit{flip} = \lambda x.\, \mathsf{case}\ (\mathsf{unfold}\ x)\ (\, \mathtt{b0} \cdot y \Rightarrow \mathsf{fold}\ (\mathtt{b1} \cdot \mathit{flip}\ y) \\ \qquad\qquad\qquad\qquad\quad \mid \mathtt{b1} \cdot y \Rightarrow \mathsf{fold}\ (\mathtt{b0} \cdot \mathit{flip}\ y) \\ \qquad\qquad\qquad\qquad\quad \mid \mathsf{e} \cdot y \Rightarrow \mathsf{case}\ y\ (\langle\,\rangle \Rightarrow \mathsf{fold}\ (\mathsf{e} \cdot \langle\,\rangle))\,) \end{array}$$

4 Linear Rule Summary

The syntax for the language of expression does not change, but the language of types is new.

$$\text{Linear types} \quad \tau \quad ::= \quad \tau_1 \multimap \tau_2 \mid \tau_1 \otimes \tau_2 \mid 1 \mid \oplus_{i \in I}(i : \tau_i) \mid \rho\alpha.\, \tau \mid \alpha$$

The definition of values and the rules for evaluation remain the same as for our nonlinear functional language.

We name the propositional rules I (for introduction, representing a constructor for a type) and E (for elimination, representing a destructor for a type). Missing here are the unrestricted variables that would be needed for recursion.

$$\frac{}{x : \tau \Vdash x : \tau}\ \mathsf{hyp}$$

$$\frac{\Delta, x : \tau \Vdash e : \sigma}{\Delta \Vdash \lambda x.\, e : \tau \multimap \sigma}\ {\multimap}I \qquad \frac{\Delta_1 \Vdash e_1 : \tau_2 \multimap \tau_1 \quad \Delta_2 \Vdash e_2 : \tau_2}{\Delta_1, \Delta_2 \Vdash e_1\, e_2 : \tau_1}\ {\multimap}E$$

$$\frac{\Delta_1 \Vdash e_1 : \tau_1 \quad \Delta_2 \Vdash e_2 : \tau_2}{\Delta_1, \Delta_2 \Vdash \langle e_1, e_2 \rangle : \tau_1 \otimes \tau_2}\ {\otimes}I \qquad \frac{\Delta \Vdash e : \tau_1 \otimes \tau_2 \quad \Delta', x_1 : \tau_1, x_2 : \tau_2 \Vdash e' : \tau'}{\Delta, \Delta' \Vdash \mathsf{case}\ e\ (\langle x_1, x_2 \rangle \Rightarrow e') : \tau'}\ {\otimes}E$$

$$\frac{}{\cdot \Vdash \langle\,\rangle : 1}\ 1I \qquad \frac{\Delta \Vdash e : 1 \quad \Delta' \Vdash e' : \tau'}{\Delta, \Delta' \Vdash \mathsf{case}\ e\ (\langle\,\rangle \Rightarrow e') : \tau'}\ 1E$$

$$\frac{(j \in I) \quad \Delta \Vdash e : \tau_j}{\Delta \Vdash j \cdot e : \oplus_{i \in I}(i : \tau_i)}\ {\oplus}I$$

$$\frac{\Delta \Vdash e : \oplus_{i \in I}(i : \tau_i) \quad (\text{for all } i \in I) \quad \Delta', x_i : \tau_i \Vdash e_i' : \tau'}{\Delta, \Delta' \Vdash \mathsf{case}\ e\ (i \cdot x_i \Rightarrow e_i')_{i \in I} : \tau'}\ {\oplus}E$$

$$\frac{\Delta \Vdash e_i : \tau_i \quad (\text{for all } i \in I)}{\Delta \Vdash \langle\!| i \Rightarrow e_i |\!\rangle_{i \in I} : \&_{i \in I}(i : \tau_i)}\ \&I \qquad \frac{\Delta \Vdash e : \&_{i \in I}(i : \tau_i) \quad (j \in I)}{\Delta \Vdash e.j : \tau_j}\ \&E$$

$$\frac{\Delta \Vdash e : [\rho\alpha.\, \tau/\alpha]\tau}{\Delta \Vdash \mathsf{fold}\ e : \rho\alpha.\, \tau}\ \rho I \qquad \frac{\Delta \Vdash e : \rho\alpha.\, \tau}{\Delta \Vdash \mathsf{unfold}\ e : [\rho\alpha.\, \tau/\alpha]\tau}\ \rho E$$

5 Linear Typing of Processes

We didn't prove preservation and progress for linear types. While they are still satisfied, they are not satisfying: we haven't changed any of the dynamics of programs! Linear types, so far, "don't buy us anything".

In this lecture we assign linear types to processes, so that the translation of a linearly typed functional expression becomes a linearly typed process. Then we show that executing a linearly typed process does not require a garbage collector since we can eagerly deallocate cells when they are read. In other words, the right level of abstraction to benefit from linear typing is at a level where memory is made explicit.

Linear typing, though, is too restrictive so what we actually want is a language that combines linear with nonlinear typing. In this combination, linearly typed cells are ephemeral, while other cells remain persistent as in our original semantics for processes. We probably will not have time to cover such a language in this course, but refer you to a recent draft paper [PP20]. Here, we just present purely linear typing.

Our judgment is

$$\Delta \Vdash P :: (z : \sigma)$$

where Δ contains linear variables. The destination z in the succedent is written to exactly once (as before), but it will also be read exactly once. Therefore, the rule for spawn/allocate is

$$\frac{\Delta \Vdash P :: (x : \tau) \quad \Delta, x : \tau \Vdash Q :: (z : \sigma)}{\Delta, \Delta' \Vdash x \leftarrow P \; ; Q :: (z : \sigma)} \; \mathsf{spawn}$$

In the computation rule, we just create a fresh cell as before.

$\mathsf{proc}\; d\; (x \leftarrow P \; ; Q) \mapsto \mathsf{proc}\; c\; ([c/x]P), \mathsf{proc}\; d\; ([c/x]Q) \quad (c \text{ fresh})$

The rule for variables: one that reads from an ephemeral (linear) cell and deallocates it.

$$\frac{}{y : \tau \Vdash x^W \leftarrow y^R :: (x : \tau)} \; \mathsf{move}$$

Computationally, this first rule *moves* while the second one *copies*.

$\mathsf{cell}\; c\; W, \mathsf{proc}\; d\; (d \leftarrow c) \mapsto \mathsf{cell}\; d\; W$ (move)

Eager Linear Pairs. As an example for linear typing, we use pairs. In general, we write linear typing rules as *left rules* (if the type constructor appears in the antecedent) and *right rules* (if the type constructor appears in the succedent). Note that left rules always read from memory, while right rules always write to memory.

$$\frac{}{x_1 : \tau_1, x_2 : \tau_2 \Vdash z^W.\langle x_1, x_2 \rangle :: (z : \tau_1 \otimes \tau_2)} \; {\otimes}R^0$$

$$\frac{\Delta, x_1 : \tau_1, x_2 : \tau_2 \Vdash P :: (z : \sigma)}{\Delta, x : \tau_1 \otimes \tau_2 \Vdash \mathsf{case}\; x^R\; (\langle x_1, x_2 \rangle \Rightarrow P) :: (z : \sigma)} \; {\otimes}L$$

Operationally, the case rule reads from memory and passes it to the continuation. These rules are general for all positive types. The only difference from before is that the cell that is read is ephemeral and therefore “deallocated”.

$$\textsf{proc}\ d\ (d^W.V) \mapsto \textsf{cell}\ d\ V \qquad \text{(write/pos)}$$
$$\textsf{cell}\ c\ V, \textsf{proc}\ d\ (\textsf{case}\ c^R\ K) \mapsto \textsf{proc}\ d\ (V \triangleright K) \qquad \text{(read/pos)}$$

where

$$\begin{array}{llcl} \text{Values} & V & ::= & \langle d_1, d_2 \rangle \mid \ldots \\ \text{Conts} & K & ::= & (\langle x_1, x_2 \rangle \Rightarrow P) \mid \ldots \end{array}$$

with

$$\langle d_1, d_2 \rangle \triangleright (\langle x_1, x_2 \rangle \Rightarrow P) = [d_1/x_1, d_2/x_2]P$$

Linear Sums. They follow the pattern of the eager pairs, since they are a positive type.

$$\frac{j \in I}{y : \tau_j \Vdash x^W.(j \cdot y) :: (x : \oplus_{i \in I}(i : \tau_i))}\ \oplus R^0$$

$$\frac{(\text{for all } i \in I) \quad \Delta, y_i : \tau_i \Vdash P_i :: (z : \sigma)}{\Delta, x : \oplus_{i \in I}(i : \tau_i) \Vdash \textsf{case}\ x^R\ (i \cdot y_i \Rightarrow P_i)_{i \in I} :: (z : \sigma)}\ \oplus L$$

where

$$j \cdot d \triangleright (i \cdot y_i \Rightarrow P_i)_{i \in I} = [d/y_j]P_j$$

Linear functions. Since functions are a negative type, the case constructs writes a continuation to memory.

$$\frac{\Delta, y : \tau \Vdash P :: (z : \sigma)}{\Delta \Vdash \textsf{case}\ x^W\ (\langle y, z \rangle \Rightarrow P) :: (x : \tau \multimap \sigma)}\ \multimap R$$

$$\frac{}{x : \tau \multimap \sigma, y : \tau \Vdash x^R.\langle y, z \rangle :: (z : \sigma)}\ \multimap L^0$$

This time, we have to provide a second set of rules since the roles of values and continuations are flipped.

$$\textsf{proc}\ d\ (\textsf{case}\ d^W\ K) \mapsto \textsf{cell}\ d\ K \qquad \text{(write/neg)}$$
$$\textsf{cell}\ c\ K, \textsf{proc}\ d\ (d^R.V) \mapsto \textsf{proc}\ d\ (V \triangleright K) \qquad \text{(read/neg)}$$

where the reduction $\langle d_1, d_2 \rangle \triangleright (\langle y, z \rangle \Rightarrow P)$ has already been defined.

The summary of all the rules for linear processes can be found in the linear rule sheet.

Recursion. We assume all functions can be mutually recursive and are defined at the top level and have no other free variables. Then we translate each definition

$$\mathit{func} = \lambda x.\, e$$

as

$$!\mathsf{cell}\ \mathit{func}\ (\langle x, z\rangle \Rightarrow [\![e]\!]\, z)$$

where

$$[\![\mathit{func}]\!]\, d = (d^W \leftarrow \mathit{func}^R)$$

Slightly more generally, if we want to allow mutually recursive definitions for arbitrary negative types constructed at the top level, we would translate each definition

$$f = e$$

to

$$!\mathsf{cell}\ f\ K \qquad \text{for } [\![e]\!]\, d_0 = \mathsf{case}\ d_0\ K$$

Under this view, functions become like *constants* that are visible throughout the program, similarly to the specific treatment we have given fixed points.

6 Example: Bit Flipping Revisited

With the treatment of recursion from the end of the previous section, the (linear) bit flipping program becomes (eliding uses of fold):

$$\begin{aligned}
\mathit{flip}_K = (\langle x, y\rangle \Rightarrow \mathsf{case}\ x^R\ (\ & \mathbf{b0} \cdot x' \Rightarrow y' \leftarrow \mathit{flip}^R.\langle x', y'\rangle \\
& \qquad\qquad\ \ y^W.(\mathbf{b1} \cdot y') \\
|\ & \mathbf{b1} \cdot x' \Rightarrow y' \leftarrow \mathit{flip}^R.\langle x', y'\rangle \\
& \qquad\qquad\ \ y^W.(\mathbf{b0} \cdot y') \\
|\ & \mathbf{e} \cdot u \Rightarrow z^W.(\mathbf{e} \cdot u)\ )\ )
\end{aligned}$$

where the initial state of running the program contains

$!\mathsf{cell}\ \mathit{flip}\ \mathit{flip}_K$

This is now entirely linearly typed, except for the references to *flip*.

7 Look Ma, No Garbage!

With linear typing, cells are deallocated as they are used. For example, the flip program started with a state such as

$!\mathsf{cell}\ \mathit{flip}\ \mathit{flip}_K,$
$\mathsf{cell}\ c_4\ \langle\,\rangle, \mathsf{cell}\ c_3\ (\mathbf{e}\cdot c_4), \mathsf{cell}\ c_2\ (\mathsf{fold}\ c_3),$
$\mathsf{cell}\ c_1\ (\mathbf{b0}\cdot c_2), \mathsf{cell}\ c_0\ (\mathsf{fold}\ c_1),$
$\mathsf{proc}\ d_0\ (\mathit{flip}^R.\langle c_0, d_0\rangle)$

will end with a state

$!\mathsf{cell}\ \mathit{flip}\ \mathit{flip}_K,$
$\mathsf{cell}\ c_4\ \langle\,\rangle, \mathsf{cell}\ d_3\ (\mathbf{e}\cdot c_4), \mathsf{cell}\ d_2\ (\mathsf{fold}\ d_3),$
$\mathsf{cell}\ d_1\ (\mathbf{b1}\cdot d_2), \mathsf{cell}\ d_0\ (\mathsf{fold}\ d_1)$

We have executed here the version that does not explicitly copy the unit element to a new cell. Note that all cells, except for *flip*, are reachable from d_0, the initial destination of the call.

In general, if we started with an empty configuration (again, excepting only the recursive functions), as would be the case for the translation of

$$[\![\mathit{flip}\ (\mathsf{fold}\ (\mathbf{b0}\cdot(\mathsf{fold}\ (\mathbf{e}\cdot\langle\,\rangle))))]\!]\ d_0$$

all cells in the resulting state would be reachable from d_0 as shown in this example.

In order to prove such a result we need to make the typing of configurations explicit and then examine the change in configurations during computation. We have:

$$\text{Configurations}\quad \mathcal{C} \quad ::= \quad \cdot \mid \mathcal{C}_1, \mathcal{C}_2 \mid \mathsf{proc}\ d\ P \mid \mathsf{cell}\ c\ W$$

where we omit the persistent cells for closed, top-level functions. We imagine they are defined in a global context. The linear typing judgment then has the form

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

for a configuration that writes to Δ' and reads from Δ. As before, any addresses in Δ not read by a process in $\mathcal{C}$ are passed on to Δ' to be read by a

process further on the right.

$$\frac{}{\Delta \Vdash (\cdot) :: \Delta}\ \mathsf{tp/empty} \qquad \frac{\Delta \Vdash P :: (d : \tau)}{\Delta', \Delta \Vdash \mathsf{proc}\ d\ P :: (\Delta', d : \tau)}\ \mathsf{tp/proc}$$

$$\frac{\Delta \Vdash c^W.V :: (c : \tau)}{\Delta', \Delta \Vdash \mathsf{cell}\ c\ V :: (\Delta', c : \tau)}\ \mathsf{tp/cell/val} \qquad \frac{\Delta \Vdash \mathsf{case}\ c^W\ K :: (c : \tau)}{\Delta', \Delta \Vdash \mathsf{cell}\ c\ K :: (\Delta', c : \tau)}\ \mathsf{tp/cell/cont}$$

$$\frac{\Delta \Vdash \mathcal{C}_1 :: \Delta_1 \quad \Delta_1 \Vdash \mathcal{C}_2 :: \Delta_2}{\Delta \Vdash (\mathcal{C}_1, \mathcal{C}_2) :: \Delta_2}\ \mathsf{tp/join}$$

8 Progress and Preservation

Progress is essentially unchanged from before.

Theorem 1 *If $\cdot \Vdash \mathcal{C} :: \Delta$ then either $\mathcal{C}$ is final (consists only of cells) or $\mathcal{C} \mapsto \mathcal{C}'$ for some $\mathcal{C}'$.*

The preservation theorem is the interesting one. In case of linearly typed processes, the cells defined (or promised to be defined by a process) does not change throughout the computation!

Theorem 2 *If $\Delta \Vdash \mathcal{C} :: \Delta'$ and $\mathcal{C} \mapsto \mathcal{C}'$ then $\Delta \Vdash \mathcal{C} :: \Delta'$.*

Contrast this with the previous statement of preservation where the output context may grow when a new cell is allocated.

The form of the preservation theorem now means that if we start, for example, with $\cdot \Vdash \mathcal{C} :: (d_0 : 1)$ then any resulting final configuration $\mathcal{C} \mapsto^* \mathcal{F}$ still has the same type. Since there are only ephemeral cells in $\mathcal{F}$, it must be of the form $\mathcal{F}', \mathsf{cell}\ d_0\ W$ for some $\mathcal{F}'$ and W. Since $d_0 : 1$, it follows by inversion that $W = \langle\,\rangle$. Moreover, $\cdot \Vdash \mathcal{F}' :: (\cdot)$. Again by inversion we find $\mathcal{F}' = (\cdot)$, so the whole configuration consists of just $\mathsf{cell}\ d_0\ \langle\,\rangle$.

Looking at the typing rules we can see that in general the context Δ acts like a frontier for an algorithm to traverse a tree with root d_0, the initial destination. It must eventually be empty which shows that every ephemeral cell is reachable and no garbage is created.

Exercises

Exercise 1 Write a linear increment function on natural numbers in binary representation.

Exercise 2 Recall the definition of a purely positive type, updated to reflect the notation for linear types.

$$\tau^+ ::= 1 \mid \tau_1^+ \otimes \tau_2^+ \mid \oplus_{i \in I}(i : \tau_i^+) \mid \rho\alpha^+ .\, \tau^+ \mid \alpha^+$$

Even in the purely linear language, it is possible to *copy* a value of purely linear type. Define a family of functions

$$\mathit{copy}_{\tau^+} : \tau^+ \multimap (\tau^+ \otimes \tau^+)$$

such that $\mathit{copy}_{\tau^+}\ v \mapsto^* \langle v, v \rangle$ for every $v : \tau^+$. You do not need to prove this property, just give the definitions of the *copy* functions. Your definitions may be mutually recursive.

Exercise 3 A type isomorphism is *linear* if the functions *Forth* and *Back* are both linear. For each of the following pairs of types provide linear functions witnessing an isomorphism if they exist, or indicate no linear isomorphism exists. You may assume all functions terminate and use either extensional or logical equality as the basis for your judgment.

1. $\tau \multimap (\sigma \multimap \rho)$ and $\sigma \multimap (\tau \multimap \rho)$
2. $\tau \multimap (\sigma \multimap \rho)$ and $(\tau \otimes \sigma) \multimap \rho$
3. $\tau \multimap (\sigma \otimes \rho)$ and $(\tau \multimap \sigma) \otimes (\tau \multimap \rho)$
4. $(\tau \oplus \sigma) \multimap \rho$ and $(\tau \multimap \rho) \otimes (\sigma \multimap \rho)$
5. $(1 \oplus 1) \multimap \tau$ and $\tau \otimes \tau$

Exercise 4 Write out the following theorems, updated to the purely linear language (where only recursively defined variables are nonlinear). We change neither the definition of value nor the rules for stepping from our previous language that does not employ linearity.

1. Canonical forms for types $\multimap$, $\otimes$, 1, $\oplus$, and ρ. No proofs are needed.
2. The substitution properties, in a form sufficient needed for preservation. No proofs are needed.
3. The preservation property for evaluation of closed linear expressions. Show the proof cases for linear functions.
4. The progress property for closed linear expressions. Show the proof cases for linear functions.

5. Where do these properties and their proofs differ when compared to our language that does not enforce linearity?

Exercise 5 Prove that $\Delta \Vdash e : \tau$ implies $\Delta \Vdash [\![e]\!]\, d :: (d : \tau)$. You only need to show the cases relevant for functions ($\lambda x.\, e$, $e_1\, e_2$ and variables x).

Exercise 6 Write a linear function *inc* on the binary representation of natural numbers.

1. Provide the code as a functional expression.

2. Following the conventions of this lecture, show the result of the translation into a process expression. You may use the optimization we presented here. Concretely, define inc_K and *inc* so that the program representation as a configuration would be $!\mathsf{cell}\ inc\ inc_K$.

3. Show the initial and final configuration of computation for incrementing the number 1 represented as $\mathsf{fold}\ (\mathbf{b1} \cdot (\mathsf{fold}\ (\mathbf{e} \cdot \langle\,\rangle)))$.

References

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

[PP20] Klaas Pruiksma and Frank Pfenning. Back to futures. *CoRR*, abs/2002.04607, February 2020.

Abstract Syntax

Types	τ	$::=$	$\tau_1 \multimap \tau_2 \mid \&_{i\in I}(i : \tau_i) \mid \tau_1 \otimes \tau_2 \mid 1 \mid \oplus_{i\in I}(i : \tau_i) \mid \rho\alpha.\,\tau$	
Contexts	Δ	$::=$	$\cdot \mid \Delta, x : \tau$	(all variables distinct)
Processes	P	$::=$	$x \leftarrow P \; ; \; Q$	allocate/spawn
		$\mid$	$x^w \leftarrow y^R$	move
		$\mid$	$x^W.V \mid \mathsf{case}\ x^R\ K$	$(1, \otimes, \oplus, \rho)$
		$\mid$	$x^R.V \mid \mathsf{case}\ x^W\ K$	$(\multimap, \&)$

Small values	V	$::=$	$\langle\,\rangle \mid \langle a_1, a_2\rangle \mid i \cdot a \mid \mathsf{fold}\ a$
Continuations	K	$::=$	$(\langle\,\rangle \Rightarrow P) \mid (\langle x_1, x_2\rangle \Rightarrow P) \mid (i \cdot x_i \Rightarrow P_i)_{i\in I} \mid (\mathsf{fold}\ x \Rightarrow P)$
Cell contents	W	$::=$	$V \mid K$
Configurations	$\mathcal{C}$	$::=$	$\cdot \mid \mathcal{C}_1, \mathcal{C}_2 \mid \mathsf{proc}\ d\ P \mid \mathsf{cell}\ c\ W$

Judgments

$\Delta \Vdash P :: (z : \sigma)$	process P reads from Δ and writes to $z : \sigma$
$\Delta \Vdash \mathcal{C} :: \Delta'$	configuration $\mathcal{C}$ reads from Δ and writes to Δ'
$\mathcal{C}$ *final*	configuration $\mathcal{C}$ is final (consists only of cells)

Theorems

Preservation. If $\Delta \Vdash \mathcal{C} :: \Delta'$ and $\mathcal{C} \mapsto \mathcal{C}'$ then $\Delta \Vdash \mathcal{C}' :: \Delta'$.

Progress. If $\cdot \Vdash \mathcal{C} :: \Delta$ then either $\mathcal{C} \mapsto \mathcal{C}'$ for some $\mathcal{C}'$ or $\mathcal{C}$ *final*.

Statics, Allocate and Move

$$\frac{\Delta \Vdash P :: (x : \tau) \quad \Delta', x : \tau \Vdash Q :: (z : \sigma)}{\Delta, \Delta' \Vdash (x \leftarrow P \; ; \; Q) :: (z : \sigma)}\ \mathsf{tp/alloc} \qquad \frac{}{y : \tau \Vdash x^W \leftarrow y^R :: (x : \tau)}\ \mathsf{tp/move}$$

Statics, Positive Types

$$\frac{}{\cdot \Vdash x^W.\langle\,\rangle :: (x : 1)}\ \textsf{w/unit} \qquad \frac{\Delta \Vdash P :: (z : \sigma)}{\Delta, x : 1 \Vdash \textsf{case}\ x^R\ (\langle\,\rangle \Rightarrow P) :: (z : \sigma)}\ \textsf{r/unit}$$

$$\frac{}{y : \tau, z : \sigma \Vdash x^W.\langle y, z\rangle :: (x : \tau \otimes \sigma)}\ \textsf{w/pair} \qquad \frac{\Delta, x_1 : \tau_1, x_2 : \tau_2 \Vdash P :: (z : \sigma)}{\Delta, x : \tau_1 \otimes \tau_2 \Vdash \textsf{case}\ x^R\ (\langle x_1, x_2\rangle \Rightarrow P) : (z : \sigma)}\ \textsf{r/pair}$$

$$\frac{(j \in I)}{y : \tau_j \Vdash x^W.(j \cdot y) :: (x : \oplus_{i \in I}(i : \tau_i))}\ \textsf{w/tag} \qquad \frac{\Delta, y_i : \tau_i \Vdash P_i :: (z : \sigma) \quad (\text{for all } i \in I)}{\Delta, x : \oplus_{i \in I}(i : \tau_i) \Vdash \textsf{case}\ x^R\ (i \cdot y_i \Rightarrow P_i)_{i \in I} :: (z : \sigma)}\ \textsf{r/tag}$$

$$\frac{}{y : [\rho\alpha.\,\tau/\alpha]\tau \Vdash x^W.(\textsf{fold}\ y) :: (x : \rho\alpha.\,\tau)}\ \textsf{w/fold} \qquad \frac{\Delta, y : [\rho\alpha.\,\tau/\alpha]\tau \Vdash P :: (z : \sigma)}{\Delta, x : \rho\alpha.\,\tau \in \Delta \Vdash \textsf{case}\ x^R\ (\textsf{fold}\ y \Rightarrow P) :: (z : \sigma)}\ \textsf{r/fold}$$

Statics, Negative Types

$$\frac{\Delta, y : \tau \Vdash P :: (z : \sigma)}{\Delta \Vdash \textsf{case}\ x^W\ (\langle y, z\rangle \Rightarrow P) :: (x : \tau \multimap \sigma)}\ \textsf{w/fun} \qquad \frac{}{x : \tau \multimap \sigma, y : \tau \Vdash x^R.\langle y, z\rangle :: (z : \sigma)}\ \textsf{r/fun}$$

$$\frac{\Delta \Vdash P_i :: (z_i : \tau_i) \quad (\text{for all } i \in I)}{\Delta \Vdash \textsf{case}\ x^W\ (i \cdot z_i \Rightarrow P_i)_{i \in I} :: (x : \&_{i \in I}(i : \tau_i))}\ \textsf{w/record} \qquad \frac{(j \in I)}{x : \&_{i \in I}(i : \tau_i) \Vdash x^R.(j \cdot z) :: (z : \tau_j)}\ \textsf{r/record}$$

Statics, Configurations

$$\frac{\Delta \Vdash P :: (c : \tau)}{\Delta', \Delta \Vdash \textsf{proc}\ c\ P :: (\Delta', c : \tau)}\ \textsf{tp/proc}$$

$$\frac{\Delta \Vdash c^W.V :: (c : \tau)}{\Delta', \Delta \Vdash \textsf{cell}\ c\ V :: (\Delta', c : \tau)}\ \textsf{tp/cell/val} \qquad \frac{\Delta \Vdash \textsf{case}\ c^W\ K :: (c : \tau)}{\Delta', \Delta \Vdash \textsf{cell}\ c\ K :: (\Delta', c : \tau)}\ \textsf{tp/cell/cont}$$

$$\frac{}{\Delta \Vdash (\cdot) :: \Delta}\ \textsf{tp/empty} \qquad \frac{\Delta \Vdash \mathcal{C}_1 :: \Delta_1 \quad \Delta_1 \Vdash \mathcal{C}_2 :: \Delta_2}{\Delta \Vdash (\mathcal{C}_1, \mathcal{C}_2) :: \Delta_2}\ \textsf{tp/join}$$

$$\frac{}{(\cdot)\ \mathit{final}}\ \textsf{fin/empty} \qquad \frac{\mathcal{C}\ \mathit{final}}{(\mathcal{C}, \textsf{cell}\ c\ W)\ \mathit{final}}\ \textsf{fin/cell}$$

Dynamics

$$
\begin{array}{lllll}
 & \mathsf{proc}\ d\ (x \leftarrow P\ ;\ Q) & \mapsto & \mathsf{proc}\ c\ ([c/x]P), \mathsf{proc}\ d\ ([c/x]Q) & \text{(alloc/spawn, } c \text{ fresh)} \\
\mathsf{cell}\ c\ W, & \mathsf{proc}\ d\ (d^W \leftarrow c^R) & \mapsto & \mathsf{cell}\ d\ W & \text{(move)} \\
 & & & & \\
 & \mathsf{proc}\ d\ (d^W.V) & \mapsto & \mathsf{cell}\ d\ V & \text{(write: } \otimes, 1, \oplus, \rho) \\
\mathsf{cell}\ c\ V, & \mathsf{proc}\ d\ (\mathsf{case}\ c^R\ K) & \mapsto & \mathsf{proc}\ d\ (V \triangleright K) & \text{(read: } \otimes, 1, \oplus, \rho) \\
 & & & & \\
 & \mathsf{proc}\ d\ (\mathsf{case}\ d^W\ K) & \mapsto & \mathsf{cell}\ d\ K & \text{(write: } \multimap, \&) \\
\mathsf{cell}\ c\ K, & \mathsf{proc}\ d\ (c^R.V) & \mapsto & \mathsf{proc}\ d\ (V \triangleright K) & \text{(read: } \multimap, \&)
\end{array}
$$

$$
\begin{array}{rcll}
\langle\,\rangle & \triangleright & (\langle\,\rangle \Rightarrow P) & = & P \\
\langle c_1, c_2 \rangle & \triangleright & (\langle x_1, x_2 \rangle \Rightarrow P) & = & [c_1/x_1, c_2/x_2]P \\
j \cdot c & \triangleright & (i \cdot x_i \Rightarrow P_i)_{i \in I} & = & [c/x_j]P_j \\
\mathsf{fold}\ c & \triangleright & (\mathsf{fold}\ x \Rightarrow P) & = & [c/x]P
\end{array}
$$

Lecture Notes on Message-Passing Concurrency

15-814: Types and Programming Languages
Frank Pfenning

Lecture 24
Tuesday, December 1, 2020

1 Introduction

So far, we have viewed concurrency through the lens of shared memory. That's because there is a direct way of translating expressions into processes that make memory allocation as well as reading and writing explicit. The usual sequential dynamics can be recovered easily, as pointed out in Section L21.5 and Exercise L21.4, but concurrency is in fact most natural. Nevertheless computation is quite *pure*, not covering mutable references, but see Lecture 22 for an approach to adding this to a call-by-value language.

Communication in shared memory takes the form of writing to and reading from shared cells. However, there are many situations where processes that execute concurrently do not have a shared address space but need to communicate with each other via messages. And even if shared memory is available, message-passing is a useful, perhaps higher-level abstraction that may prevent certain kinds of errors. For example, the Go concurrency slogan[1] exhorts:

> *Do not communicate by sharing memory; instead, share memory by communicating.*

So, given our type-based approach, can we model message-passing concurrency? The answer to this rhetorical question is of course "Yes!" In fact, the whole approach to concurrency we have taken in this course originated by examining message-passing concurrency through the lens of *linear*

[1] https://golang.org/doc/effective_go.html

logic [CP10, CPT16]. The adaption to shared memory did not come until later [PP20].

There are different computational models for message-passing concurrency, for example, *actors* [Agh85] or the π*-calculus* [MPW92]. Our model resembles the *asynchronous* π*-calculus* [Bou92] in that a sender can proceed immediately, while a recipient blocks until a message is received. Unlike the asynchronous π-calculus our calculus enforces that message are received in the order they are sent, which is essential for type soundness (that is, preservation and progress).

2 Reinterpreting Process Typing

The key step towards message-passing concurrency from where we are is to reinterpret the typing judgment for processes. With shared-memory concurrency we have

$$\underbrace{x_1 : \tau_1, \ldots, x_n : \tau_n}_{\text{read from}} \Vdash P :: \underbrace{(y : \sigma)}_{\text{write to}}$$

where all variables x_i and y stand for *addresses* in shared memory. The process P reads from the x_i and writes to y. Linearity ensured that a terminating process is guaranteed to read all the x_i and write to y.

With message-passing concurrency, each variable stands for a *channel for bidirectional communication*,

$$\underbrace{x_1 : \tau_1, \ldots, x_n : \tau_n}_{\substack{\text{use} \\ \text{may send or recv}}} \Vdash P :: \underbrace{(y : \sigma)}_{\substack{\text{provide} \\ \text{may send or recv}}}$$

where the channel y on the right represents a service provided and the channels x_i on the left a service used. We say P is a *provider* for y and a *client* to all x_i. It is now the types σ and τ_i that determine whether messages are sent or received.

The rule to allocate or spawn (called *cut* under its logical interpretation) still has the same effect, except that it allocates a *fresh private channel* connecting a provider P to its client Q instead of a memory cell.

$$\frac{\Delta \Vdash P :: (x : \tau) \quad \Delta', x : \tau \Vdash Q :: (y : \sigma)}{\Delta, \Delta' \Vdash (x \leftarrow P \,;\, Q) :: (y : \sigma)}\ \mathsf{cut}$$

Because the channel x here is shared between P and Q, any type τ prescribes two complementary actions: one by the provider (say, a send) another one by the client (say, a corresponding receive).

The dynamics is as before, except we no longer record a distinguished destination since communication is bidirectional.

$$\mathsf{proc}\ (x \leftarrow P\ ;\ Q) \mapsto \mathsf{proc}\ ([c/x]P), \mathsf{proc}\ ([c/x]Q) \qquad (c \text{ fresh})$$

In Section 8 we return to the identity rule, which corresponded to moving a value from one cell to another but now *forwards* messages from one channel to another.

3 Tagged Sums Become Internal Choice

Taking the provider's perspective, constructs associated with positive types will *send* a message. Therefore, from the client's perspective, they will *receive*.

Intuitively, we think of a sequence of messages on a channel as beads on a string. While this image is correct, we have to be careful that multiple messages on a channel arrive in the order they were sent—otherwise, type safety might fail when consecutive messages have different type. The way we accomplish this is that every message (except one that closes a channel) carries a *continuation channel* for further communication. In an implementation, this could be achieved with an explicit message queue, which in our formulation is an emergent structure rather than a primitive.

The *tagged sum* $\sum_{i\in I}(i : \tau_i)$ corresponds to *internal choice* $\oplus_{i\in I}(i : \tau_i)$. A provider of a channel $x : \oplus_{i\in I}(i : \tau_i)$ will *send* a label $j \in I$ along x and a continuation channel of type τ_j. This is called *internal choice* because the *provider can choose* which label to send.

$$\frac{(j \in I)}{y : \tau_j \Vdash x.(j \cdot y) :: (x : \oplus_{i\in I}(i : \tau_i))}\ \oplus R^0$$

Correspondingly, the recipient will branch based on the label received.

$$\frac{\Delta, y_i : \tau_i \Vdash P_i :: (z : \sigma) \quad (\text{for all } i \in I)}{\Delta, x : \oplus_{i\in I}(i : \tau_i) \Vdash \mathsf{case}\ x\ (i \cdot y_i \Rightarrow P_i)_{i\in I} :: (z : \sigma)}\ \oplus L$$

These linear rules are exactly as before. In order to describe the changed dynamics, we need a second form of semantic object $\mathsf{msg}\ c\ V$ which means

that the small value V is a message on channel c. Then we have

$$\begin{array}{rcl} & \mathsf{proc}\ (c.(j \cdot c')) & \mapsto & \mathsf{msg}\ c\ (j \cdot c') \\ \mathsf{msg}\ c\ (j \cdot c'), & \mathsf{proc}\ (\mathsf{case}\ c\ (i \cdot y_i \Rightarrow P_i)_{i \in I}) & \mapsto & \mathsf{proc}\ ([c'/y_j]P_j) \end{array}$$

Observe the significance of linearity here: in the second rule, the message is ephemeral so it is removed from the configuration so that now the next message can be received. Meanwhile, the continuation channel c' is substituted for y_j, which is the placeholder for the continuation channel.

4 Generalizing to Other Types

Before we write out our bit flipping pipeline once again (now in a message-passing interpretation), we can summarize the possible configurations and even the transition rules. Small values V are as before, except they are comprised of channels, not addresses. Continuation processes K are also as before (see the Lecture 23 Rule Sheet for reference).

$$\begin{array}{llcll} \text{Processes} & P & ::= & x \leftarrow P\ ;\ Q & \text{spawn} \\ & & | & x.V & \text{send} \\ & & | & \mathsf{case}\ x\ K & \text{receive} \\ \\ \text{Configurations} & \mathcal{C} & ::= & \cdot \mid \mathcal{C}_1, \mathcal{C}_2 \mid \mathsf{proc}\ P \mid \mathsf{msg}\ c\ V & \end{array}$$

The dynamics then, so far, consists of only three rules, where we have speculatively generalized, sending arbitrary V and passing them to arbitrary K when received.

$$\begin{array}{lcll} \mathsf{proc}\ (x \leftarrow P\ ;\ Q) & \mapsto & \mathsf{proc}\ ([c/x]P), \mathsf{proc}\ ([c/x]Q) & (c\ \text{fresh}) \\ \mathsf{proc}\ (c.V) & \mapsto & \mathsf{msg}\ c\ V & (\text{send}) \\ \mathsf{msg}\ c\ V, \mathsf{proc}\ (\mathsf{case}\ c\ K) & \mapsto & \mathsf{proc}\ (V \triangleright K) & (\text{receive}) \end{array}$$

In fact, the last two rules will be sufficient for all positive and negative types!

A perhaps unexpected aspect of these rules is that a process that sends a message terminates. That works out because in programs we create a continuation channel c' and a very short-lived process that corresponds to just a message. This is actually quite similar to the asynchronous π-calculus where there is no explicit construct for sending a message, just parallel composition with a process that (intuitively) represents a message.

5 The Bit-Flipping Pipeline Revisited

In order to consider recursively defined processes, we allow global declarations of f in a fixed signature Σ in the form

$x_1 : \tau_1, \ldots, x_n : \tau_n \Vdash f :: (y : \sigma)$
$y \leftarrow f\, x_1\, \ldots\, x_n = P$

where the process expression P is typed with

$$x_1 : \tau_1, \ldots, x_n : \tau_n \Vdash P :: (y : \sigma)$$

We then add to the language of processes a *call*

$$d \leftarrow f\, c_1\, \ldots\, c_n$$

which is typed as follows (taking care to allow the concrete arguments to be different from the names of the parameters):

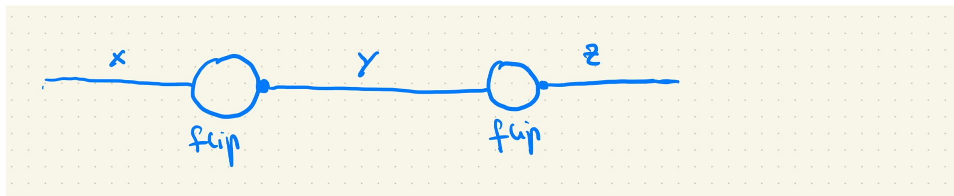
$$\frac{(x_1 : \tau_1, \ldots, x_n : \tau_n \Vdash f :: (y : \sigma)) \in \Sigma}{c_1 : \tau_1, \ldots, c_n : \tau_n \Vdash f :: (d : \sigma)}\ \mathsf{call}$$

In the concrete example of the bit-flipping pipeline we have

$$\mathit{bits} \cong (\mathbf{b0} : \mathit{bits}) + (\mathbf{b1} : \mathit{bits})$$

where we elide the fold constructor in the examples to simplify the code, as we have done in recent lectures.

Then the pipeline below



consists of two running processes, both executing *flip*. The channel y is a private channel connecting these two processes. We use a small dot to indicate the channel provided by a process (y for the process on the left and z for the process on the right).

$x : \textit{bits} \Vdash \textit{flip} :: (y : \textit{bits})$
$y \leftarrow \textit{flip}\ x = \ldots \qquad$ *% to be written later*

$x : \textit{bits} \Vdash \textit{flip2} :: (z : \textit{bits})$
$z \leftarrow \textit{flip2}\ x =$
$\quad y \leftarrow (y \leftarrow \textit{flip}\ x)$
$\quad z \leftarrow \textit{flip}\ y$

The first line in the definition of *flip2* creates a new channel y, which is provided by the first *flip* process and used by the second. Because this pattern is pervasive, we use a derived notation that combines a cut and abbreviate $x \leftarrow (x \leftarrow f\, y_1\ \ldots\ y_n)\ ;\ Q$ by $x \leftarrow f\, y_1\ \ldots\ y_n\ ;\ Q$.

$x : \textit{bits} \Vdash \textit{flip2} :: (z : \textit{bits})$
$z \leftarrow \textit{flip2}\ x =$
$\quad y \leftarrow \textit{flip}\ x$
$\quad z \leftarrow \textit{flip}\ y$

We can quickly verify that this definition is *linear*: x is used in the first call to *flip* which provides y, used in the second call to *flip*.

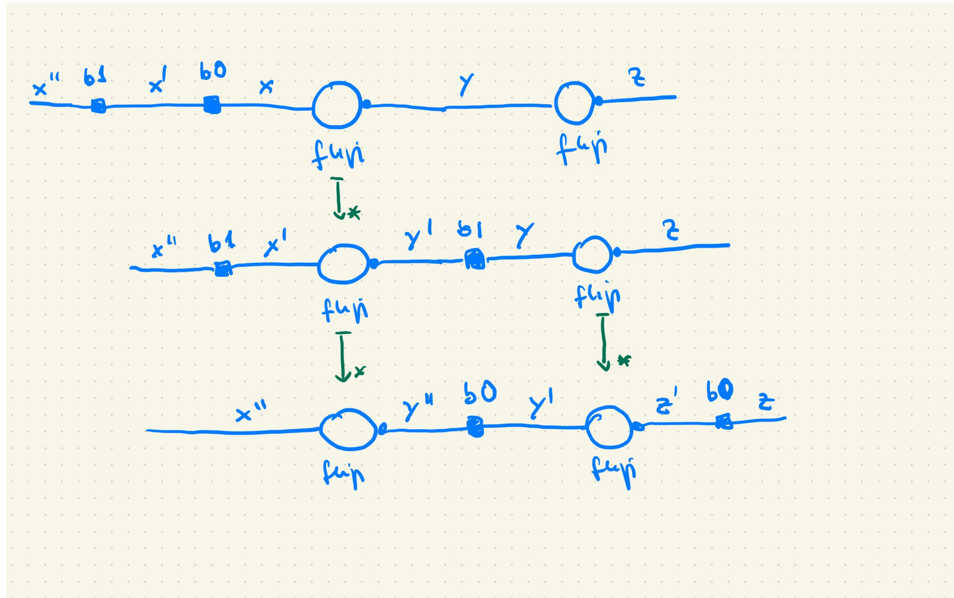
Now the code for *flip* itself receives a bit along channel x, together with a continuation channel x'.

$$
\begin{array}{l}
y \leftarrow \textit{flip}\ x = \\
\mathsf{case}\ x\ \ (\, \mathbf{b0} \cdot x' \Rightarrow \ldots \\
\qquad\quad\ \mid \mathbf{b1} \cdot x' \Rightarrow \ldots \\
\qquad\quad\ )
\end{array}
$$

Before we can send the negated bit $\mathbf{b1}$ along y, we have to create the continuation channel for it. We obtain this from the recursive call, because after the interaction the *flip* process has continuation channels x' and y' on the two sides. The handling of $\mathbf{b1}$ is symmetric.

$$
\begin{array}{l}
y \leftarrow \textit{flip}\ x = \\
\mathsf{case}\ x\ \ (\, \mathbf{b0} \cdot x' \Rightarrow y' \leftarrow \textit{flip}\ x'\ ; \\
\qquad\qquad\qquad\qquad\ y.(\mathbf{b1} \cdot y') \\
\qquad\quad\ \mid \mathbf{b1} \cdot x' \Rightarrow y' \leftarrow \textit{flip}\ x'\ ; \\
\qquad\qquad\qquad\qquad\ y.(\mathbf{b0} \cdot y') \\
\qquad\quad\ )
\end{array}
$$

In pictures:



The configuration shown can make the following transition:

$$\begin{array}{ll} & \mathsf{msg}\ x'\ (\mathsf{b1} \cdot x''), \mathsf{msg}\ x\ (\mathsf{b0} \cdot x'), \mathsf{proc}\ (y \leftarrow \mathit{flip}\ x), \mathsf{proc}\ (z \leftarrow \mathit{flip}\ y) \\ \mapsto^* & \mathsf{msg}\ x'\ (\mathsf{b1} \cdot x''), \mathsf{proc}\ (y' \leftarrow \mathit{flip}\ x'), \mathsf{msg}\ y\ (\mathsf{b1} \cdot y'), \mathsf{proc}\ (z \leftarrow \mathit{flip}\ y) \end{array}$$

Because we have concurrent language, the two messages to the two processes (in the middle of the picture) can proceed independently:

$$\begin{array}{ll} & \mathsf{msg}\ x'\ (\mathsf{b1} \cdot x''), \mathsf{proc}\ (y' \leftarrow \mathit{flip}\ x'), \mathsf{msg}\ y\ (\mathsf{b1} \cdot y'), \mathsf{proc}\ (z \leftarrow \mathit{flip}\ y) \\ \mapsto^* & \mathsf{proc}\ (y'' \leftarrow \mathit{flip}\ x''), \mathsf{msg}\ y'(\mathsf{b0} \cdot y''), \mathsf{proc}\ (z' \leftarrow \mathit{flip}\ y'), \mathsf{msg}\ z\ (\mathsf{b0} \cdot z') \end{array}$$

6 Lazy Records Become External Choice

In the process language with shared memory, lazy records $\&_{i\in I}(i : \tau_i)$ write a continuation $K = (i \cdot x_i \Rightarrow P_i)_{i\in I}$ to memory. Here, in the message passing setting, a process providing a channel $x : \&_{i\in I}(i : \tau_i)$ *receives* one of the labels $j \in I$ and continuation channel $y : \tau_j$. It is dual to $\oplus_{i\in I}(i : \tau_i)$ in the sense that it just reverses the role of provider and client. We call it *external choice* since the client can make the choice which label j to send.

As for internal choice, the typing rules remain the same.

$$\frac{\Delta \Vdash P_i :: (y_i : \tau_i) \quad (\text{for all } i \in I)}{\Delta \Vdash \mathsf{case}\ x\ (i \cdot y_i \Rightarrow P_i)_{i \in I} :: (x : \&_{i \in I}(i : \tau_i))} \ \& R$$

$$\frac{(j \in I)}{1em]x : \&_{i \in I}(i : \tau_i) \Vdash x.(j \cdot y) :: (y : \tau_j)} \ \& L^0$$

Dynamically, the external choice is already handled with the rules

$$\begin{array}{lcll}
\mathsf{proc}\ (c.V) & \mapsto & \mathsf{msg}\ c\ V & \text{(send)} \\
\mathsf{proc}\ (\mathsf{case}\ c\ K), \mathsf{msg}\ c\ V & \mapsto & \mathsf{proc}\ (V \triangleright K) & \text{(receive)}
\end{array}$$

even though the direction of the message flow has changed: it now goes to the provider from the client.

7 Queues Revisited

Before revisiting queues, let's summarize the message-passing interpretation of the various type constructors. Note that the statics is unchanged from Lecture 23 and the dynamics (excepting only the identity) is presented in Section 4. Furthermore, each provider action implies a matching complementary client action.

Type	Provider Action	Continuation Channel
$x : \oplus_{i \in I}(i : \tau_i)$	send label j	$x' : \tau_j$
$x : \tau_1 \otimes \tau_2$	send channel $y : \tau_1$	$x' : \tau_2$
$x : 1$	send $\langle\,\rangle$	*none*
$x : \rho\alpha.\,\tau$	send fold	$x' : [\rho\alpha.\,\tau/\alpha]\tau$
$x : \&_{i \in I}(i : \tau_i)$	receive label j	$x' : \tau_j$
$x : \tau_1 \multimap \tau_2$	receive channel $y : \tau_1$	$x' : \tau_2$
$x : \delta\alpha.\,\tau$	receive fold	$x' : [\delta\alpha.\,\tau/\alpha]\tau$

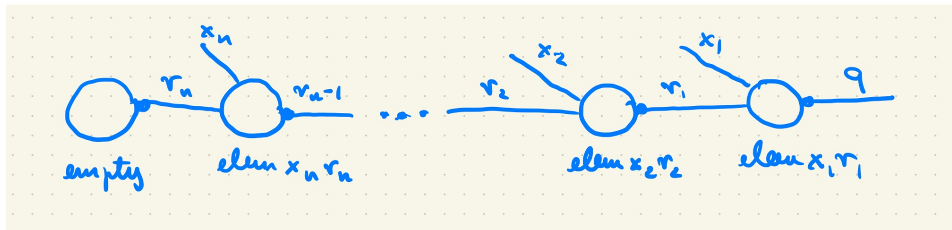
Here, the *corecursive types* $\delta\alpha.\,\tau$ is a lazy alternative to $\rho\alpha.\,\tau$ (see Exercise L20.3). Here reuses the fold constructor and pattern since it is dual to $\rho\alpha.\,\tau$.

A possible type for queues then is

$$\begin{array}{rl}
\mathit{queue}\ \alpha \cong & (\mathbf{enq} : \alpha \multimap \mathit{queue}\ \alpha) \\
& \&\ (\mathbf{deq} : \quad (\mathbf{none} : 1) \\
& \qquad\qquad \oplus (\mathbf{some} : \alpha \otimes \mathit{queue}\ \alpha))
\end{array}$$

The elements of the queue are channels of type α. Compared the previous incarnation of queues (Exercise L22.3), we close the channel and terminate the providing process if there is an attempt to dequeue from the empty queue. This is expressed in the type 1. In the linear setting, we would otherwise need a separate choice for the client to deallocate the queue, and that would only be possible if the queue is empty. We didn't make this explicit here, but if we define *queue* explicitly it would be as a corecursive type $\mathit{queue} = \lambda\alpha.\, \delta q.\, \ldots$.

We implement the queue as a *bucket brigade*, with the first element at the head of the queue. A new element to be enqueued is then passed all the way to the back of the queue. A queue with elements $x_1, \ldots x_n$ can be depicted as follows.

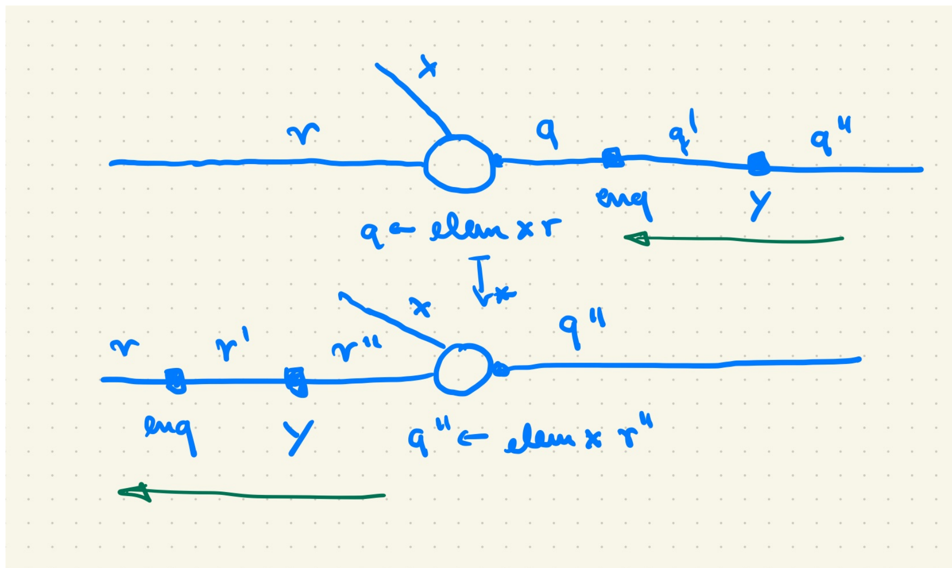


All the channels $r_1, \ldots, r_n$ have type *queue* α. We see we need two kinds of processes: one, *elem* x r that holds an element x, and *empty* marking the end of the queue.

We define the *elem* process.

$$
\begin{array}{l}
x : \alpha, r : \mathit{queue}\ \alpha \Vdash \mathit{elem} :: (q : \mathit{queue}\ \alpha) \\
q \leftarrow \mathit{elem}\ x\ r = \\
\quad \mathsf{case}\ q\ (\,\mathbf{enq} \cdot q' \Rightarrow \mathsf{case}\ q'\ (\langle y, q'' \rangle \Rightarrow r' \leftarrow r.(\mathbf{enq} \cdot r')\ ; \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad r'' \leftarrow r'.\langle y, r'' \rangle\ ; \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad q'' \leftarrow \mathit{elem}\ x\ r''\,) \\
\qquad\quad \mid \mathbf{deq} \cdot q' \Rightarrow q'' \leftarrow q''.\langle x, r \rangle\ ; \\
\qquad\qquad\qquad\qquad q'.(\mathbf{some} \cdot q'')\,)
\end{array}
$$

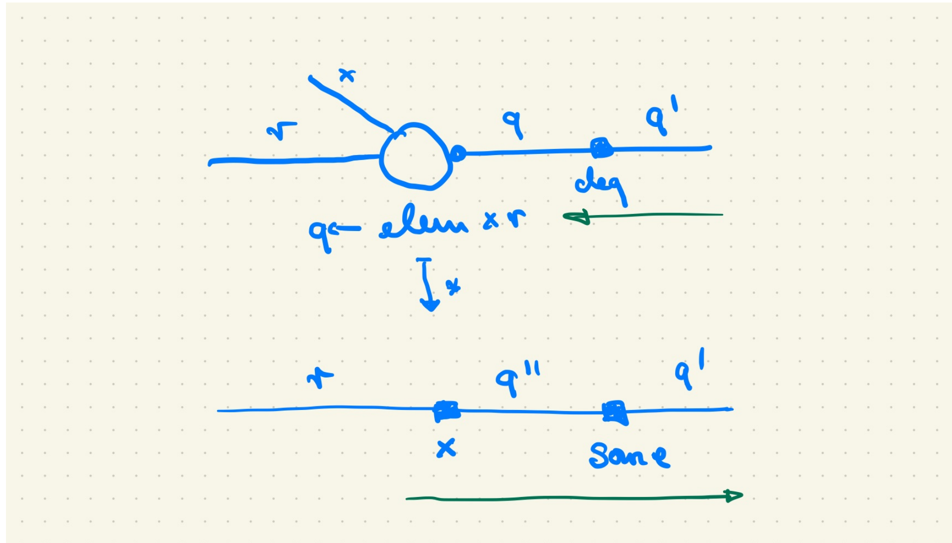
In case the client chooses to enqueue, the transition of the configuration can be depicted as follows.



As indicated by the green arrows, the messages here flow from right to left (from client to provider), unlike the bit flipping example where the flowed from left to right. As indicated in the table at the beginning of this section, messages of positive type flow from the provider to the client and message of negative type flow from the client to the provider.

In the case of the dequeue the provider has to respond to the client, so the direction of message flow changes. Moreover, the process holding the

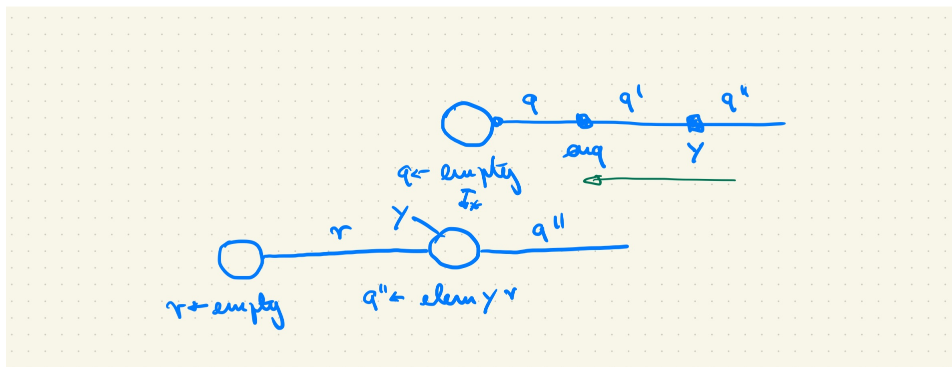
element x terminates, since that channel is returned to the client.



We did not have time to write this in lecture, but here is the code for the empty process.

$$
\begin{array}{l}
\cdot \Vdash \mathit{empty} :: (q : \mathit{queue}\ \alpha) \\
q \leftarrow \mathit{empty} = \\
\quad \mathsf{case}\ q\ (\,\mathbf{enq} \cdot q' \Rightarrow \mathsf{case}\ q'\ (\langle y, q'' \rangle \Rightarrow r \leftarrow \mathit{empty}\ ; \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad q'' \leftarrow \mathit{elem}\ y\ r\,) \\
\qquad\quad \mid \mathbf{deq} \cdot q' \Rightarrow q'' \leftarrow q''.\langle\,\rangle\ ; \\
\qquad\qquad\qquad\quad q'.(\mathbf{none} \cdot q'')\,)
\end{array}
$$

The action of the enqueue operation for the empty queue can be depicted as follows:



8 Move Becomes Forwarding

The one construct have not discussed yet is $x \leftarrow y$. In the shared memory interpretation this either *copied* the contents of y to x (in the nonlinear version) or *moved* the contents of y to x (in the linear version). Here it *forwards messages* from one channel to another and terminates.

$$\begin{array}{lcll} \mathsf{msg}\ y\ V, \mathsf{proc}\ (x \leftarrow y) & \mapsto & \mathsf{msg}\ x\ V & \text{(positive types)} \\ \mathsf{proc}\ (x \leftarrow y), \mathsf{msg}\ x\ V & \mapsto & \mathsf{msg}\ y\ V & \text{(negative types)} \end{array}$$

9 Rule Summary

The small values V, continuations K and typing rules can be found in the Lecture 23 Rule Sheet and remain unchanged.

The statics for configurations consists of the following rules.

$$\frac{\Delta \Vdash P :: (c : \tau)}{\Delta', \Delta \Vdash \mathsf{proc}\ P :: (\Delta', c : \tau)}\ \mathsf{tp/proc} \qquad \frac{\Delta \Vdash c.V :: (d : \tau)}{\Delta', \Delta \Vdash \mathsf{msg}\ c\ V :: (\Delta', d : \tau)}\ \mathsf{tp/msg}$$

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

The dynamics has very few rules, since we have factored out the (unchanged) passing of a value V to a continuation K.

$$\begin{array}{lcll} \mathsf{proc}\ (x \leftarrow P\ ;\ Q) & \mapsto & \mathsf{proc}\ ([c/x]P), \mathsf{proc}\ ([c/x]Q) & (\text{spawn; } c \text{ fresh}) \\ \mathsf{proc}\ (c.V) & \mapsto & \mathsf{msg}\ c\ V & (\text{send}) \\ \mathsf{msg}\ c\ V, \mathsf{proc}\ (\mathsf{case}\ c\ K) & \mapsto & \mathsf{proc}\ (V \triangleright K) & (\text{receive}) \\ \mathsf{msg}\ d\ V, \mathsf{proc}\ (c \leftarrow d) & \mapsto & \mathsf{msg}\ c\ V & (\text{pos. forward}) \\ \mathsf{proc}\ (c \leftarrow d), \mathsf{msg}\ c\ V & \mapsto & \mathsf{msg}\ d\ V & (\text{neg. forward}) \end{array}$$

References

[Agh85] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. PhD thesis, Massachusetts Institute of Technology, June 1985.

[Bou92] Gérard Boudol. Asynchrony and the π-calculus. Rapport de Recherche 1702, INRIA, Sophia-Antipolis, 1992.

[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.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–77, September 1992. Parts I and II.

[PP20] Klaas Pruiksma and Frank Pfenning. Back to futures. *CoRR*, abs/2002.04607, February 2020.

Lecture Notes on
Arithmetic Refinements

15-814: Types and Programming Languages
Frank Pfenning

Lecture 25
Thursday, December 3, 2020

1 Introduction

We have seen a number of benefits of types as an organizing principle in the design of programming languages. When statically checked, they provide the guarantees of *type safety* (summarized in the properties *preservation* and *progress*) and characterize the results of evaluation (via the *canonical forms theorem*). Static typing breaks down the properties of a whole program into individually checkable properties of the expressions, functions, and modules making up this program. They also allow us to express data abstraction and representation independence in a way that it can be enforced by a type-checker rather than just by convention.

So far, however, the properties that can be expressed in a type are rather rudimentary when compared to complete specifications. For example, any unary function on natural numbers has the type $nat \rightarrow nat$, including varieties such as the successor, predecessor, power of two, integer logarithm, etc.

In this lecture we explore how type theory supports the expression of a whole range of program properties, from *simple*, *recursive*, and *polymorphic* types we have seen so far to full "functional" program specifications. In the next lecture we will see how to go even further and capture *intensional* properties of programs, such as their computational complexity, entirely within the type.

A key idea is that of a *dependent type*, that is, a type that may *depend* on a value. General dependent type theory is highly expressive but also highly complex. It necessarily blends programming with theorem proving, which should not come as a surprise given our exploration of *Types as*

Propositions in Lecture 15. Even an introductory treatment would require a whole semester's worth of lectures. Despite this complexity, dependent type theories and related programming languages are becoming more widespread, including, for example Nuprl [C+86], Coq [BC04], Agda [Nor07], and Idris [Bra13].

Instead of tackling dependent type theories in their full generality, we explore *dependent refinement types* [XP99] that avoid the need for general theorem proving by restricting dependencies so that type-checking can be accomplished by a decision procedure. In this way we preserve the character of a programming language rather than asking the programmer to also prove their program correct. This comes at a cost: there are many program properties of interest that we will not be able to express.

2 Arithmetic Refinements

The particular instance of dependent refinement types we consider here are *arithmetic refinements*, that is, types may be *indexed* by arithmetic expressions. The exact language of arithmetic expressions is somewhat open-ended. For example, we may want to preserve decidability and restrict ourselves to *Presburger arithmetic* which has constants, addition, multiplication by a constant, and the propositions including conjunction, disjunction, negation, and quantification. Or we could allow more general expressions and "do our best" with heuristics for proving arithmetic expressions.

Instead of using integers $\mathbb{Z}$ we will use natural numbers $\mathbb{N}$, which is the same as $\mathbb{Z}$ where every variable x is constrained by $x \geq 0$. Here are some examples of arithmetically indexed types we can express:

list $\alpha\ n$	Lists of length n
bin n	Binary numbers of value n
nat n	Unary numbers of value n
stack $\alpha\ n$	Stack with n elements
incstream n	Increasing streams of natural numbers $\geq n$
tree $l\ u$	Trees of natural numbers x with $l < x < u$

In each of these cases, there are two challenges:

1. expressing the type itself so it accurately captures the property we care about, and

2. giving correct types to the functions that operate on elements of the type.

That's over and above the general challenge to tie together the language, the type checker, and the decision procedure (or heuristic algorithm) for deciding the validity of propositions in arithmetic.

3 Example: Lists Indexed by Length

We would like $\mathit{list}\ \alpha\ n$ to be a type. So *list* is now a function that maps types α and natural numbers n to types. One can formally describe this via so-called *kinds* for *type constructors* or *type families* as in the system F^ω [Gir71] or the Calculus of Constructions [CH88]. We avoid a full formalization of this since we would like to focus on programming aspects of indexed types. Let's start with a type

$\mathit{list}\ \alpha \cong (\mathbf{nil} : 1) + (\mathbf{cons} : \alpha \times \mathit{list}\ \alpha)$

Let's first consider the **cons** branch. If the whole list has length n, then the tail of the list has length $n - 1$:

$\mathit{list}\ \alpha\ n \cong (\mathbf{nil} : 1) + (\mathbf{cons} : \alpha \times \mathit{list}\ \alpha\ (n-1))$

We use the color blue to highlight all arithmetic expressions and propositions that belong to the refinement layer. The above is not quite correct because a list of length 0 may have a tail of length -1, so we need to enforce the $n - 1 \geq 0$ in case that list is nonempty. We express this with a type in the form of $\phi \wedge \tau$ where ϕ is a proposition from arithmetic. We often refer to ϕ as a *constraint*.

$\mathit{list}\ \alpha\ n \cong (\mathbf{nil} : 1) + (\mathbf{cons} : n > 0 \wedge \alpha \times \mathit{list}\ \alpha\ (n-1))$

Finally, in the alternative **nil** we must constrain n to be 0, since nil always represents a list of length zero.

$\mathit{list}\ \alpha\ n \cong (\mathbf{nil} : n = 0 \wedge 1) + (\mathbf{cons} : n > 0 \wedge \alpha \times \mathit{list}\ \alpha\ (n-1))$

Before we more rigorously express how do handle arithmetic propositions and types such a $\phi \wedge \tau$, let's examine how they will be used. Let's start with the *constructor functions nil* and *cons*. We would like to have

$\mathit{nil} : \forall \alpha.\ \mathit{list}\ \alpha\ 0$
$\mathit{nil} = \Lambda \alpha.\ \mathsf{fold}\ (\mathbf{nil} \cdot \langle\,\rangle)$

because *nil* constructs a list of length 0. Let's walk through the typing derivation of this.

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \alpha\ \mathit{type} \vdash (\mathbf{nil}\cdot\langle\,\rangle) : (\mathbf{nil} : 0 = 0 \wedge 1) + (\mathbf{cons} : 0 > 0 \wedge \alpha \times \mathit{list}\ \alpha\ (0-1))\end{array}}{\alpha\ \mathit{type} \vdash \mathsf{fold}\ (\mathbf{nil}\cdot\langle\,\rangle) : \mathit{list}\ \alpha\ 0}\ \mathsf{tp/fold}}{\cdot \vdash \Lambda\alpha.\, \mathsf{fold}\ (\mathbf{nil}\cdot\langle\,\rangle) : \forall\alpha.\, \mathit{list}\ \alpha\ 0}\ \mathsf{tp/tplam}$$

Note here that at the step where we unfolded the type we substituted 0 for n, because the second index to *list* is 0. Using the rule for sums, we find

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ \alpha\ \mathit{type} \vdash \langle\,\rangle : 0 = 0 \wedge 1\end{array}}{\alpha\ \mathit{type} \vdash (\mathbf{nil}\cdot\langle\,\rangle) : (\mathbf{nil} : 0 = 0 \wedge 1) + (\mathbf{cons} : 0 > 0 \wedge \alpha \times \mathit{list}\ \alpha\ (0-1))}\ \mathsf{tp/tag}}{\alpha\ \mathit{type} \vdash \mathsf{fold}\ (\mathbf{nil}\cdot\langle\,\rangle) : \mathit{list}\ \alpha\ 0}\ \mathsf{tp/fold}}{\cdot \vdash \Lambda\alpha.\, \mathsf{fold}\ (\mathbf{nil}\cdot\langle\,\rangle) : \forall\alpha.\, \mathit{list}\ \alpha\ 0}\ \mathsf{tp/tplam}$$

At this point we have to show that $0 = 0$, in general employing assumptions from the context (although there are none here). This is a new judgment from arithmetic, so we write it as $\Gamma \models \phi\ \mathit{true}$.

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ \cdot \models 0 = 0\ \mathit{true}\end{array} \quad \dfrac{}{\alpha\ \mathit{type} \vdash \langle\,\rangle : 1}\ \mathsf{tp/unit}}{\alpha\ \mathit{type} \vdash \langle\,\rangle : 0 = 0 \wedge 1}\ \mathsf{tp/and}}{\alpha\ \mathit{type} \vdash (\mathbf{nil}\cdot\langle\,\rangle) : (\mathbf{nil} : 0 = 0 \wedge 1) + (\mathbf{cons} : 0 > 0 \wedge \alpha \times \mathit{list}\ \alpha\ (0-1))}\ \mathsf{tp/tag}}{\dfrac{\alpha\ \mathit{type} \vdash \mathsf{fold}\ (\mathbf{nil}\cdot\langle\,\rangle) : \mathit{list}\ \alpha\ 0}{\cdot \vdash \Lambda\alpha.\, \mathsf{fold}\ (\mathbf{nil}\cdot\langle\,\rangle) : \forall\alpha.\, \mathit{list}\ \alpha\ 0}\ \mathsf{tp/tplam}}\ \mathsf{tp/fold}$$

Fortunately, $0 = 0$ is true, so this typing should be valid.

We observe here a new phenomenon, namely a typing rule that does not change the expression. It is in part this property which makes this system a system of *type refinement* rather than a full dependent type theory. In the general form:

$$\dfrac{\Gamma \models \phi\ \mathit{true} \qquad \Gamma \vdash e : \tau}{\Gamma \vdash e : \phi \wedge \tau}\ \mathsf{tp/and/i}$$

Note that not all the assumptions in Γ can actually be relevant to the truth of ϕ, but we will gloss over this detail here.

Next, there should be counterpart, tp/and/e. Our form of conjunction should be *positive* (as we will see shortly), so the elimination should in the form of a case-like rule. However, this is difficult because we do not want the expression e to change, but a case rule requires two expressions. Instead, we build the elimination rule into pattern matching. Recall from Lecture 12:

$$\begin{array}{llcl} \text{Expressions} & e & ::= & \ldots \mid \mathsf{case}\ e\ (bs) \\ \text{Patterns} & p & ::= & x \mid \langle p_1, p_2 \rangle \mid \langle\,\rangle \mid i \cdot p \mid \mathsf{fold}\ p \\ \text{Branches} & bs & ::= & \cdot \mid (p \Rightarrow e \mid bs) \end{array}$$

There were two relevant judgments beyond typing of expressions:

Matching: $\Gamma \vdash \tau \triangleright bs : \sigma$ which expresses a subject of type τ matches the branches bs, all of which have type σ.

Patterns: $\Gamma \Vdash p : \tau$ which expresses that pattern p has type τ.

This latter judgment was an early example of a *linear* judgment, because we wanted every variable in Γ to occur exactly once in p.

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau \triangleright bs : \sigma}{\Gamma \vdash \mathsf{case}\ e\ (bs) : \sigma}\ \mathsf{case}$$

$$\frac{\Gamma' \Vdash p : \tau \quad \Gamma, \Gamma' \vdash e : \sigma \quad \Gamma \vdash \tau \triangleright bs : \sigma}{\Gamma \vdash \tau \triangleright (p \Rightarrow e \mid bs) : \sigma}\ \mathsf{tp/bs/alt} \qquad \frac{}{\Gamma \vdash \tau \triangleright (\cdot) : \sigma}\ \mathsf{tp/bs/none}$$

$$\frac{}{x : \tau \Vdash x : \tau}\ \mathsf{pat/var} \qquad \frac{\Gamma_1 \Vdash p_1 : \tau_1 \quad \Gamma_2 \Vdash p_2 : \tau_2}{\Gamma_1, \Gamma_2 \Vdash \langle p_1, p_2 \rangle : \tau_1 \times \tau_2}\ \mathsf{pat/pair} \qquad \frac{}{\cdot \Vdash \langle\,\rangle : 1}\ \mathsf{pat/unit}$$

$$\frac{(k \in I) \quad \Gamma \Vdash p : \tau_k}{\Gamma \Vdash k \cdot p : \sum_{i \in I}(i : \tau_i)}\ \mathsf{pat/inject} \qquad \frac{\Gamma \Vdash p : [\rho\alpha.\,\tau/\alpha]\tau}{\Gamma \Vdash \mathsf{fold}\ p : \rho\alpha.\,\tau}\ \mathsf{pat/fold}$$

We now add to the pattern typing the rule

$$\frac{\Gamma, \phi\ \mathit{true} \Vdash p : \tau}{\Gamma \Vdash p : \phi \wedge \tau}\ \mathsf{pat/and}$$

In analogy with the typing rule for expressions, the pattern p does not change in this rule.

Now let's return to the constructor and see if we can type

$\mathit{cons} : \forall\alpha.\, \forall n.\, \alpha \to \mathit{list}\ \alpha\ n \to \mathit{list}\ \alpha\ (n+1)$

Already we note that we need a universal quantifier over index expressions. It's not difficult to image what the rules for the universal quantifier might look like:

$$\frac{\Gamma, n : \mathbb{N} \vdash e : \tau}{\Gamma \vdash \lambda n.\, e : \forall x.\, \tau}\ \mathsf{tp/nlam} \qquad \frac{\Gamma \vdash e : \forall n.\, \tau \quad \Gamma \vdash t : \mathbb{N}}{\Gamma \vdash e\ t : [t/x]\tau}\ \mathsf{tp/napp}$$

$$\frac{}{\lambda n.\, e\ \mathit{value}}\ \mathsf{val/nlam} \qquad \frac{e \mapsto e'}{e\ t \mapsto e'\ t}\ \mathsf{step/napp}_1 \qquad \frac{}{(\lambda n.\, e)\ t \mapsto [t/n]\, e}\ \mathsf{step/napp/nlam}$$

Here, t ranges over arithmetic terms. This is in effect the extension needed if we wanted to generalize the Curry-Howard interpretation of intuitionistic logic to include universal quantification, except that here we committed the quantification to be over natural numbers. We do not evaluate arithmetic terms because we think of them as serving the purpose of refined types, not additional computations. In fact, it is also possible to remove abstraction over and application to arithmetic terms from the language of expressions, but this complicates type-checking significantly.

Returning to the *cons* constructor for lists, here is what are hoping for.

$\mathit{cons} : \forall\alpha.\, \forall n.\, \alpha \to \mathit{list}\ \alpha\ n \to \mathit{list}\ \alpha\ (n+1)$
$\mathit{cons} = \Lambda\alpha.\, \lambda n.\, \lambda x.\, \lambda l.\, \mathsf{fold}\ (\mathbf{cons} \cdot \langle x, l\rangle)$

After discharging the abstractions into the context we arrive at the judgment

$$\alpha\ \mathit{type}, n : \mathbb{N}, x : \alpha, l : \mathit{list}\ \alpha\ n \vdash \mathsf{fold}\ (\mathbf{cons} \cdot \langle x, l\rangle) : \mathit{list}\ \alpha\ (n+1)$$

Unfolding the recursive type, this holds if we can show:

$\alpha\ \mathit{type}, n : \mathbb{N}, x : \alpha, l : \mathit{list}\ \alpha\ n \vdash$
$(\mathbf{cons} \cdot \langle x, l\rangle) : (\mathbf{nil} : \ldots) + (\mathbf{cons} : n+1 > 0 \wedge \alpha \times \mathit{list}\ \alpha\ ((n+1)-1))$

Selecting the **cons** alternative of the sum and noting that $n : \mathbb{N} \models n+1 > 0$ we arrive at

$$\alpha\ \mathit{type}, n : \mathbb{N}, x : \alpha, l : \mathit{list}\ \alpha\ n \vdash l : \mathit{list}\ \alpha\ ((n+1)-1)$$

Clearly, this should hold because for every $n : \mathbb{N}$ we have that $n = (n+1)-1$ and therefore also $\mathit{list}\ \alpha\ n = \mathit{list}\ \alpha\ ((n+1)-1)$. Here, we think of equality as meaning the two types are inhabited by exactly the same values (for every α and n).

To handle this rigorously, we should allow a rule of type conversion and some rules to formalize some notion of type equality (see Mini-Project 1.1 or [LBN17]). We should have at least the rule of *type conversion* and allow provably equal index terms to be used interchangeably.

$$\frac{\Gamma \vdash \tau = \tau' \quad \Gamma \vdash e : \tau'}{\Gamma \vdash e : \tau}\ \mathsf{tp/conv} \qquad \frac{\Gamma \vdash \tau = \tau' \quad \Gamma \models t = t'}{\Gamma \vdash \tau\ t = \tau'\ t'}\ \mathsf{conv/idx}$$

Different notions of type equality or subtyping are a vast subject, so we just pragmatically assume we have at least the two rules above to allow judgments such as the typing of *cons* and other examples.

4 Checking Recursive Functions

As a generic example of a recursive function, we consider typing the append function for lists. We repeat the indexed type of lists for reference and the expected type for *append*.

$$\mathit{list}\ \alpha\ n \cong (\mathbf{nil} : n = 0 \wedge 1) + (\mathbf{cons} : n > 0 \wedge \alpha \times \mathit{list}\ \alpha\ (n-1))$$

$$\mathit{append} : \forall \alpha.\, \forall n.\, \forall k.\, \mathit{list}\ \alpha\ n \rightarrow \mathit{list}\ \alpha\ k \rightarrow \mathit{list}\ \alpha\ (n+k)$$

We first write the definition, which is straightforwardly extended from the usual (unindexed) definition.

$$\begin{array}{l} \mathit{append} = \mathsf{rec}\ \mathit{append}.\, \Lambda\alpha.\, \lambda n.\, \lambda k.\, \lambda l_1.\, \lambda l_2. \\ \qquad\qquad \mathsf{case}\ l_1\ (\,\mathsf{fold}\ (\mathbf{nil} \cdot \langle\,\rangle) \Rightarrow l_2 \\ \qquad\qquad\qquad \mid \mathsf{fold}\ (\mathbf{cons} \cdot \langle x, l_1' \rangle) \Rightarrow \mathit{cons}\ \alpha\ ((n-1)+k)\ x\ (\mathit{append}\ \alpha\ (n-1)\ k\ l_1'\ l_2)\,) \end{array}$$

Here, the recursive call to append is at $n-1$ and k because $l_1' : \mathit{list}\ \alpha\ (n-1)$. This means the recursive call

$$\mathit{append}\ \alpha\ (n-1)\ k\ l_1'\ l_2 : \mathit{list}\ \alpha\ ((n-1)+k)$$

so the index argument to *cons* should be $(n-1)+k$. However, for this to be well-formed we need to know $n > 0$. Because $l_1 : \mathit{list}\ \alpha\ n$, when we match l_1 against $\mathsf{fold}\ (\mathbf{cons} \cdot \langle x, l' \rangle)$ we find that

$$n > 0\ \mathit{true}, x : \alpha, l' : \mathit{list}\ \alpha\ (n-1) \Vdash \mathsf{fold}\ (\mathbf{cons} \cdot \langle x, l' \rangle) : \mathit{list}\ \alpha\ n$$

and we obtain the necessary assumption $n > 0$.

The result of applying the *cons* constructor will have type $((n-1)+k)+1$ and we have

$$n : \mathbb{N}, k : \mathbb{N}, n > 0 \ \mathit{true} \models ((n-1)+k)+1 = n+k \ \mathit{true}$$

where $n+k$ is the required result type.

It remains to check the case of **nil**. In that case, pattern matching l_1 against *fold* $(\mathbf{nil} \cdot \langle\,\rangle)$ obtains the information that $n = 0$. Then $l_2 : \mathit{list}\ \alpha\ k$ and $n : \mathbb{N}, k : \mathbb{N}, n = 0\ \mathit{true} \models k = 0 + k\ \mathit{true}$ gives us the type-correctness of the first branch.

It is a straightforward variation on this theme to check, for example, that *reverse* preserves the length of the list.

5 Contradictory Constraints

Consider the type *list* α 1. It describes a list with just one element. Consequently, the following case statement should be cover all possible cases:

$\mathit{the} : \forall\alpha.\ \mathit{list}\ \alpha\ 1 \to \alpha$
$\mathit{the} = \Lambda\alpha.\ \lambda l.\ \mathsf{case}\ l\ (\mathsf{fold}\ (\mathbf{cons} \cdot \langle x, \mathsf{fold}\ (\mathbf{nil} \cdot \langle\,\rangle)\rangle) \Rightarrow x)$

For the sake of argument, say we added a nil branch:

$\mathit{the} : \forall\alpha.\ \mathit{list}\ \alpha\ 1 \to \alpha$
$\mathit{the} = \Lambda\alpha.\ \lambda l.\ \mathsf{case}\ l\ (\ \mathsf{fold}\ (\mathbf{cons} \cdot \langle x, \mathsf{fold}\ (\mathbf{nil} \cdot \langle\,\rangle)\rangle) \Rightarrow x$
$\qquad\qquad\qquad\qquad \mid \mathsf{fold}\ (\mathbf{nil} \cdot \langle\,\rangle) \Rightarrow e\ )$

Playing through the rules for pattern matching we note that when we reach e it will be checked with

$$\alpha\ \mathit{type}, l : \mathit{list}\ \alpha\ 1, 0 = 1\ \mathit{true} \vdash e : \alpha$$

Of course, we have no way to return an element of the parameter type α. But we shouldn't have to since we are in an impossible branch! There is no value v such that $v : \mathit{list}\ \alpha\ 1$ and $v = \mathsf{fold}\ (\mathbf{nil}\ \langle\,\rangle)$. So type-checking this branch should succeed because it is impossible for it ever to be taken. We therefore have the rule

$$\frac{\Gamma \models \bot\ \mathit{true}}{\Gamma \vdash e : \tau}\ \mathsf{tp/contra}$$

By such reasoning, when the exhaustiveness of pattern matching is checked then the absence of impossible branches should not be flagged, just like in the first definition of *the*.

6 Binary Numbers and Singleton Types

Numbers in binary representation present some new challenges. We would like *bin* n be the type of binary numbers with value n. Note that the binary numbers (or unary numbers) as recursive types are different from the natural numbers $\mathbb{N}$ used in the index domain. In a fully dependent type system, this does not have to be so, but it is more difficult to obtain the power of the decision procedures for Presburger arithmetic.

Here is the first attempt

$$\begin{aligned} \mathit{bin}\ n \cong\ & \quad (\mathbf{e} : n = 0 \wedge 1) \\ & + (\mathbf{b0} : \mathsf{even}\ n \wedge \mathit{bin}\ (n/2)) \\ & + (\mathbf{b1} : \mathsf{odd}\ n \wedge \mathit{bin}\ ((n-1)/2)) \end{aligned}$$

But even though $\mathsf{even}\ n \triangleq \exists k.\, 2k = n$ can be defined in Presburger arithmetic, integer division can not (directly). So we introduce an existential quantifier to span the condition and the recursive occurrence of the type.

$$\begin{aligned} \mathit{bin}\ n \cong\ & \quad (\mathbf{e} : n = 0 \wedge 1) \\ & + (\mathbf{b0} : \exists k.\, n = 2k \wedge \mathit{bin}\ k) \\ & + (\mathbf{b1} : \exists k.\, n = 2k + 1 \wedge \mathit{bin}\ k) \end{aligned}$$

If we would like to prevent leading zeros in the representation we can constrain k (or n) in the case of a bit 0.

$$\begin{aligned} \mathit{bin}\ n \cong\ & \quad (\mathbf{e} : n = 0 \wedge 1) \\ & + (\mathbf{b0} : \exists k.\, k > 0 \wedge n = 2k \wedge \mathit{bin}\ k) \\ & + (\mathbf{b1} : \exists k.\, n = 2k + 1 \wedge \mathit{bin}\ k) \end{aligned}$$

With the particular restriction we see two phenomena that didn't exist with lists:

1. There are values $v : \mathit{bin}$ such that there is no index n such that $v : \mathit{bin}\ n$. In other words, some values (namely those with leading zeros) are no longer well-typed, sharpening the canonical form theorem.

2. Each type *bin* n is inhabited by exactly one value, namely *the* binary representation of n. We call such a type a *singleton type*, characterizing its value precisely.

We do not write any programs over refined binary numbers; see Exercise 1, but we can state the types of the constructors:

$e : \mathit{bin}\ 0$
$b0 : \forall n.\, n > 0 \supset \mathit{bin}\ n \to \mathit{bin}\ (2n)$
$b1 : \forall n.\, \mathit{bin}\ n \to \mathit{bin}\ (2n+1)$

We see that the type of *b0* requires a constraint implication. Perhaps that is not so surprising since we have a $\exists/\forall$ duality but also related $\wedge/\supset$ duality. We complete our language in the next section to include the additional quantifier.

7 Completing the Language

We summarize the language and supply the rules we have omitted so far. As mentioned before, the language of arithmetic terms and propositions is somewhat open-ended but with certain extensions we will lose decidability.

Types	τ	$::=$	$\ldots \mid \phi \wedge \tau \mid \phi \supset \tau \mid \exists n.\, \tau \mid \forall n.\, \tau$
Expressions	e	$::=$	$\ldots \mid \langle t, e\rangle \mid \lambda n.\, e \mid e\ t$
Patterns	p	$::=$	$\ldots \mid \langle n, p\rangle$
Constants	c	$::=$	$0 \mid 1 \mid \ldots$
Arith. Terms	t	$::=$	$c \mid t_1 + t_2 \mid t_1 - t_2 \mid c\,t \mid \ldots$
Arith. Props	ϕ	$::=$	$t_1 = t_2 \mid t_1 > t_2 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \supset \phi_2 \mid \top \mid \bot \mid \exists n.\, \phi \mid \forall n.\, \phi$
Contexts	Γ	$::=$	$\ldots \mid \Gamma, n : \mathbb{N} \mid \Gamma, \phi\ \mathit{true}$

Besides the arithmetic entailment $\Gamma \models \phi\ \mathit{true}$ we also use the judgment $\Gamma \vdash t : \mathbb{N}$, which checks that t is well-formed and that $\Gamma \models t \geq 0\ \mathit{true}$.

$$\frac{\Gamma \models \phi\ \mathit{true} \quad \Gamma \vdash e : \tau}{\Gamma \vdash e : \phi \wedge \tau}\ \mathsf{tp/and/i} \qquad \frac{\Gamma, \phi\ \mathit{true} \Vdash p : \tau}{\Gamma \Vdash p : \phi \wedge \tau}\ \mathsf{pat/and}$$

$$\frac{\Gamma, \phi\ \mathit{true} \vdash e : \tau}{\Gamma \vdash e : \phi \supset \tau}\ \mathsf{tp/imp/i} \qquad \frac{\Gamma \vdash e : \phi \supset \tau \quad \Gamma \models \phi}{\Gamma \vdash e : \tau}\ \mathsf{tp/imp/e}$$

$$\frac{\Gamma \vdash t : \mathbb{N} \quad \Gamma \vdash e : [t/n]\,\tau}{\Gamma \vdash \langle t, e\rangle : \exists n.\, \tau}\ \mathsf{exists/i} \qquad \frac{\Gamma, n : \mathbb{N} \Vdash p : \tau}{\Gamma \Vdash \langle n, p\rangle : \exists n. \tau}\ \mathsf{pat/exists}$$

$$\frac{\Gamma, n : \mathbb{N} \vdash e : \tau}{\Gamma \vdash \lambda n.\, e : \forall n.\, \tau}\ \mathsf{forall/i} \qquad \frac{\Gamma \vdash e : \forall n.\, \tau \quad \Gamma \vdash t : \mathbb{N}}{\Gamma \vdash e\,t : [t/n]\tau}\ \mathsf{forall/e}$$

$$\frac{\Gamma \models \bot\ \mathit{true}}{\Gamma \vdash e : \tau}\ \mathsf{tp/contra}$$

8 Some Additional Types

In this section we represent the encoding of some additional types using indexed refinement.

$$\begin{array}{l} \textit{incstream}\ n \cong \exists k.\, k \geq n \wedge (\mathbf{hd} : \textit{bin}\ k)\ \&\ (\mathbf{tl} : \textit{incstream}\ k) \\ \textit{stack}\ \alpha\ n \cong (\mathbf{push} : \alpha \to \textit{stack}\ \alpha\ (n+1)) \\ \qquad\qquad\quad \&\ (\mathbf{pop} : (\mathbf{none} : n = 0 \wedge 1) + (\mathbf{some} : n > 0 \wedge \alpha \times \textit{stack}\ \alpha\ (n-1))) \\ \textit{tree}\ l\ u \cong (\mathbf{leaf} : 1) + (\mathbf{node} : \exists n.\, l < n \wedge n < u \wedge \textit{tree}\ l\ n \times \textit{bin}\ n \times \textit{tree}\ n\ u) \end{array}$$

Exercises

Exercise 1 Using the example for natural numbers in binary form, explore the following functions. Highlight in each case the constraints that would have to be checked to verify type correctness. Which of these are checkable in Presburger arithmetic?

(i) Provide the type and write the implementation of the successor function.

(ii) Provide the type and write the implementation of the predecessor function on positive numbers.

(iii) Provide the type and write the implementation of the addition functions.

(iv) Provide the type and write the implementation of the exponential function specified mathematically with $\mathsf{exp2}(x) = 2^x$.

Exercise 2 Give a definition for natural numbers in unary form so that every type *nat* n is the singleton with the representation of n in unary form. Then revisit the functions in Exercise 1.

Exercise 3 Write an implementation of stacks as specified in Section 8, perhaps recycling an earlier implementation, explicit stating the indexed types for every function you need. Isolate the constraints that have to be checked in each case and verify that they are true.

Exercise 4 Specify the type of queues with n elements and revisit your implementation of queues with two stacks. Does it type-check? You may assume that the stack operations check according to the type in Section 8 and Exercise 3.

Exercise 5 Consider the *ordered trees of natural numbers* as specified in Section 8. These trees can be seen as an efficient representation of sets of natural numbers, assuming they are sufficiently balanced (a requirement we ignore in this exercise).

Attempt each of the following steps with the goal of completing them all to arrive at an implementation where the ordering invariant of binary search trees is enforced via type-checking. The ordering invariant states that for a node with element n, all elements in the left subtree are strictly less than n and all elements in the right subtree are strictly greater than n.

(i) Provide the type and implementation of *empty* for the empty binary search tree.

(ii) Provide the type and write a function *lookup* to determine if an element is in a given tree.

(iii) Provide the type and write a function *insert* to insert an element into a given tree.

(iv) Write any functions you may need on binary numbers, making sure they type-check.

(v) Discuss any difficulties or limitations with refinement types you encountered in parts (i)–(iii).

Exercise 6 Instead of the ordering invariant from Exercise 5 we may wish to verify the balance invariant for the case of AVL trees. Specify a different type of tree with sufficient information to express the balance invariant, write implementations of *lookup* and *insert* (including the necessary rotations to restore the balance invariant) and explore whether the implementation type-checks.

References

[BC04] Yves Bertot and Pierre Castéran. *Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[Bra13] Edwin Brady. Idris, a general purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013.

[C⁺86] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[CH88] Thierry Coquand and Gerard Huet. The Calculus of Constructions. *Information and Computation*, 76(2–3):95–120, 1988.

[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.

[LBN17] Jay Ligatti, Jeremy Blackburn, and Michael Nachtigal. On subtyping-relation completeness, with an application to iso-recursive types. *ACM Transactions on Programming Languages and Systems*, 39(4):4:1–4:36, March 2017.

[Nor07] Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology, 2007.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In A. Aiken, editor, *Conference Record of the 26th Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM Press, January 1999.

Lecture Notes on
Ergometric Types

15-814: Types and Programming Languages
Frank Pfenning

Lecture 26
Tuesday, December 8, 2020

1 Introduction

In the last lecture we introduced arithmetic refinements to capture some internal invariants of data structures, including, for example, characterizing their size. This information is critical to capture computational complexity of functions or, in our case, processes. The key goal is to capture this information in *types*, using the kind of conceptual tools and techniques we have developed so far.

We carry out our development in the context of message-passing concurrency, capturing the total amount of *work* that a configuration of communicating processes does. This may also be called the *sequential complexity* of the program, because it is the complexity if all operations are performed one at a time. Its counterpart is the *span*, or the *parallel complexity*, which arises from the assumption that all operations that can be done in parallel (on an abstract machine with arbitrarily many processors), will in fact be done in parallel. Once both work and span are known one can derive some bounds on running time based on the number of available processors using Brent's Theorem [Bre74]. Because the last true lecture in this course is cancelled in favor of a seminar, we encourage you to read up on how to think about parallel complexity in this context [DHP18a]. The development we present in this lecture is a reformulation of a prior ergometric type system [DHP18b], close to the one implemented in Rast [DP20].

The ingredients that come together to give a sound ergometric type system are the following:

Cost Model: We do not want an abstract type system to depend on the particulars of a machine or a compiler. Instead we want to parametrize the ergometric types with a *cost model*. Depending on the model chosen by a programmer or language designer, concrete cost may be different but the type system will remain the same.

Potential: When analyzing code we do not "*count up*" to track the cost, but we provide each function or process with some potential $q \geq 0$ and have it "*count down*" the remaining potential. This simplifies the statics and the dynamics in the formalization.

Transfer of Potential: A process may decide not to perform some operations itself, but instead transfer some of its potential to other processes. This allows us to perform *amortized analysis*, which is a common and powerful technique for characterizing the computational complexity of a function or process.

Linearity: Once potential is invested in processes or data structures, it is important that it not be duplicated. This is where the nature of substructural type systems in general and linear ones in particular comes to the rescue because it prevents just such duplication.

For all these reasons we chose *message-passing concurrency* as presented in Lecture 24 as our basis, but it could easily be adapted to linear functional or linear shared memory code (Lecture 23). For nonlinear programs we need what we called a *quasi-linear type system* in Mini-Project 2.1 or *sharing* in Resource-Aware ML (RAML, see [HAH12]).

2 Cost Model

The cost model is represented by a transformation from an original process P to a new process P' that insert instances of the construct $\mathsf{work}\ w$. As an example, consider the cost model where each send operation costs 1 erg (= 1 unit of work). Then we transform every process $x.V$ to $\mathsf{work}\ 1 \ ; x.V$. As another example, in a model where each call (recursive or not) costs 1 erg, we would transform each call $x \leftarrow f\, y_1 \ldots y_n$ to $\mathsf{work}\ 1 \ ; x \leftarrow f\, y_1 \ldots y_n$.

We omit from the cost model the index terms and quantifiers introduced for expressions in the last lecture. That's because we think of their purpose as *static*, more precisely capturing properties of the data and functions we work with, rather than *dynamic*, contributing to the outcome of computation itself.

For this lecture we remain within the cost model where every send costs 1 erg. Because every channel is linear, every send is matched by a receive (except perhaps at the top level), so also counting receives would just double the cost.

In addition to the predefined costs, the programmer may also insert $\mathsf{work}\ n$ explicitly. This turns out to be useful in two circumstances. One is that different branches of a process require different amount work, in which case the programmer may equalize them. Another is that the programmer may wish to choose the "*free*" cost model where all operations are free, and then insert explicit $\mathsf{work}\ w$ to mark whatever they would like to count.

3 Statics

Isolating work in a separate construct $\mathsf{work}\ w$ has the nice consequence that we update the rule in a simple and systematic way to track the potential available to a process. The new judgment has the form

$$\Delta \Vdash^{q} P :: (x : \tau)$$

which means that process P with potential q uses the channels in Δ (according to their type) and provides channel $x : \tau$. The rules are derived from those in the Lecture 23 Rule Set, except that variables are no longer annotated as read or write since we are working in a message-passing setting here.

The one new rule is the one that consumes potential by doing work.

$$\frac{\Delta \Vdash^{q} P :: (x : \tau)}{\Delta \Vdash^{w+q} \mathsf{work}\ w \ ; P :: (x : \tau)}\ \mathsf{tp/work}$$

Because w and q must be natural numbers, for this rule to apply there must be at least w potential available. Otherwise, the process $\mathsf{work}\ w \ ; P$ cannot be typed.

The remaining rules are derived in a systematic way from the prior rules. For rules with zero premises, we require the potential to be 0. This is because we want to track the potential exactly, instead of an upper bound. For example:

$$\frac{}{\cdot \Vdash^{0} x.\langle\rangle :: (x : 1)}\ \mathsf{send/unit} \qquad \frac{}{y : \tau \Vdash^{0} x \leftarrow y :: (x : \tau)}\ \mathsf{forward}$$

The rule that spawns a new process must split the potential between the two processes.

$$\frac{\Delta \Vdash^{r} P :: (x : \tau) \quad \Delta', x : \tau \Vdash^{q} Q :: (z : \sigma)}{\Delta, \Delta' \Vdash^{r+q} (x \overset{r}{\leftarrow} P \;;\; Q) :: (z : \sigma)}\ \mathsf{spawn}$$

In the notation we note the potential that is imparted on the freshly spawned process. This is not strictly necessary, but it simplifies the dynamics.

For the remaining rules, the potential is preserved to all premises. For internal choice ($\oplus$) or external choice ($\&$) this may be surprising at first, but remember that at runtime exactly one branch will be chosen, based on the message received. This branch should have the full potential of the case: no more and no less. Here are two example rules:

$$\frac{\Delta, y_i : \tau_i \Vdash^{q} P_i :: (z : \sigma) \quad (\text{for all } i \in I)}{\Delta, x : \oplus_{i \in I}(i : \tau_i) \Vdash^{q} \mathsf{case}\ x\ (i \cdot y_i \Rightarrow P_i)_{i \in I} :: (z : \sigma)}\ \mathsf{recv/label}$$

$$\frac{\Delta, y : \tau \Vdash^{q} P :: (z : \sigma)}{\Delta \Vdash^{q} \mathsf{case}\ x\ (\langle y, z \rangle \Rightarrow P) :: (x : \tau \multimap \sigma)}\ \mathsf{recv/channel}$$

4 Dynamics

The dynamics is adapted from the one in Lecture 24 on message-passing concurrency. Process objects carry potential, while messages do not. Since all the potentials are checked statically, before the program is ever executed, these potential annotations are strictly necessary, but they play a key role in the proofs of progress and preservation.

$$\text{Configurations} \quad \mathcal{C} \quad ::= \quad \mathsf{proc}^{q}\ P \mid \mathsf{msg}\ c\ V \mid \mathcal{C}_1, \mathcal{C}_2 \mid (\cdot)$$

The rules adapt straightforwardly, with the first rule being new.

$$\begin{array}{lcll}
\mathsf{proc}^{w+q}\ (\mathsf{work}\ w \;;\; P) & \mapsto & \mathsf{proc}^{q}\ P & \\
 & & & \\
\mathsf{proc}^{r+q}\ (x \overset{r}{\leftarrow} P \;;\; Q) & \mapsto & \mathsf{proc}^{r}\ ([c/x]P), \mathsf{proc}^{q}\ ([c/x]Q) & (\text{spawn; } c \text{ fresh}) \\
\mathsf{proc}^{0}\ (c.V) & \mapsto & \mathsf{msg}\ c\ V & (\text{send}) \\
\mathsf{msg}\ c\ V, \mathsf{proc}^{q}\ (\mathsf{case}\ c\ K) & \mapsto & \mathsf{proc}^{q}\ (V \triangleright K) & (\text{receive}) \\
\mathsf{msg}\ d\ V, \mathsf{proc}^{0}\ (c \leftarrow d) & \mapsto & \mathsf{msg}\ c\ V & (\text{pos. forward}) \\
\mathsf{proc}^{0}\ (c \leftarrow d), \mathsf{msg}\ c\ V & \mapsto & \mathsf{msg}\ d\ V & (\text{neg. forward})
\end{array}$$

Note that dynamically a process would not be able to make a transition if it didn't have sufficient potential to do the work. Therefore, the progress theorem expresses that there will always be sufficient potential. The preservation theorem expresses that the overall potential in a configuration plus the amount of work performed remains invariant. The transitions cannot create new potential out of thin air, but they also cannot drop any existing potential. See Exercise 1 for further thoughts.

5 Example: Lists

As a first example we consider the cost of list constructors. We ignore here the cost of fold, primarily because the prior work [DHP18a, DP20] uses so-called *equirecursive types* where a type is considered equal with its unfolding so no fold message is actually necessary. It is not difficult to update the examples with such messages.

$\mathit{list}\ \alpha = (\mathbf{nil} : 1) \oplus (\mathbf{cons} : \alpha \otimes \mathit{list}\ \alpha)$

A prior, we would expect the empty list to require 2 messages ($\mathbf{nil}$ and $\langle\,\rangle$), which is also the case for *cons* ($\mathbf{cons}$ and x).

First the regular types.

$$\begin{array}{l} \alpha\ \mathit{type} \Vdash \mathit{nil} :: (l : \mathit{list}\ \alpha) \\ l \leftarrow \mathit{nil}\ \alpha = l' \leftarrow l'.\langle\,\rangle\ ; \\ \qquad\qquad\quad l.(\mathbf{nil} \cdot l') \end{array}$$

$$\begin{array}{l} \alpha\ \mathit{type}, x : \alpha, l : \mathit{list}\ \alpha \Vdash \mathit{cons} :: (k : \mathit{list}\ \alpha) \\ k \leftarrow \mathit{cons}\ \alpha\ x\ l = k' \leftarrow k'.\langle x, l\rangle\ ; \\ \qquad\qquad\qquad\quad k.(\mathbf{cons} \cdot k') \end{array}$$

Next we apply the cost model. We also annotate the process definitions with the potential these processes require.

$$\begin{array}{l} \alpha\ \mathit{type} \Vdash^{2} \mathit{nil} :: (l : \mathit{list}\ \alpha) \\ l \overset{2}{\leftarrow} \mathit{nil}\ \alpha = l' \overset{1}{\leftarrow} \mathsf{work}\ 1\ ; l'.\langle\,\rangle\ ; \\ \qquad\qquad\quad \mathsf{work}\ 1\ ; l.(\mathbf{nil} \cdot l') \end{array}$$

$$\begin{array}{l} \alpha\ \mathit{type}, x : \alpha, l : \mathit{list}\ \alpha \Vdash^{2} \mathit{cons} :: (k : \mathit{list}\ \alpha) \\ k \overset{2}{\leftarrow} \mathit{cons}\ \alpha\ x\ l = k' \overset{1}{\leftarrow} \mathsf{work}\ 1\ ; k'.\langle x, l\rangle\ ; \\ \qquad\qquad\qquad\quad \mathsf{work}\ 1\ ; k.(\mathbf{cons} \cdot k') \end{array}$$

It is easy to verify that the these definitions and types for the definitions are ergometrically correct.

6 Exploiting Arithmetic Refinements

In the next example we will assign a type to the *append* process for two lists. Clearly, this should be proportional to the size of the first list l_1, because when we reach its end we just forward to the second list l_2.

$$\mathit{list}\ \alpha\ n = (\mathbf{nil} : n = 0 \wedge 1) \oplus (\mathbf{cons} : n > 0 \wedge \alpha \otimes \mathit{list}\ \alpha (n-1))$$

We propose the following type

$$\alpha\ \mathit{type}, n : \mathbb{N}, m : \mathbb{N}, l_1 : \mathit{list}\ \alpha\ n, l_2 : \mathit{list}\ \alpha\ m \Vdash^{2n} \mathit{append} :: (k : \mathit{list}\ \alpha\ (n+m))$$

because the *append* process will have to send two messages (**cons** and an element x) for each element of the list l_1 which has length n. Indeed, this is easy to check. We have annotated lines with the known arithmetic constraints (in blue) and also the remaining potential (in red).

$$\begin{array}{ll}
k \overset{2n}{\leftarrow} \mathit{append}\ \alpha\ n\ m\ l_1\ l_2 = & \%\ 2n \\
\quad \mathsf{case}\ l_1\ (\ \mathbf{nil} \cdot \langle\,\rangle \Rightarrow k \leftarrow l_2 & \%\ n = 0,\ 2n = 0 \\
\qquad\qquad \mid \mathbf{cons} \cdot \langle x, l_1' \rangle \Rightarrow & \%\ n > 0,\ 2n \\
\qquad\qquad\quad k' \overset{2(n-1)}{\leftarrow} \mathit{append}\ \alpha\ (n-1)\ m\ l_1'\ l_2\ ; & \%\ 2n - 2(n-1) = 2 \\
\qquad\qquad\quad k \overset{2}{\leftarrow} \mathit{cons}\ \alpha\ ((n-1)+m)\ x\ k'\ ) & \%\ 2 - 2 = 0
\end{array}$$

As a second, similar example we consider *reverse* which calls *revapp* as an auxiliary process. *revapp* moves all the elements from the first list onto the second (the accumulator), to be returned at the end.

$$\alpha\ \mathit{type}, n : \mathbb{N}, m : \mathbb{N}, l : \mathit{list}\ \alpha\ n, \mathit{acc} : \mathit{list}\ \alpha\ m \Vdash^{2n} \mathit{revapp} :: (k : \mathit{list}\ \alpha\ (n+m))$$

$$\begin{array}{ll}
k \overset{2n}{\leftarrow} \mathit{revapp}\ \alpha\ n\ m\ l\ \mathit{acc} = & \%\ 2n \\
\quad \mathsf{case}\ l\ (\ \mathbf{nil} \cdot \langle\,\rangle \Rightarrow k \leftarrow \mathit{acc} & \%\ n = 0,\ 2n = 0 \\
\qquad\qquad \mid \mathbf{cons} \cdot \langle x, l' \rangle \Rightarrow & \%\ n > 0,\ 2n \\
\qquad\qquad\ \mathit{acc}' \overset{2}{\leftarrow} \mathit{cons}\ \alpha\ m\ x\ \mathit{acc}\ ; & \%\ 2n - 2 \\
\qquad\qquad\ k \overset{2(n-1)}{\leftarrow} \mathit{append}\ \alpha\ (n-1)\ (m+1)\ l'\ \mathit{acc}'\ ) & \%\ 2n - 2 - 2(n-1) = 0
\end{array}$$

$$\alpha\ \mathit{type}, n : \mathbb{N}, l : \mathit{list}\ \alpha\ n \Vdash^{2n+2} \mathit{reverse} :: (k : \mathit{list}\ \alpha\ n)$$

$$\begin{array}{ll}
k \overset{2n+2}{\leftarrow} \mathit{reverse}\ \alpha\ n\ l = & \%\ 2n + 2 \\
\quad \mathit{acc} \overset{2}{\leftarrow} \mathit{nil}\ \alpha\ ; & \%\ 2n \\
\quad k \overset{2n}{\leftarrow} \mathit{revapp}\ \alpha\ n\ 0\ l\ \mathit{acc} & \%\ 0
\end{array}$$

7 Transferring Potential

A key operation in amortized analysis is to be able to transfer potential between functions or processes. For this purpose we need two new type operators: one to send and one to receive potential (taking the point of view of the provider). We write $\triangleright^q \tau$ for sending potential q (a positive type) and $\triangleleft^q \tau$ for receiving potential q (a negative type).

$$\begin{array}{llll} \text{Types} & \tau & ::= & \ldots \mid \triangleright^q \tau \mid \triangleleft^q \tau \\ \text{Small Values} & V & ::= & \ldots \mid \mathsf{pot}\ q\ x \\ \text{Continuations} & K & ::= & \ldots \mid (\mathsf{pot}\ q\ x \Rightarrow P) \end{array}$$

The potential q in the receiving continuation is *not* a variable: we must be able to predict statically, via the type, how much potential is being received. Due to the duality between positive and negative types, we only have one new small value and one new continuation. However, we need new rules since the generic ones do not account for potential transfer.

$$\begin{array}{lll} \mathsf{proc}^q\ (c.(\mathsf{pot}\ q\ c')) & \mapsto & \mathsf{msg}\ c\ (\mathsf{pot}\ q\ c') \\ \mathsf{msg}\ c\ (\mathsf{pot}\ q\ c'), \mathsf{proc}^r\ (\mathsf{case}\ c\ (\mathsf{pot}\ q\ x \Rightarrow P)) & \mapsto & \mathsf{proc}^{r+q}\ ([c'/x]P) \end{array}$$

The generic rules from Section 4 remain the same.

8 Example: A Binary Counter

Incrementing a binary counter may flip as few as one bit (the lowest, if it is 0) and as many as there are bits in the number (if they are all 1). Amortized analysis tells us that n increments starting from 0 will flip at most $2n$ bits. In this section we have the type-checker prove this for us.

We represent each bit in the binary number by a process, *bit0* or *bit1*, plus one process *emp* for the end of the bit string. The client interacts with the lowest bit, sending it **inc** messages (eliding other parts of the interface that may be present). Every message by a process corresponds to exactly one bit flip, so counting the number of messages as our default cost model does is appropriate. Ignoring potential transfer, the type would be a an external choice with just one alternative.

$$\mathit{ctr} = (\mathbf{inc} : \mathit{ctr})\ \&\ ()$$

The key idea of this amortized analysis is that each *bit1* process maintains one unit of potential that it can use to send the carry bit when it is incremented. Furthermore, the client sends not only the increment message, but

an additional unit of potential. If the lowest bit is a *bit0*, it is flipped to *bit1* and the extra unit stored. If the lowest bit is *bit1* it becomes a *bit0*, sends the increment message representing the carry, plus the required additional unit of potential. In code:

$$
\begin{array}{l}
\mathit{ctr} = (\mathbf{inc} : \triangleleft^{1}\, \mathit{ctr}) \\[4pt]
y : \mathit{ctr} \Vdash^{0} \mathit{bit0} :: (x : \mathit{ctr}) \\
y : \mathit{ctr} \Vdash^{1} \mathit{bit1} :: (x : \mathit{ctr}) \\
\cdot \Vdash^{0} \mathit{emp} :: (x : \mathit{ctr}) \\[4pt]
x \xleftarrow{0} \mathit{bit0}\ y = \\
\quad \mathsf{case}\ x\ (\, \mathbf{inc} \cdot x' \Rightarrow \\
\quad\quad \mathsf{case}\ x'\ (\, \mathsf{pot}\ 1\ x'' \Rightarrow \\
\quad\quad\quad x'' \xleftarrow{1} \mathit{bit1}\ y\,)\,) \\[4pt]
x \xleftarrow{1} \mathit{bit1}\ y = \\
\quad \mathsf{case}\ x\ (\, \mathbf{inc} \cdot x' \Rightarrow \\
\quad\quad \mathsf{case}\ x'\ (\, \mathsf{pot}\ 1\ x'' \Rightarrow \\
\quad\quad\quad y' \xleftarrow{1}\ \mathsf{work}\ 1\ ;\ y.(\mathbf{inc} \cdot y')\ ; \\
\quad\quad\quad y'' \xleftarrow{1}\ y'.(\mathsf{pot}\ 1\ y'')\ ; \\
\quad\quad\quad x'' \xleftarrow{0} \mathit{bit0}\ y''\,)\,) \\[4pt]
x \xleftarrow{0} \mathit{emp} = \\
\quad \mathsf{case}\ x\ (\, \mathbf{inc} \cdot x' \Rightarrow \\
\quad\quad \mathsf{case}\ x'\ (\, \mathsf{pot}\ 1\ x'' \Rightarrow \\
\quad\quad\quad y \xleftarrow{0} \mathit{emp}\ ; \\
\quad\quad\quad x'' \xleftarrow{1} \mathit{bit1}\ y\,)\,)
\end{array}
$$

Type-checking these definitions prove our theorem. Every increment message costs the clients two units of potential (1 to send the message, and 1 to be stored as internal potential). Since every increment message corresponds one bit being flipped, and each message send costs 1 erg, overall there are less than $2n$ bits being flipped. More precisely, if the k bits 1 in the representation, then $2n - k$ bit will have been flipped (because each of the bits 1 will store one unit of potential that has not been used yet to flip a bit). Except for zero, there will always be at least one bit 1 in the counter, so the total number of bit flips is in fact bounded by $2n - 1$ for $n > 0$.

Exercises

Exercise 1 Formulate the properties of *progress* and *preservation* for the ergometric type system, as mentioned at the end of Section 4. Sketch the key steps in these proofs as they pertain to potential and work.

Exercise 2 Revisit the example of queues, as in Exercise L22.3 and Section L24.7.

(i) Give an ergometric definition of stacks with push and pop operations.

(ii) Give an ergometric definition of stack reversal.

(iii) Give an ergometric definition of queues with enqueue and dequeue operations, implemented as a bucket brigade.

(iv) Give an ergometric definitions of queues using two stacks. If this necessitates additional ergometric types, or new types for processes such as stack reversal, please state these revised definitions explicitly.

References

[Bre74] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.

[DHP18a] Ankush Das, Jan Hoffmann, and Frank Pfenning. Parallel complexity analysis with temporal session types. In M. Flatt, editor, *Proceedings of International Conference on Functional Programming (ICFP'18)*, pages 91:1–91:30, St. Louis, Missouri, USA, September 2018. ACM.

[DHP18b] Ankush Das, Jan Hoffmann, and Frank Pfenning. Work analysis with resource-aware session types. In Anuj Dawar and Erich Grädel, editors, *Proceedings of 33rd Symposium on Logic in Computer Science (LICS'18)*, pages 305–314, Oxford, UK, July 2018.

[DP20] Ankush Das and Frank Pfenning. Rast: Resource-aware session types with arithmetic refinements. In Z. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, pages 4:1–4:17. LIPIcs 167, June 2020. System description.

[HAH12] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ML. In P. Madhusudan and S. Seshia, editors, *24th International Conference on Computer Aided Verification (CAV 2012)*, pages 781–786, Berkeley, California, July 2012. Springer LNCS 7358.