Typed λ-calculus: Concepts and Syntax

P. B. Levy

University of Birmingham

1 Introduction

λ-calculus is a small language based on some common mathematical idioms. It was invented by Alonzo Church in 1936, but his version was *untyped*, making the connection with mathematics rather problematic. In this course we'll be looking at a *typed* version.

λ-calculus has had an impact throughout computer science and logic. For example

- it is the basis of functional programming languages such as Haskell, Standard ML, OCaml, Lisp, Scheme, Erlang, Scala, F♯.
- it is often used to give semantics for programming languages. This was initiated by Peter Landin, who in 1965 described the semantics of Algol-60 by translating it into λ-calculus.
- it closely corresponds to a kind of logic called *intuitionistic* logic, via the *Curry-Howard isomorphism.* That isn't in this course, but you may notice that a lot of notation (e.g. $\vdash$) and terminology ("introduction/elimination rule") has been imported from logic into λ-calculus. And the influence in the opposite direction has been much greater.

2 Notations for Sets and Elements

or **Sums your primary school never taught you**

In this section, we're going to learn some notations and abbreviations for describing *sets* and *elements of sets.*

Recall that $x \in A$ means "x is an element of the set A".

2.1 Sets

First, the notations for describing sets.

integers We define $\mathbb{Z}$ to be the set of integers.
booleans We define $\mathbb{B}$ to be the set of booleans $\{\mathsf{true}, \mathsf{false}\}$.
cartesian product Suppose A and B are sets. Then we write $A \times B$ for the set of ordered pairs

$$\{\langle x, y\rangle | x \in A, y \in B\}$$

disjoint union Suppose A and B are sets. Then we write $A + B$ for the set of ordered pairs

$$\{\mathsf{inl}\ x | x \in A\} \cup \{\mathsf{inr}\ x | x \in B\}$$

Here we use inl and inr as "tags". If you like, you could define

$$\begin{aligned} \mathsf{inl}\ x &\stackrel{\text{def}}{=} \langle 0, x\rangle \\ \mathsf{inr}\ x &\stackrel{\text{def}}{=} \langle 1, x\rangle \end{aligned}$$

function space Suppose A and B are sets. Then we write $A \to B$ for the set of functions from A to B. (You will also see this written as B^A.)

These operations on sets correspond to familiar operations on natural numbers. If A is finite with m elements, and B is finite with n elements, then

- $A \times B$ has mn elements
- $A + B$ has $m + n$ elements
- $A \to B$ has n^m elements.

2.2 Integers and Booleans

Recall that $\mathbb{Z}$ is the set of integers, and $\mathbb{B}$ is the set of booleans.
Some ways of describing integers.

Arithmetic Here is an integer:

$$3 + (7 \times 2)$$

Conditionals Here is another integer:

$$\mathsf{case}\ (7 > 5)\ \mathsf{of}\ \{\mathsf{true}.\ 20 + 3, \mathsf{false}.\ 53\}$$

This is an "if... then ... else" construction.

Local definitions Here is another integer:

$$\mathsf{let}\ (2 \times 18) + (3 \times 102)\ \mathsf{be}\ y.\ (y + 17 \times y)$$

This is a shorthand for

$y + 17 \times y$, where we define y to be $(2 \times 18) + (3 \times 102)$

It's rather like a constant declaration in programming.

Exercise 1. What integer is

1. $(2 + 5) \times 8$
2. $\mathsf{case}\ (\mathsf{case}\ 1 > 8\ \mathsf{of}\ \{\mathsf{true}.\ 5 > 2 + 4, \mathsf{false}.\ 3 > 2\})\ \mathsf{of}\ \{\mathsf{true}.\ 3 \times 7, \mathsf{false}.\ 100\}$
3. $\mathsf{let}\ (\mathsf{let}\ 3 + 2\ \mathsf{be}\ x.\ x \times (x + 3))\ \mathsf{be}\ y.\ y + 15$
4. $\mathsf{let}\ (5 + 7)\ \mathsf{be}\ x.\ \mathsf{case}\ x > 3\ \mathsf{of}\ \{\mathsf{true}.\ 12, \mathsf{false}.\ 3 + 3\}$

?

2.3 Cartesian Product

Recall that $A \times B$ is the set of ordered pairs $\langle x, y\rangle$ such that $x \in A$ and $y \in B$.

projections If x is an ordered pair, we write πx for its first component, and $\pi' x$ for its second component. For example, here is another integer

$$\mathsf{let}\ \langle 3, 7 + 2\rangle\ \mathsf{be}\ x.\ (\pi x) \times (\pi' x) + (\pi' x)$$

pattern-match We can also pattern-match an ordered pair. For example:

$$\mathsf{let}\ \langle 3, 7 + 2\rangle\ \mathsf{be}\ x.\ \mathsf{case}\ x\ \mathsf{of}\ \langle y, z\rangle.\ y \times z + z$$

Here, you don't need to select the appropriate case, because there's only one. Since x is the pair $\langle 3, 9\rangle$, it matches the pattern $\langle y, z\rangle$, and y and z are thereby defined to be 3 and 9 respectively.

Pattern-matching is often a more convenient notation than projections.

Exercise 2. What integer is

1. $\mathsf{let}\ \langle 7, \mathsf{let}\ 3\ \mathsf{be}\ x.\ x + 7\rangle\ \mathsf{be}\ y.\ \pi y + (\mathsf{case}\ y\ \mathsf{of}\ \langle u, v\rangle.\ u + v)$
2. $\mathsf{case}\ (\pi\langle 7, 357 \times 128\rangle > 2)\ \mathsf{of}\ \{\mathsf{true}.\ 13, \mathsf{false}.\ 2\}$
3. $\mathsf{let}\ \langle 5, \langle 2, \mathsf{true}\rangle\rangle\ \mathsf{be}\ x.\ \pi x + \pi(\mathsf{case}\ x\ \mathsf{of}\ \langle y, z\rangle.\ z)$

?

2.4 Disjoint Union

Recall that $A + B$ is the set of all ordered pairs $\mathsf{inl}\ x$, where $x \in A$, and all ordered pairs $\mathsf{inr}\ x$ where $x \in B$.

We can pattern-match an element of $A + B$. For example, here is an integer:

$$\begin{array}{l}\mathsf{let}\ \mathsf{inl}\ 3\ \mathsf{be}\ x.\ \mathsf{let}\ 7\ \mathsf{be}\ y.\\ \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{inl}\ z.\ z + y, \mathsf{inr}\ z.\ z \times y\}\end{array}$$

Since x is defined here to be $\mathsf{inl}\ 3$, it matches the pattern $\mathsf{inl}\ z$, and z is thereby defined to be 3.

Exercise 3. What integer is

1. $\begin{array}{l}\mathsf{case}\ (\mathsf{case}\ (3 < 7)\ \mathsf{of}\ \{\mathsf{true}.\ \mathsf{inr}\ (8 + 1), \mathsf{false}.\ \mathsf{inl}\ 2\})\\ \ \mathsf{of}\ \{\mathsf{inl}\ u.\ u + 8, \mathsf{inr}\ u.\ u + 3\}\end{array}$
2. $\begin{array}{l}\mathsf{let}\ \langle 3, \mathsf{inr}\ \langle 7, \mathsf{true}\rangle\rangle\ \mathsf{be}\ z.\pi z + \mathsf{case}\ \pi' z\\ \ \mathsf{of}\ \{\mathsf{inl}\ y.\ y + 2, \mathsf{inr}\ y.\ \mathsf{let}\ 4\ \mathsf{be}\ x.\ ((x + \pi y) + \pi z)\}\end{array}$

?

2.5 Function Space

Recall that $A \to B$ is the set of all functions from A to B.

λ-abstraction Suppose A is a set. We write $\lambda x_A.$ to mean "the function that takes each $x \in A$ to ". For example, $\lambda x_{\mathbb{Z}}.(2 \times x + 1)$ is the function taking each integer x to $2 \times x + 1$.

application If f is a function from A to B, and $x \in A$, then we write fx to mean f applied to x. For example, here is another integer:

$$(\lambda x_{\mathbb{Z}}.\ (2 \times x + 1))7$$

And that completes our notation.

Exercise 4. What integer is

1. $((\lambda f_{\mathbb{Z}\to\mathbb{Z}}.\ \lambda x_{\mathbb{Z}}.\ (f\ (f\ x)))\ \lambda x_{\mathbb{Z}}.\ (x + 3))\ 2$
2. $\mathsf{let}\ \lambda x_{\mathbb{Z}+\mathbb{B}}.\ \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{inl}\ y.\ y + 3, \mathsf{inr}\ y.7\}\ \mathsf{be}\ f.\ (f\ \mathsf{inl}\ 5) + (f\ \mathsf{inr}\ \mathsf{false})$
3. $\mathsf{let}\ \lambda x_{\mathbb{Z}\times\mathbb{Z}}.\ (\mathsf{case}\ x\ \mathsf{of}\ \langle y, z\rangle.\ (2 \times y + z))\ \mathsf{be}\ f.\ f\ \langle \mathsf{let}\ 4\ \mathsf{be}\ u.\ u + 1, 8\rangle$

?

3 A Calculus For Integers and Booleans

3.1 Calculus of Integers

We want to turn all of the above notations into a calculus. Typically, calculi are defined inductively. As an example, here is a little calculus of integer expressions:

- $\underline{n}$ is an integer expression for every $n \in \mathbb{Z}$.
- If M is an integer expression, and N is an integer expression, then $M + N$ is an integer expression.
- If M is an integer expression, and N is an integer expression, then $M \times N$ is an integer expression.

Thus an integer expression is a finite string of symbols. Don't get confused between the integer *expression* $\underline{3}+\underline{4}$, and the *integer* $3+4$, which is 7. (I normally won't bother with the underlining, but in principle it's necessary.)

Actually, I lied: an integer expression isn't really a finite string of symbols, it's a finite *tree* of symbols. So $(\underline{3}+\underline{4}) \times \underline{2}$ and $\underline{3}+\underline{4} \times \underline{2}$ represent different expressions. But $\underline{3}+\underline{4} \times \underline{2}$ and $\underline{3}+((\underline{4} \times \underline{2}))$ are the same expression i.e. the same tree.

Remark 1. Since this isn't a course on induction, I'm not delving into this in any more detail. But here is something for your notes, anticipating what you'll learn in the categories course.

The above inductive definition can be understood as describing a *category*. An object of this category is an *algebra* consisting of a set X, equipped with an element $\underline{n} \in X$, for each $n \in \mathbb{Z}$, and two binary operations $+$ and $\times$. A morphism is an *algebra homomorphism* i.e. a function between sets that preserves all this structure. Then the set of integer expressions (trees of symbols) is an *initial algebra*, i.e. an initial object in this category of algebras.

Let us write $\vdash M : \texttt{int}$ to mean "M is an integer expression". Then the above inductive definition can be abbreviated as follows.

$$\frac{}{\vdash \underline{n} : \texttt{int}}\ n \in \mathbb{Z}$$

$$\frac{\vdash M : \texttt{int} \quad \vdash N : \texttt{int}}{\vdash M + N : \texttt{int}} \qquad \frac{\vdash M : \texttt{int} \quad \vdash N : \texttt{int}}{\vdash M \times N : \texttt{int}}$$

The two expressions shown above can be written as "proof trees", this time with the root at the bottom (like in botany).

$$\dfrac{\dfrac{\dfrac{}{\vdash 3 : \mathtt{int}} \quad \dfrac{}{\vdash 4 : \mathtt{int}}}{\vdash 3+4 : \mathtt{int}} \qquad \dfrac{}{\vdash 2 : \mathtt{int}}}{\vdash (3+4)\times 2 : \mathtt{int}}$$

and

$$\dfrac{\dfrac{}{\vdash 3 : \mathtt{int}} \qquad \dfrac{\dfrac{}{\vdash 4 : \mathtt{int}} \quad \dfrac{}{\vdash 2 : \mathtt{int}}}{\vdash 4\times 2 : \mathtt{int}}}{\vdash 3+4\times 2 : \mathtt{int}}$$

3.2 Calculus of Integers and Booleans

Next we want to make a calculus of integers and booleans. We define the set of types (i.e. set expressions) to be $\{\mathtt{int}, \mathtt{bool}\}$. We write $\vdash M : A$ to mean that M is an expression of type A. To the above rules we add:

$$\dfrac{}{\vdash \mathtt{true} : \mathtt{bool}} \qquad \dfrac{}{\vdash \mathtt{false} : \mathtt{bool}}$$

$$\dfrac{\vdash M : \mathtt{int} \quad \vdash N : \mathtt{int}}{\vdash M > N : \mathtt{bool}} \qquad \dfrac{\vdash M : \mathtt{bool} \quad \vdash N : B \quad \vdash N' : B}{\vdash \mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{true}.\ N, \mathtt{false}.N'\} : B}$$

3.3 Local Definitions

We next want to add local definitions to our calculus, but this presents a problem. On the one hand, `let 3 be x. x` + 4 should definitely be an integer expression. If we type it into the computer, we get

```
Answer: 7
```

So we want $\vdash \mathtt{let}\ 3\ \mathtt{be}\ \mathtt{x}.\ \mathtt{x} + 4 : \mathtt{int}$.

But $\mathtt{x} + 4$ is not valid as an integer expression. If we type it into the computer, we get

```
Error: you haven't defined x.
```

So we don't want $\vdash \mathtt{x} + 4 : \mathtt{int}$.

How then can we define the calculus? We have a valid expression with a subterm that is not syntactically valid!

The solution is to write

$$\mathtt{x} : \mathtt{int} \vdash \mathtt{x} + 4 : \mathtt{int}$$

This means: "once x has been defined to be some integer, $\mathtt{x}+4$ is an integer expression".

Exercise 5. Which of the following would you expect to be correct statements?

1. $\mathtt{x} : \mathtt{int} \vdash \mathtt{x} + \mathtt{y} : \mathtt{int}$
2. $\mathtt{x} : \mathtt{int} \vdash \mathtt{let}\ 3\ \mathtt{be}\ \mathtt{y}.\ \mathtt{x} + \mathtt{y} : \mathtt{int}$
3. $\mathtt{x} : \mathtt{int}, \mathtt{y} : \mathtt{int} \vdash \mathtt{x} + \mathtt{y} : \mathtt{int}$
4. $\mathtt{x} : \mathtt{int}, \mathtt{y} : \mathtt{int} \vdash \mathtt{x} + 3 : \mathtt{int}$

Some terminology.

1. A, B and C range over types.
2. M and N and (if I'm desperate) P range over terms.
3. x, y and z are called *identifiers* (not "variables" please, the binding doesn't change over time).
4. A finite set of distinct identifiers with associated types, such as

$$\mathtt{x} : \mathtt{int}, \mathtt{y} : \mathtt{int}, \mathtt{z} : \mathtt{bool} \tag{1}$$

is called a *typing context.* Note that[1] a typing context is a *set*, so the order doesn't matter: the typing context

$$\mathtt{x} : \mathtt{int}, \mathtt{z} : \mathtt{bool}, \mathtt{y} : \mathtt{int}$$

is the same as (1).

5. Γ and Δ range over typing contexts.

[1] At least in these notes. Different papers may follow different conventions.

6. If Γ is a typing context, $\mathtt{x}$ an identifier and A a type, we write

$$\Gamma, \mathtt{x} : A$$

to mean Γ *extended* with the declaration $\mathtt{x} : A$. What if $\mathtt{x}$ already appears in Γ? Then that declaration is overwritten by the new one. For example,

$$\mathtt{x} : \mathtt{bool}, \mathtt{y} : \mathtt{int}, \mathtt{z} : \mathtt{bool}, \mathtt{x} : \mathtt{int}$$

describes the typing context (1).
7. Any term that can be proved in the *empty context*, i.e. $\vdash M : A$, is said to be *closed*.

Before I can give you the rules for `let`, I have to go back and change all the rules we've seen so far to incorporate a context. So the rule for $+$ becomes

$$\frac{\Gamma \vdash M : \mathtt{int} \quad \Gamma \vdash N : \mathtt{int}}{\Gamma \vdash M + N : \mathtt{int}}$$

and similarly for $\times$ and $>$.

The rule for 3 becomes

$$\frac{}{\Gamma \vdash 3 : \mathtt{int}}$$

and similarly for all the other integers, and `true` and `false`.

And the rule for conditionals becomes

$$\frac{\Gamma \vdash M : \mathtt{bool} \quad \Gamma \vdash N : B \quad \Gamma \vdash N' : B}{\Gamma \vdash \mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{true}.\ N, \mathtt{false}.\ N'\} : B}$$

We need a rule for identifiers, so that we can prove things like $\mathtt{x} : \mathtt{int}, \mathtt{y} : \mathtt{int} \vdash \mathtt{x} : \mathtt{int}$. Here's the rule:

$$\frac{}{\Gamma \vdash \mathtt{x} : A}\ (\mathtt{x} : A) \in \Gamma$$

And finally we want a rule for `let`. How do we prove that $\Gamma \vdash \mathtt{let}\ M\ \mathtt{be}\ \mathtt{x}.\ N : B$? Certainly we would have to prove something about M and something about N. To be more precise: we have to show that $\Gamma \vdash M : A$, and we have to show $\Gamma, \mathtt{x} : A \vdash N : B$. So the rule is

$$\frac{\Gamma \vdash M : A \quad \Gamma, \mathtt{x} : A \vdash N : B}{\Gamma \vdash \mathtt{let}\ M\ \mathtt{be}\ \mathtt{x}.\ N : B}$$

Exercise 6. Prove $\vdash \mathtt{let}\ 3\ \mathtt{be}\ \mathtt{x}.\ \mathtt{x} + 2 : \mathtt{int}$

4 Bound Identifiers

4.1 Scope and Shadowing

Let's consider the following term:

$$\mathtt{x} : \mathtt{int}, \mathtt{y} : \mathtt{int} \vdash (\mathtt{x} + \mathtt{y}) + \mathtt{let}\ 3\ \mathtt{be}\ \mathtt{y}.\ (\mathtt{x} + \mathtt{y}) : \mathtt{int}$$

There are 4 occurrences of identifiers in this term. The two occurrences of `x` are *free*. The first occurrence of `y` is free, but the second is *bound*. More specifically, it is bound to a particular place.

We can draw a *binding diagram* for any term:

– replace every binding of an identifier by a rectangle
– replace each bound occurrence by a circle, and draw an arrow from the circle to the rectangle where it is bound
– leave the free occurrences

How do we draw this? Every binding has a *scope* which is the term that it is applied to. Any occurrence of `x` that is outside the scope of an `x`-binder is a free occurrence. If it is inside the scope of an `x`-binding, it is bound to that `x`-binding. Sometimes, an `x`-binder sits inside the scope of another `x`-binder:

$$\mathtt{let}\ 3\ \mathtt{be}\ \mathtt{x}.\ \mathtt{let}\ 4\ \mathtt{be}\ \mathtt{x}.\ (\mathtt{x} + 2)$$

This is called *shadowing*, and the scope of the inner binder is subtracted from the scope from the outer binder. So the occurrence of `x` at the end is bound to the second binder. The rule is always

> Given an occurrence of `x`, move up the branch of the tree, and as soon as you hit an `x`-binder, that's the place the occurrence is bound to. If you never hit an `x`-binder, the occurrence is free.

Exercise 7. Draw a binding diagram for

$$\mathtt{let}\ 3\ \mathtt{be}\ \mathtt{x}.\ \mathtt{let}\ (\mathtt{let}\ \mathtt{x} + 2\ \mathtt{be}\ \mathtt{y}.\ \mathtt{y} + 7)\ \mathtt{be}\ \mathtt{y}.\ \mathtt{x} + \mathtt{y}$$

4.2 α-equivalence

Now here is a variation on the above term:

$$\mathtt{x} : \mathtt{int}, \mathtt{y} : \mathtt{int} \vdash (\mathtt{x} + \mathtt{y}) + \mathtt{let}\ 3\ \mathtt{be}\ \mathtt{z}.\ (\mathtt{x} + \mathtt{z}) : \mathtt{int}$$

The only difference is that we've changed a bound identifier. So the binding diagrams are the same. We say that two terms are *α-equivalent* when the binding diagrams are the same.

α-equivalent terms are, to all intents and purposes, the same. In fact, it would be more accurate to define a term to be a binding diagram. We take this as the definition. Bound identifiers are just a convenient way of writing a term (rather like brackets are), but the term itself is a binding diagram.

Remark 2. An elegant approach to syntax with binders is to describe the set of binding diagrams as an initial algebra. This may be done in several ways, e.g. using a presheaf category as in Fiore, Plotkin and Turi's paper "Abstract syntax and variable binding" in LICS 1999. Since this is beyond the scope of the course, we will make do with the informal description of binding diagrams above.

5 The λ-calculus

5.1 Types

Now that we've learnt the general concepts of a calculus with binding, we're ready to make a calculus out of all the notations that we saw. The *types* of this calculus are given by the inductive definition:

$$A ::= \ \mathtt{int} \mid \mathtt{bool} \mid A \times A \mid A + A \mid A \to A \mid 0 \mid 1$$

where 0 is a type corresponding to the empty set, and 1 is a type corresponding to a singleton set (a set with one element).

Like a term, a type is just a tree of symbols. Don't confuse the *type* $\mathtt{int} \to \mathtt{int}$ with the *set* $\mathbb{Z} \to \mathbb{Z}$.

As we look at the typing rules for $A \times B$ and $A + B$ and $A \to B$, we'll see that there are two kinds.

- The *introduction rules* for a type tell us how to *form* something of that type.

– The *elimination rules* for a type tell us how to *use* something of that type.

In fact, we've already seen these for the type `bool`. The typing rules for `true` and `false` are introduction rules. The typing rule for conditionals is an elimination rule.

(The type `int` is an exception to this neat pattern. Because of problems with infinity, there isn't a simple elimination rule.)

5.2 Cartesian Product

How do we form something of type $A \times B$? We use pairing. So the introduction rule is

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N \rangle : A \times B}$$

How do we use something of type $A \times B$? As we saw before, there's actually a choice here: we can either project or pattern-match. For projections, our elimination rules are

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi M : A} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi' M : B}$$

For pattern-matching, how do we prove that $\Gamma \vdash \texttt{case}\ M\ \texttt{of}\ \langle \texttt{x}, \texttt{y} \rangle.\ N : C$? Certainly we have to show something about M and something about N. And to be more precise: we have to show that $\Gamma \vdash M : A \times B$, and that $\Gamma, \texttt{x} : A, \texttt{y} : B \vdash N : C$. So the elimination rule is

$$\frac{\Gamma \vdash M : A \times B \quad \Gamma, \texttt{x} : A, \texttt{y} : B \vdash N : C}{\Gamma \vdash \texttt{case}\ M\ \texttt{of}\ \langle \texttt{x}, \texttt{y} \rangle.\ N : C}$$

We also include a type 1, representing a singleton set—the nullary product. The introduction rule is

$$\frac{}{\Gamma \vdash \langle\rangle : 1}$$

If we are using projection syntax, there are no elimination rules. If we are using pattern-match syntax, there is one elimination rule:

$$\frac{\Gamma \vdash M : 1 \quad \Gamma \vdash N : C}{\Gamma \vdash \texttt{case}\ M\ \texttt{of}\ \langle\rangle.\ N : C}$$

5.3 Disjoint Union

The rules for disjoint union are fairly similar to those for `bool`. You might like to think about why this should be so.

How do we form something of type $A + B$? By pairing with a tag. So we have two introduction rules:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \texttt{inl}\ M : A + B} \qquad \frac{\Gamma \vdash M : B}{\Gamma \vdash \texttt{inr}\ M : A + B}$$

How do we use something of type $A + B$? By pattern-matching it. To prove that $\Gamma \vdash \texttt{case}\ M\ \texttt{of}\ \{\texttt{inl x}.\ N, \texttt{inr x}.\ N'\} : C$, we have to prove something about M, something about N and something about N'. To be more precise, we have to prove that $\Gamma \vdash M : A + B$, that $\Gamma, \texttt{x} : A \vdash N : C$ and that $\Gamma, \texttt{x} : B \vdash N' : C$. So here's the elimination rule:

$$\frac{\Gamma \vdash M : A + B \quad \Gamma, \texttt{x} : A \vdash N : C \quad \Gamma, \texttt{x} : B \vdash N' : C}{\Gamma \vdash \texttt{case}\ M\ \texttt{of}\ \{\texttt{inl x}.\ N, \texttt{inr x}.\ N'\} : C}$$

We also include a type 0 representing the empty set—the nullary disjoint union. It has no introduction rule and the following elimination rule:

$$\frac{\Gamma \vdash M : 0}{\Gamma \vdash \texttt{case}\ M\ \texttt{of}\ \{\} : A}$$

5.4 Function Space

We're almost done now—we just need the rules for $A \to B$. How do we form something of type $A \to B$? We use λ-abstraction. To show that $\Gamma \vdash M : A \to B$, we need to show that $\Gamma, \texttt{x} : A \vdash M : B$. So the introduction rule is

$$\frac{\Gamma, \texttt{x} : A \vdash M : B}{\Gamma \vdash \lambda \texttt{x}_A.\, M : A \to B}$$

How do we use something of type $A \to B$? By applying it to something of type A. And that gives us something of type B. So the elimination rule is

$$\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B}$$

6 Substitution

The most important operation on terms (i.e. operation on binding diagrams) is *substitution.* If M and N are terms, we write $M[N/\mathtt{x}]$ for the term in which we substitute N for $\mathtt{x}$ in M. For example, if M is $(\mathtt{x}+\mathtt{y})\times 3$ and N is $(\mathtt{y}\times 2)$ then $M[N/\mathtt{x}]$ is $((\mathtt{y}\times 2)+\mathtt{y})\times 3$. It is most important to remember here that terms are binding diagrams:

1. Suppose M is $\mathtt{x}+\mathtt{let}\ 3\ \mathtt{be}\ \mathtt{x}.\ \mathtt{x}\times 7$, and N is $\mathtt{y}\times 2$, Writing these as binding diagrams ensures that we substitute for only the *free* occurrences. We therefore obtain $(\mathtt{y}\times 2)+\mathtt{let}\ 3\ \mathtt{be}\ \mathtt{x}.\ \mathtt{x}\times 7$.
2. Suppose M is $\mathtt{let}\ 3\ \mathtt{be}\ \mathtt{y}.\ \mathtt{x}+\mathtt{y}$, and N is $\mathtt{y}\times 2$. Writing these as binding diagrams ensures that the free occurrence of $\mathtt{y}$ in N remains free. So we obtain $\mathtt{let}\ 3\ \mathtt{be}\ \mathtt{z}.\ (\mathtt{y}\times 2)+\mathtt{z}$. If we try to substitute naively, we get $\mathtt{let}\ 3\ \mathtt{be}\ \mathtt{y}.\ (\mathtt{y}\times 2)+\mathtt{y}$. That's the wrong answer, because the free occurrence of $\mathtt{y}$ in N has been *captured.* "Substitution" always means *capture-free* substitution.

Exercise 8. Substitute

$$\mathtt{let}\ \mathtt{x}+1\ \mathtt{be}\ \mathtt{x}.\ \mathtt{x}+\mathtt{y}$$

for $\mathtt{x}$ in

$$\mathtt{x}+(\mathtt{let}\ \mathtt{x}+2\ \mathtt{be}\ \mathtt{y}.\ \mathtt{let}\ \mathtt{x}+\mathtt{y}\ \mathtt{be}\ \mathtt{x}.\ \mathtt{x}+\mathtt{y})$$

7 Exercises

1. Turn some of the descriptions of integers from the notes into expressions. Write out binding diagrams and proof trees for these examples (hint: use a large piece of paper in landscape orientation).
2. What integer is

$$\begin{array}{l}\mathsf{let}\ 3\ \mathsf{be}\ x.\\ \mathsf{let}\ \mathsf{inl}\ \lambda y_{\mathbb{Z}}.\,(x+y)\ \mathsf{be}\ u.\\ \mathsf{let}\ 4\ \mathsf{be}\ x.\\ x+(\mathsf{case}\ u\ \mathsf{of}\ \{\mathsf{inl}\ f.\ f\,2,\mathsf{inr}\ f.\ 0\})\end{array}$$

?

3. What integer is

$$\begin{array}{l}\mathsf{let}\ \lambda x_{\mathbb{Z}}.\ \mathsf{inl}\ \lambda y_{\mathbb{Z}}.\ (x+y)\ \mathsf{be}\ f.\\ \mathsf{let}\ f\,0\ \mathsf{be}\ u.\\ \mathsf{case}\ u\ \mathsf{of}\ \{\\ \quad \mathsf{inl}\ g.\ \mathsf{let}\ f\,1\ \mathsf{be}\ v.\ \mathsf{case}\ v\ \mathsf{of}\ \{\mathsf{inl}\ h.\ g\,3, \mathsf{inr}\ h.\ 0\},\\ \quad \mathsf{inr}\ g.\ 0\\ \}\end{array}$$

?

4. (variant record type) For sets A, B, C, D, E, we define $\alpha(A, B, C, D, E)$ to be the set of tuples

$$\{\langle \#\mathsf{left}, x, y\rangle | x \in A, y \in B\} \cup \{\langle \#\mathsf{right}, x, y, z\rangle | x \in C, y \in D, z \in E\}$$

Now think of α as an operation on types. Give typing rules for
   - $\langle \#\mathsf{left}, M, N\rangle$
   - $\langle \#\mathsf{right}, M, N, P\rangle$
   - $\mathtt{case}\ M\ \mathtt{of}\ \{\langle \#\mathsf{left}, \mathtt{x}, \mathtt{y}\rangle.\ N,\ \langle \#\mathsf{right}, \mathtt{x}, \mathtt{y}, \mathtt{z}\rangle.\ N'\}$

   i.e. two introduction rules and one elimination rule for α.

5. (variant function type) For sets A, B, C, D, E, F, G, we define $\beta(A, B, C, D, E, F, G)$ to be the set of functions that take
   - a sequence of arguments $(\#\mathsf{left}, x, y)$, where $x \in A$ and $y \in B$, to an element of C
   - a sequence of arguments $(\#\mathsf{right}, x, y, z)$, where $x \in D$ and $y \in E$ and $z \in F$, to an element of G.

   Thus the first argument is always a tag, indicating how many other arguments there are, what their type is, and what the type of the result should be.
   Now think of β as an operation on types. Give typing rules for
   - $M(\#\mathsf{left}, N, N')$
   - $M(\#\mathsf{right}, N, N', N'')$
   - $\lambda\{(\#\mathsf{left}, \mathtt{x}, \mathtt{y}).M, (\#\mathsf{right}, \mathtt{x}, \mathtt{y}, \mathtt{z}).M'\}$

   i.e. two elimination rules and one introduction rule for β.

Typed λ-calculus: Substitution and Equations

P. B. Levy

University of Birmingham

1 Substitution again

1.1 Substitutions and Renamings

Suppose we have a term $\Gamma \vdash M : B$, and we want to turn it into a term in context Δ, by replacing the identifiers. For example, we're given the term

$$\mathtt{x} : \mathtt{int}, \mathtt{y} : \mathtt{bool}, \mathtt{z} : \mathtt{int} \vdash \mathtt{z} + \mathtt{case}\ \mathtt{y}\ \mathtt{of}\ \{\mathtt{true}.\ \mathtt{x}+\mathtt{z}, \mathtt{false}.\ \mathtt{x}+1\} : \mathtt{int}$$

and we want to change it to something in the context $\mathtt{u} : \mathtt{bool}, \mathtt{x} : \mathtt{int}, \mathtt{y} : \mathtt{bool}$.

A *substitution* from Γ to Δ is a function k taking each identifier $\mathtt{x} : A$ in Γ to a term $\Delta \vdash k(\mathtt{x}) : A$.

For example, using the above Γ and Δ, a substitution from Γ to Δ is

$$\begin{aligned} &\mathtt{x} \mapsto 3 + \mathtt{x} \\ &\mathtt{y} \mapsto \mathtt{u} \\ &\mathtt{z} \mapsto \mathtt{case}\ \mathtt{y}\ \mathtt{of}\ \{\mathtt{true}.\ \mathtt{x} + 2, \mathtt{false}.\ \mathtt{x}\} \end{aligned}$$

We write k^*M for the result of replacing all the free identifiers in M according to k (avoiding capture, of course). In the above example, we obtain

$$\begin{aligned} &\mathtt{u} : \mathtt{bool}, \mathtt{x} : \mathtt{int}, \mathtt{y} : \mathtt{bool} \vdash \\ &\mathtt{case}\ \mathtt{y}\ \mathtt{of}\ \{\mathtt{true}.\ \mathtt{x} + 2, \mathtt{false}.\ \mathtt{x}\} + \\ &\mathtt{case}\ \mathtt{u}\ \mathtt{of}\ \{\mathtt{true}.\ (3 + \mathtt{x}) + \mathtt{case}\ \mathtt{y}\ \mathtt{of}\ \{\mathtt{true}.\ \mathtt{x} + 2, \mathtt{false}.\ \mathtt{x}\}, \\ &\mathtt{false}.\ (3 + \mathtt{x}) + 1\} : \mathtt{int} \end{aligned}$$

Exercise 1. Apply to the term

$$\mathtt{x} : \mathtt{int} \to \mathtt{int}, \mathtt{y} : \mathtt{int} \vdash \mathtt{let}\ 5\ \mathtt{be}\ \mathtt{w}.\ (\mathtt{xy}) + (\mathtt{xw}) : \mathtt{int}$$

the substitution

$$\begin{aligned} \mathtt{x} &\mapsto \mathtt{y} \\ \mathtt{y} &\mapsto \mathtt{w} + 1 \end{aligned}$$

to obtain a term in context

$$\mathtt{w} : \mathtt{int}, \mathtt{y} : \mathtt{int} \to \mathtt{int}, \mathtt{z} : \mathtt{int}$$

An important special kind of substitution is one that maps each identifier to an identifier; this is called a *renaming.* An even more special case is the inclusion from Γ to Γ', where $\Gamma \subseteq \Gamma'$. This is called *weakening.* You will often see it expressed as a proposition.

Proposition 1. *If* $\Gamma \subseteq \Gamma'$ *and* $\Gamma \vdash M : A$ *then* $\Gamma' \vdash M : A$*.*

This is proved by induction, using the fact that if $\Gamma \subseteq \Gamma'$ then $\Gamma, \mathtt{x} : B \subseteq \Gamma', \mathtt{x} : B$.

1.2 Substitution by Induction

Let us think how to define substitution on terms (rather than on binding diagrams) by induction. Some of the inductive clauses are easy:

$$\begin{aligned} k^*3 &= 3 \\ k^*(M + N) &= k^*M + k^*N \\ k^*\mathtt{x} &= k(\mathtt{x}) \end{aligned}$$

But what about substituting into a `let` expression? Let's first remember the typing rule for `let` :

$$\frac{\Gamma \vdash M : A \quad \Gamma, \mathtt{x} : A \vdash N : B}{\Gamma \vdash \mathtt{let}\ M\ \mathtt{be}\ \mathtt{x}.\ N : B}$$

We define

$$k^*(\mathtt{let}\ M\ \mathtt{be}\ \mathtt{x}.\ N) = \mathtt{let}\ k^*M\ \mathtt{be}\ \mathtt{w}.\ (k, \mathtt{x} \mapsto \mathtt{w})^*N$$

where $\mathtt{w}$ is some identifier that doesn't appear in Δ, and the substitution $\Gamma, \mathtt{x} : A \xrightarrow{k, \mathtt{x} \mapsto \mathtt{w}} \Delta, \mathtt{x} : A$ is defined to map $(\mathtt{y} : B) \in \Gamma$ (provided $\mathtt{y} \neq \mathtt{x}$) to $k(\mathtt{y})$, and $\mathtt{x}$ to $\mathtt{w}$. Note the use of Proposition 1 in this definition: $\Delta, \mathtt{w} : A \vdash k(\mathtt{y}) : B$ follows from $\Delta \vdash k(\mathtt{y}) : B$ since $\mathtt{w} \notin \Delta$.

A consequence of this is that if you want to prove a theorem about substitution, you'll first have to prove it for renaming, or at least for weakening.

Next we define

- the identity subsitution on Γ to send each $(\mathtt{x} : A) \in \Gamma$ to $\mathtt{x}$
- the composite of substitutions $\Gamma \xrightarrow{k} \Gamma' \xrightarrow{l} \Gamma''$ to send $(\mathtt{x} : A) \in \Gamma$ to $l^*(k(\mathtt{x}))$.

While we have defined subsitution for terms, this involves an arbitrary choice of fresh identifier. Because of this, it is only on binding diagrams that we obtain a *canonical* operation. Furthermore, provided we work with binding diagrams (or up to α-equivalence), we have equations:

$$(k; l)^* M = l^* k^* M$$
$$\mathsf{id}_\Gamma^* M = M$$

It follows that contexts and substitutions form a category, i.e. composition satisfies the associativity, left unital and right unital laws.

2 Evaluation Through β-reduction

Intuitively, a β-reduction means simplification. I'll write $M \rightsquigarrow N$ to mean that M can be simplified to N. We begin with some arithmetic simplifications, sometimes called δ*-reductions*:

$$\underline{m} + \underline{n} \rightsquigarrow \underline{m + n}$$
$$\underline{m} \times \underline{n} \rightsquigarrow \underline{m \times n}$$
$$\underline{m} > \underline{n} \rightsquigarrow \mathtt{true} \text{ if } m > n$$
$$\underline{m} > \underline{n} \rightsquigarrow \mathtt{false} \text{ if } m \leqslant n$$

There is a β-reduction rule for local definitions:

$$\texttt{let } M \texttt{ be x. } N \leadsto N[M/\texttt{x}]$$

But the most interesting are the β-reductions for all the types. The rough idea is: if you use an introduction rule and then, immediately, use an elimination rule, then they can be simplified.

For the boolean type, the β-reduction rule is

$$\begin{array}{r}\texttt{case true of } \{\texttt{true}.N, \texttt{false}.N'\} \leadsto N \\ \texttt{case false of } \{\texttt{true}.N, \texttt{false}.N'\} \leadsto N'\end{array}$$

For the type $A \times B$, if we use projections the β-reduction rule is

$$\begin{array}{r}\pi\langle M, M'\rangle \leadsto M \\ \pi'\langle M, M'\rangle \leadsto M'\end{array}$$

If we use pattern-matching, the β-reduction rule is

$$\texttt{case } \langle M, M'\rangle \texttt{ of } \langle \texttt{x}, \texttt{y}\rangle.\ N \leadsto N[M/\texttt{x}, M'/\texttt{y}]$$

For the type $A + B$, the β-reduction rule is

$$\begin{array}{l}\texttt{case inl } M \texttt{ of } \{\texttt{inl x}.\ N, \texttt{inr y}.\ N'\} \leadsto N[M/\texttt{x}] \\ \texttt{case inr } M \texttt{ of } \{\texttt{inl x}.\ N, \texttt{inr y}.\ N'\} \leadsto N'[M/\texttt{y}]\end{array}$$

For the type $A \to B$, the β-reduction rule is

$$(\lambda\texttt{x}.M)N \leadsto M[N/\texttt{x}]$$

A term which is the left-hand-side of a β-reduction is called a *β-redex.*

You can simplify any term M by picking a subterm that's a β-redex, and reduce it. Do this again and again until you get a β-*normal* term, i.e. one that doesn't contain any β-redex. It can be shown that this process has to terminate (the *strong normalization theorem*).

Proposition 2. *A closed term M that is β-normal must have an introduction rule at the root. (Remember that we consider $\underline{n}$ to be an introduction rule, but not $+ \times >$.) Hence, if M has type* `int`*, then it must be $\underline{n}$ for some $n \in \mathbb{Z}$.*

We prove the first part by induction on M.

Exercise 2. All the sums that we did can be turned into expressions and evaluated using β-reduction. Try:

1. $\texttt{let } \langle 5, \langle 2, \texttt{true}\rangle\rangle \texttt{ be x. } \pi\texttt{x} + \pi(\texttt{case x of } \langle \texttt{y}, \texttt{z}\rangle\texttt{. z})$
2. $\texttt{case } (\texttt{case } (3 < 7) \texttt{ of } \{\texttt{true. inr } 8+1, \texttt{false. inl } 2\}) \texttt{ of}$
   $\{\texttt{inl u. u} + 8, \texttt{inr u. u} + 3\}$
3. $((\lambda \texttt{f}_{\texttt{int}\to\texttt{int}}.\lambda \texttt{x}_{\texttt{int}}.(\texttt{f}(\texttt{fx})))\lambda \texttt{x}_{\texttt{int}}.(\texttt{x}+3))2$

3 η-expansion

The η-expansion laws express the idea that

– everything of type `bool` is `true` or `false`
– everything of type $A \times B$ is a pair $\langle x, y\rangle$
– everything of type $A + B$ is a pair `inl` x or a pair `inr` x
– everything of type $A \to B$ is a function.

They are given by first applying an elimination, then an introduction (the opposite of β-reduction).

Let's begin with the type `bool`. Suppose we have a term $\Gamma \vdash M : \texttt{bool}$. Then for any term $\Gamma, \texttt{z} : \texttt{bool} \vdash N : B$, we can expand $N[M/\texttt{z}]$ to

$$\texttt{case } M \texttt{ of } \{\texttt{true. } N[\texttt{true}/\texttt{z}], \texttt{false. } N[\texttt{false}/\texttt{z}]\}$$

The reason this ought to be true is that, whatever we define the identifiers in Γ to be, M will be either `true` or `false`. Either way, both sides should be the same.

What about $A \times B$? If we're using projections, then any $\Gamma \vdash M : A \times B$ can be η-expanded to $\langle \pi M, \pi' M\rangle$.

And if we're using pattern-match, for terms $\Gamma \vdash M : A \times B$ and $\Gamma, \texttt{z} : A \times B \vdash N : C$, we can expand $N[M/\texttt{z}]$ into

$$\texttt{case } M \texttt{ of } \langle \texttt{x}, \texttt{y}\rangle N[\langle \texttt{x}, \texttt{y}\rangle/\texttt{z}]$$

(I'm supposing the $\mathtt{x}$ and $\mathtt{y}$ we use here don't appear in $\Gamma, \mathtt{z} : A \times B$.)

For $A+B$, it's similar. Suppose $\Gamma \vdash M : A+B$ and $\Gamma, \mathtt{z} : A+B \vdash N : C$. Then $N[M/\mathtt{z}]$ can be expanded into

$$\mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{inl}\ \mathtt{x}.N[\mathtt{inl}\ \mathtt{x}/\mathtt{z}], \mathtt{inr}\ \mathtt{y}.N[\mathtt{inr}\ \mathtt{y}/\mathtt{z}]\}$$

(Again, I'm supposing the $\mathtt{x}$ and $\mathtt{y}$ don't appear in $\Gamma, \mathtt{z} : A + B$.)

And finally, $A \to B$. Any term $\Gamma \vdash M : A \to B$ can be expanded as $\lambda \mathtt{x}_A. (M\mathtt{x})$.

(Again, I'm supposing the $\mathtt{x}$ doesn't appear in Γ.)

Exercise 3. Take the term

$$\mathtt{f} : (\mathtt{int} + \mathtt{bool}) \to (\mathtt{int} + \mathtt{bool}) \vdash \mathtt{f} : (\mathtt{int} + \mathtt{bool}) \to (\mathtt{int} + \mathtt{bool})$$

Apply an η-expansion for $\to$, then for $+$, then for $\mathtt{bool}$.

4 Equality

λ-calculus isn't just a set of terms; it comes with an equational theory. If $\Gamma \vdash M : B$ and $\Gamma \vdash N : B$, we write $\Gamma \vdash M = N : B$ to express the intuitive idea that, no matter what we define the identifiers in Γ to be, M and N have the same "meaning" (even though they're different expressions).

First of all we need rules to say that this is an equivalence relation:

$$\frac{\Gamma \vdash M : B}{\Gamma \vdash M = M : B} \qquad \frac{\Gamma \vdash M = N : B}{\Gamma \vdash N = M : B}$$

$$\frac{\Gamma \vdash M = N : B \quad \Gamma \vdash N = P : B}{\Gamma \vdash M = P : B}$$

Secondly, we need rules to say that this is *compatible*—preserved by every construct:

$$\frac{\Gamma \vdash M = M' : A \quad \Gamma, \mathtt{x} : A \vdash N = N' : B}{\Gamma \vdash \mathtt{let}\ M\ \mathtt{be}\ \mathtt{x}.\ N = \mathtt{let}\ M'\ \mathtt{be}\ \mathtt{x}.\ N' : B}$$

and so forth. A compatible equivalence relation is often called a *congruence*.

Thirdly, each of the β-reductions that we've seen is an axiom of this theory.

$$\frac{\Gamma \vdash N : B \quad \Gamma \vdash N' : B}{\Gamma \vdash \texttt{case true of } \{\texttt{true}.\ N, \texttt{false}.\ N'\} = N : B}$$

$$\frac{\Gamma, \texttt{x} : A \vdash M : B \quad \Gamma \vdash N : A}{\Gamma \vdash (\lambda \texttt{x}_A . M)N = M[N/\texttt{x}] : B}$$

Fourthly, each of the η-expansions is an axiom of the theory, e.g.

$$\frac{\Gamma \vdash M : A \to B}{\Gamma \vdash M = \lambda \texttt{x}_A .\, (M\texttt{x}) : A \to B}$$

Proposition 3. *If* $\Gamma \vdash M = N : B$ *and* $\Gamma \xrightarrow{k} \Delta$ *is a substitution, then* $\Delta \vdash k^* M = k^* N : B$

As usual we prove this first for renaming, or at least for substitution.

5 Exercises

1. Suppose that $\Gamma \vdash M : \texttt{bool}$ and $\Gamma \vdash N_0, N_1, N_2, N_3 : C$. Show that

$$\begin{aligned}
&\Gamma \vdash \texttt{case } M \texttt{ of } \{\\
&\quad \texttt{true. case } M \texttt{ of } \{\texttt{true}.N_0, \texttt{false}.N_1\},\\
&\quad \texttt{false. case } M \texttt{ of } \{\texttt{true}.N_2, \texttt{false}.N_3\}\\
&\}\\
&= \texttt{case } M \texttt{ of } \{\texttt{true}.N_0, \texttt{false}.N_3\} : C
\end{aligned}$$

2. Show that $\texttt{inl} -$ is injective, i.e. if $\Gamma \vdash M, M' : A$ and $\Gamma \vdash \texttt{inl } M = \texttt{inl } M' : A + B$ then $\Gamma \vdash M = M' : A$.
3. Write down the η-law for the 0 type.
4. Given a term $\Gamma, \texttt{x} : A \vdash M : 0$, show that it is an "isomorphism" in the sense that there is a term $\Gamma, \texttt{y} : 0 \vdash N : A$ satisfying

$$\Gamma, \texttt{y} : 0 \vdash M[N/\texttt{x}] = \texttt{y} : 0$$
$$\Gamma, \texttt{x} : A \vdash N[M/\texttt{y} = \texttt{x} : A$$

5. Give β and η laws for $\alpha(A, B, C, D, E)$ and for $\beta(A, B, C, D, E, F, G)$. (See yesterday's exercises for a description of these types.)

Typed $\boldsymbol{\lambda}$-calculus: From Pure To Effectful

P. B. Levy

University of Birmingham

1 Denotational Semantics

Now we relate our syntax to the "real" world of sets and functions.

The first step: to each type A, we associate a set $[\![A]\!]$. This is by induction on A.

$$\begin{aligned}
[\![\mathtt{int}]\!] &= \mathbb{Z} \\
[\![\mathtt{bool}]\!] &= \mathbb{B} \\
[\![A+B]\!] &= [\![A]\!] + [\![B]\!] \\
[\![0]\!] &= 0 \\
[\![A \times B]\!] &= [\![A]\!] \times [\![B]\!] \\
[\![1]\!] &= 1 \\
[\![A \to B]\!] &= [\![A]\!] \to [\![B]\!]
\end{aligned}$$

Recall that a *context* Γ is a set of distinct identifiers with types e.g. $\mathtt{x} : A, \mathtt{y} : B$.

A *syntactic environment* for Γ provides, for each identifier $\mathtt{x} : A$ in Γ, a closed term of type A. (If you like, it's a substitution from Γ to the empty context.)

A *semantic environment* for Γ provides, for each identifier $\mathtt{x} : A$ in Γ, an element of $[\![A]\!]$.

For example

$$\mathtt{x} : \mathtt{int} \to \mathtt{int}, \mathtt{y} : \mathtt{bool}$$

is a context.

$$\begin{aligned}
&\mathtt{x} \mapsto \lambda \mathtt{x}_{\mathtt{int}}.(\mathtt{x}+1) \\
&\mathtt{y} \mapsto \mathtt{true}
\end{aligned}$$

is a syntactic environment.

$$\begin{aligned} \mathtt{x} &\mapsto \lambda x_{\mathbb{Z}}.(x+1) \\ \mathtt{y} &\mapsto \mathsf{true} \end{aligned}$$

is a semantic environment.

We define $[\![\Gamma]\!]$ to be the set of semantic environments for Γ. (This is *after* defining the semantics of types.)

Now suppose we have a term $\Gamma \vdash M : B$. The denotation of M provides, for each semantic environment $\rho \in [\![\Gamma]\!]$, an element $[\![M]\!]\rho \in [\![B]\!]$. So we can say

$$[\![\Gamma]\!] \xrightarrow{[\![M]\!]} [\![B]\!]$$

(To be completely precise we should write $[\![\Gamma \vdash M : B]\!]$ rather than $[\![M]\!]$ but I will not bother to do this.)

This denotation is defined by induction on the proof of $\Gamma \vdash M : B$. For example,

$$\begin{gathered} [\![\mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{inl}\ \mathtt{x}.N, \mathtt{inr}\ \mathtt{y}.N'\}]\!]\rho \\ = \\ \mathsf{case}\ [\![M]\!]\rho\ \mathsf{of}\ \{\mathsf{inl}\ x.[\![N]\!](\rho, \mathtt{x} \mapsto x), \mathsf{inr}\ y.[\![N']\!](\rho, \mathtt{y} \mapsto y)\} \end{gathered}$$

Next, given a substitution $\Gamma \xrightarrow{k} \Delta$, we obtain a function $[\![\Delta]\!] \xrightarrow{[\![k]\!]} [\![\Gamma]\!]$ (note the change of direction). It maps $\rho \in [\![\Delta]\!]$ to the semantic environment for Γ that takes each identifier $\mathtt{x} : A$ in Γ to $[\![k(\mathtt{x})]\!]\rho$.

We use this to formulate a *substitution lemma.* For $\rho \in [\![\Gamma]\!]$,

$$[\![k^* M]\!]\rho = [\![M]\!]([\![k]\!]\rho)$$

or as a diagram:

$$\begin{tikzcd} {[\![\Delta]\!]} \arrow[d, "{[\![k]\!]}"'] \arrow[dr, "{[\![k^* M]\!]}"] & \\ {[\![\Gamma]\!]} \arrow[r, "{[\![M]\!]}"'] & {[\![B]\!]} \end{tikzcd}$$

As always, this must be proved in two stages, first for renaming (or at least weakening) and then for general substitution.

Armed with the substitution lemma, it is easy to prove the soundness of all our equations:

Proposition 1. *If* $\Gamma \vdash M = N : A$ *then* $[\![M]\!] = [\![N]\!]$.

Now, let's write $[\Gamma \vdash B]$ to mean the set of $\beta\eta$-equivalence classes of terms $\Gamma \vdash M : B$. And let's write $[\![\Gamma \vdash B]\!]$ to mean the set of functions from $[\![\Gamma]\!]$ to $[\![B]\!]$. Our denotational semantics provides a function from $[\Gamma \vdash B]$ to $[\![\Gamma \vdash B]\!]$.

2 Reversible Rules

Each type (except `int`) has a *reversible rule* that indicates its deep structure. For example for `bool` we have

$$\frac{\Gamma \vdash B \quad \Gamma \vdash B}{\Gamma, \texttt{bool} \vdash B}$$

That means that we have a bijection from $[\Gamma \vdash B] \times [\Gamma \vdash B]$ to $[\Gamma, \texttt{x} : \texttt{bool} \vdash B]$. And, moreover, that we have a bijection from $[\![\Gamma \vdash B]\!] \times [\![\Gamma \vdash B]\!]$ to $[\![\Gamma, \texttt{x} : \texttt{bool} \vdash B]\!]$.

For the sum type, we have

$$\frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A + B \vdash C}$$

For the function type, we have

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$$

For the product type, we have two reversible rules, just as there are two versions of the elimination rules. The one that fits projections is

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \times B}$$

The one that fits pattern-matching is

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \times B \vdash C}$$

Generally, we see that product type with pattern-matching is very similar to a sum type, whereas product type with projection is similar to a function type. To see how this can be, imagine $\langle M, N\rangle$ as a function that maps $\#\mathsf{left}$ to M and $\#\mathsf{right}$ to N. Thus πM is M applied to $\#\mathsf{left}$, whereas $\pi' M$ is M applied to $\#\mathsf{right}$. In the rest of this course, the difference between projection and pattern-matching becomes significant, and so we will omit product types. However, if you're following the exercises on α and β, you should be able to see how to do both kinds of product.

3 Something Imperative

So far we have seen simply typed λ-calculus, as an equational theory. This is a purely functional language. But, sometimes, allegedly functional languages allow programmers to throw in something imperative.

1. In ML you can command the computer to print a character before evaluating a term.

$$\frac{\Gamma \vdash M : B}{\Gamma \vdash \texttt{print}\ c.\ M : B}\ c \in \mathcal{A}$$

   Here $\mathcal{A}$ is the set of characters that can be printed.
2. You can cause the computer to halt with an error message

$$\frac{}{\Gamma \vdash \texttt{error}\ e : B}\ e \in E$$

   Here E is the set of error messages.
3. In both Haskell and ML, we can write a program that *diverges* i.e. fails to terminate.

$$\frac{}{\Gamma \vdash \texttt{diverge} : B}$$

Indeed, it is an annoying fact that any language in which you can program every *total* computable function from $\mathbb{Z}$ to $\mathbb{Z}$ must also have programs that diverge.

Proposition 2. *Let $f : \mathbb{Z} \times \mathbb{Z} \rightharpoonup \mathbb{Z}$ be a computable partial function. (Think: f is an interpreter for the programming language. The first argument encodes a program of type* `int` $\to$ `int`*, and $f(m, n)$ applies the program that m encodes to n.) Suppose that, for every total computable function $g : \mathbb{Z} \longrightarrow \mathbb{Z}$, there exists m such that $\forall n \in \mathbb{Z}.\ f(m, n) = g(n)$. Then f is not total.*

It must be admitted that terms like

$$\texttt{print "hello"}.\ \lambda \texttt{x}_{\texttt{int}}.3$$

$$\lambda \texttt{x}_{\texttt{bool}}.\texttt{case x of } \{\texttt{true}.\ 3, \texttt{false}.\ \texttt{error CRASH}\}$$

seem very strange in the way that they mix functional idioms with imperative features (sometimes called *computational effects*). It's not apparent that they have any meaning whatsoever.

And the situation is even worse than this. Let's say we have two terms $\Gamma \vdash M, N : B$. Then in the $\beta\eta$ theory we have

$$\begin{aligned}
\Gamma \vdash M &= M[\texttt{error CRASH}/\texttt{z}] && \texttt{z} : 0 \text{ fresh for } \Gamma \\
&= \texttt{case (error CRASH) of } \{\} && \text{by the } \eta\text{-law for } 0 \\
&= N[\texttt{error CRASH}/\texttt{z}] && \text{by the } \eta\text{-law for } 0 \\
&= N : B
\end{aligned}$$

So our equational theory tells us that any two terms are equal. Even `true` and `false`. That theory goes straight into the bin.

4 Operational Semantics

4.1 Introduction

We can give meaning to this kind of hybrid functional/imperative language by giving a way of executing/evaluating terms. This is called an *operational semantics*.

Really, our task is to give a way of evaluating closed terms of type `int` to a value $\underline{n}$. To do this, we need to evaluate closed terms of other types. So, for every type, we need a set of terminal terms, where we stop evaluating.

For `bool`, the terminal terms are the values `true` and `false`.

For function type, we'll say that the terminal terms are λ-abstractions. It seems silly to evaluate under λ`x` when we don't know what `x` is.

Having made these decisions, several questions remain.

- To evaluate $\texttt{let}\ M\ \texttt{be x}.\ N$, do we
  1. evaluate M to a terminal term T, and then evaluate $N[T/\mathtt{x}]$
  2. or just substitute M, unevaluated, for $\mathtt{x}$?
- To evaluate MN, we certainly have to evaluate M to a λ-abstraction $\lambda\mathtt{x}.P$. But what about N? Do we
  1. evaluate N to a terminal term T (perhaps before evaluating M, perhaps after)?
  2. substitute N, unevaluated for $\mathtt{x}$?
- To evaluate $\texttt{inl}\ M$, do we
  1. evaluate M—so $\texttt{inl}\ T$ is terminal only if T is
  2. stop straight away—so $\texttt{inl}\ M$ is always terminal?

This seems to open up a huge space of different languages, all with the same syntax. However, there is really a single, fundamental question underlying all the ones above. Do we bind an identifier to

1. a terminal term
2. a wholly unevaluated term?

The first answer is known as *call-by-value* and the second answer is known as *call-by-name*. To put it another way,

- in call-by-value, a syntactic environment consists of terminal terms
- in call-by-name, a syntactic environment consists of unevaluated terms.

It's clear that this decision determines the answer to the first two questions. In fact, though it is not so obvious, it determines the answer to the third question too.

To see this, suppose we want to evaluate

$$\texttt{case}\ M\ \texttt{of}\ \{\texttt{inl x}.N, \texttt{inr y}.N'\}$$

Clearly the first stage is to evaluate M. So we evaluate M to $\texttt{inl}\ P$, and we then know we want to evaluate N with a suitable binding for $\mathtt{x}$. In call-by-value, we must evaluate P, and then bind $\mathtt{x}$ to the result, so $\texttt{inl}\ P$ is not terminal. But in call-by-name, we bind $\mathtt{x}$ to P unevaluated, so $\texttt{inl}\ P$ must be terminal.

Thus, in call-by-value, the closed terms that are terminal are given by

$$T ::= \ \underline{n} \mid \texttt{true} \mid \texttt{false} \mid \texttt{inl}\ T \mid \texttt{inr}\ T \mid \lambda\mathtt{x}.M$$

whereas in call-by-name, the closed terms that are terminal are given by

$$T ::= \underline{n} \mid \texttt{true} \mid \texttt{false} \mid \texttt{inl}\ M \mid \texttt{inr}\ M \mid \lambda\texttt{x}.M$$

i.e. anything whose root is an introduction rule.

4.2 First-Order Interpreters

Here is a little interpreter to evaluate terms in call-by-value (using left-to-right order). It is a recursive first-order program. To evaluate

- $\underline{n}$, return $\underline{n}$.
- `true`, return `true`.
- `false`, return `false`.
- $\lambda\texttt{x}.M$, return $\lambda\texttt{x}.M$.
- `inl` M, evaluate M. If it returns T, return `inl` T.
- `inr` M, evaluate M. If it returns T, return `inr` T.
- $M + N$, evaluate M. If it returns $\underline{m}$, evaluate N. If that returns $\underline{n}$, return $\underline{m+n}$.
- `let` M `be x.` N, evaluate M. If it returns T, evaluate $N[T/\texttt{x}]$.
- `case` M `of` $\{\texttt{true}.N, \texttt{false}.N'\}$, evaluate M. If it returns `true`, evaluate N, but if it returns `false`, evaluate N'.
- `case` M `of` $\{\texttt{inl x}.N, \texttt{inr x}.N'\}$, evaluate M. If it returns `inl` T, evaluate $N[T/\texttt{x}]$, but if it returns `inr` T, evaluate $N'[T/\texttt{x}]$.
- MN, evaluate M. If it returns $\lambda\texttt{x}.P$, evaluate N. If that returns T, evaluate $P[T/\texttt{x}]$.
- `print` $c.\ M$, print c and then evaluate M.
- `error` e, halt with error message e.
- `diverge`, diverge.

Note that we only ever substitute terminal terms.

Now here is an interpreter for call-by-name. To evaluate

- $\underline{n}$, return $\underline{n}$.
- `true`, return `true`.
- `false`, return `false`.
- $\lambda\texttt{x}.M$, return $\lambda\texttt{x}.M$.
- `inl` M, return `inl` M.
- `inr` M, return `inr` M.

- $M + N$, evaluate M. If it returns $\underline{m}$, evaluate N. If that returns $\underline{n}$, return $\underline{m+n}$.
- $\texttt{let}\ M\ \texttt{be}\ \texttt{x}.\ N$, evaluate $N[M/\texttt{x}]$.
- $\texttt{case}\ M\ \texttt{of}\ \{\texttt{true}.N, \texttt{false}.N'\}$, evaluate M. If it returns $\texttt{true}$, evaluate N, but if it returns $\texttt{false}$, evaluate N'.
- $\texttt{case}\ M\ \texttt{of}\ \{\texttt{inl}\ \texttt{x}.N, \texttt{inr}\ \texttt{x}.N'\}$, evaluate M. If it returns $\texttt{inl}\ P$, evaluate $N[P/\texttt{x}]$, but if it returns $\texttt{inr}\ P$, evaluate $N'[P/\texttt{x}]$.
- MN, evaluate M. If it returns $\lambda\texttt{x}.P$, evaluate $P[N/\texttt{x}]$.
- $\texttt{print}\ c.\ M$, print c and then evaluate M.
- $\texttt{error}\ e$, halt with error message e.
- $\texttt{diverge}$, diverge.

Note that we only ever substitute unevaluated terms.

Exercise 1. 1. Evaluate

$$\texttt{let error CRASH be x. } 5$$

in CBV and CBN

2. Evaluate

$$(\lambda\texttt{x}.(\texttt{x}+\texttt{x}))(\texttt{print "hello"}.\ 4)$$

in CBV and CBN.

3. Evaluate

$$\begin{array}{l}\texttt{case (print "hello". inr error CRASH) of} \\ \quad \{\texttt{inl x. x} + 1, \texttt{inr y. } 5\}\end{array}$$

in CBV and CBN.

4.3 Big-Step Semantics

We'll leave aside printing now, and just think about errors.

One way of turning the big-step semantics into a mathematical description is using an evaluation relation. We will write $M \Downarrow T$ to mean that M (a closed term) evaluates to T (a terminal term), and $M \lightning e$ to mean that M halts with error message e.

We define $\Downarrow$ and $\lightning$ inductively. Here are some of the clauses:

$$\frac{}{\lambda \mathtt{x}.M \Downarrow \lambda \mathtt{x}.M} \qquad \frac{}{\mathtt{error}\ e \lightning e}$$

$$\frac{M \Downarrow \lambda \mathtt{x}.P \quad N \Downarrow T \quad P[T/\mathtt{x}] \Downarrow T'}{MN \Downarrow T'} \qquad \frac{M \lightning e}{MN \lightning e}$$

$$\frac{M \Downarrow \lambda \mathtt{x}.P \quad N \lightning e}{MN \lightning e} \qquad \frac{M \Downarrow \lambda \mathtt{x}.P \quad N \Downarrow T \quad P[T/\mathtt{x}] \lightning e}{MN \lightning e}$$

Evaluation always terminates:

Proposition 3. *Let $\vdash M : B$ be a closed term. Then either*

- *$M \Downarrow T$ for unique terminal $T : B$, and there does not exist e such that $M \lightning e$, or*
- *$M \lightning e$ for unique error $e \in E$, and there does not exist T such that $M \Downarrow T$.*

This can be proved using a method due to Tait.

Similarly, we can inductively define $\Downarrow$ and $\lightning$ for CBN, and Prop. 3 holds for these predicates.

5 Observational Equivalence

With the pure λ-calculus, we knew what the intended meaning was, so we could easily write down equations between terms. But we do not have, at this stage, a denotational semantics for the calculus with errors or printing. So what does it mean for two terms to be "the same"?

Well, if M and N are closed terms of type `int`, it's pretty clear. They're the same when they either evaluate to the same number or raise the same error. With printing, they must also print the same string.

But what about other terms? Here's a way of answering this question. Let's say that a *program context* $\mathcal{C}[\cdot]$ is a closed term of type `int`, with a "hole". If we have two terms $\Gamma \vdash M, N : B$, and we plug them into the hole of a program context, and they behave *differently*, then we definitely need to consider M and N to be different. On the other hand, if they behave the same when plugged into *any* program context (assuming the hole itself is typed $\Gamma \vdash [\cdot] : B$), then we could regard as the same. In this situation, we say that they are *observationally equivalent*, and we write $\Gamma \vdash M \simeq N : B$. This is really the coarsest reasonable equivalence relation we could consider.

Let's look at some examples of this. I should tell you first that in both CBV and CBN there's a result called the *context lemma* that tells us that if two terms behave the same in every syntactic environment, then they behave the same in every program context.

Let's start with the equivalence

$$(\lambda \mathtt{x}.M)N \simeq M[N/\mathtt{x}]$$

This, the β-law for $\rightarrow$, holds in CBN but not in CBV. As an example, put N to be `error CRASH`, and put M to be 3.

Next, consider the equivalence

$$\mathtt{z} : \mathtt{bool} \vdash 3 \simeq \mathtt{case}\ \mathtt{z}\ \mathtt{of}\ \{\mathtt{true}.3, \mathtt{false}.3\} : \mathtt{int}$$

This is an instance of the η-law for `bool`. It holds in CBV because a syntactic environment must consist of terminal terms, so `z` must be either `true` or `false`. But it fails in CBN because we can apply the program context `let` (`error CRASH`) `be z`. $[\cdot]$.

Remark 1. This program context is different from `let` (`error CRASH`) `be y`. $[\cdot]$. So, by contrast with terms, we can't α-convert a program context.

Next, consider the equivalence

$$\vdash \lambda \mathtt{x}_{\mathtt{int}}.\mathtt{error}\ e \simeq \mathtt{error}\ e : \mathtt{int} \rightarrow \mathtt{int}$$

This seems unlikely: the LHS terminates whereas the RHS raises an error. It fails in CBV: take the program context `let` $[\cdot]$ `be y`. 3. In CBN it holds, but it is rather subtle. The reason is that there is no way of causing the hole's contents to be evaluated *except* to apply it to something. And when we apply it, it raises an error.

A very similar example is this one

$$\vdash \lambda \mathtt{x}_{\mathtt{int}}.\mathtt{print}\ c.\ M \simeq \mathtt{print}\ c.\ \lambda \mathtt{x}_{\mathtt{int}}.M : \mathtt{int} \rightarrow \mathtt{int}$$

Again, this fails in CBV but holds in CBN.

6 Exercises

1. Find a context to show that

   $$\begin{aligned}&\texttt{z} : \texttt{bool} \vdash\\ &\quad \texttt{case z of } \{\texttt{true}.\texttt{case z of } \{\texttt{true}.3, \texttt{false}.3\}, \texttt{false}.3\}\\ &\quad \simeq \texttt{case z of } \{\texttt{true}.3, \texttt{false}.3\} : \texttt{int}\end{aligned}$$

   fails in CBN with printing (no errors or divergence). Using the context lemma, explain why this equivalence is valid in CBV.
2. Give reversible rules for $\alpha(A, B, C, D, E)$ and for $\beta(A, B, C, D, E, F, G)$. (See first handout, Section 8, for a description of these types.)
3. Extend each set of terminal terms and each definitional interpreter to incorporate $\alpha(A, B, C, D, E)$ and $\beta(A, B, C, D, E, F, G)$.

Typed $\boldsymbol{\lambda}$-calculus: Denotational Semantics of Call-By-Value

P. B. Levy

University of Birmingham

Preliminary note: substitution in CBV

For the pure calculus, we gave a substitution lemma expressing $[\![M[N/\mathtt{x}]]\!]$ in terms of $[\![M]\!]$ and $[\![N]\!]$. But that will not be possible in CBV, as the following example demonstrates. We define terms $\mathtt{x} : \mathtt{bool} \vdash M, M' : \mathtt{bool}$ and $\vdash N : \mathtt{bool}$ by

$$\begin{aligned} M &\stackrel{\text{def}}{=} \mathtt{true} \\ M' &\stackrel{\text{def}}{=} \mathtt{case\ x\ of}\ \{\mathtt{true.\ true},\ \mathtt{false.\ true}\} \\ N &\stackrel{\text{def}}{=} \mathtt{error\ CRASH} \end{aligned}$$

But in any CBV semantics we will have

$$\begin{aligned} [\![M]\!] &= [\![M']\!] \qquad \text{because } M =_{\eta\,\mathtt{bool}} M' \\ [\![M[N/\mathtt{x}]]\!] &\neq [\![M'[N/\mathtt{x}]]\!] \end{aligned}$$

However, what we *will* be able to describe semantically is the substitution of a restricted class of terms, called *values.*

$$V ::= \ \mathtt{x} \mid \underline{n} \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{inl}\ V \mid \mathtt{inr}\ V \mid \lambda\mathtt{x}.M$$

A value, in any syntactic environment, is terminal. And a closed term is a value iff it is terminal. In the study of call-by-value, we define a *substitution* $\Gamma \xrightarrow{k} \Delta$ to be a function mapping each identifier $\mathtt{x} : A$ in Γ to a *value* $\Delta \vdash V : A$. If W is a value, then k^*W is a value, for any substitution k.

1 Denotational Semantics for CBV

Let us think about how to give a denotational semantics for call-by-value λ-calculus with errors. Let E be the set of errors.

1.1 First Attempt

Let's propose that for a type A, its denotation $[\![A]\!]$ will be a set that's a *universe for terms*: by this I mean that a closed term of type A will denote an element of $[\![A]\!]$. Then we should have

$$\begin{aligned}
[\![\mathtt{bool}]\!] &= \mathbb{B} + E \\
[\![\mathtt{int}]\!] &= \mathbb{Z} + E \\
[\![\mathtt{bool} \times \mathtt{int}]\!] &= (\mathbb{B} \times \mathbb{Z}) + E \\
[\![A \times B]\!] &= [\![A]\!] * [\![B]\!]
\end{aligned}$$

where $*$ is an operation on sets that would have to satisfy

$$(\mathbb{B} + E) * (\mathbb{Z} + E) = (\mathbb{B} \times \mathbb{Z}) + E$$

I can't see any such operation, so we give up on this proposal.

1.2 Second Attempt

Let's make $[\![A]\!]$ a set that's a *universe for values*, meaning that a closed value of type A will denote an element of type $[\![A]\!]$. In particular we want

$$\begin{aligned}
[\![\mathtt{bool}]\!] &= \mathbb{B} \\
[\![\mathtt{int}]\!] &= \mathbb{Z} \\
[\![A + B]\!] &= [\![A]\!] + [\![B]\!] \\
[\![A \times B]\!] &= [\![A]\!] \times [\![B]\!]
\end{aligned}$$

and we postpone the semantic equation for $\rightarrow$.

A *semantic environment* for Γ maps each identifier $\mathtt{x} : A$ in Γ to an element of $[\![A]\!]$. We write $[\![\Gamma]\!]$ for the set of semantic environments.

A closed term of type B either returns a closed value or raises an error. So it should denote an element of $[\![B]\!] + E$. More generally, a term $\Gamma \vdash M : B$ should denote, for each semantic environment $\rho \in [\![\Gamma]\!]$, an element of $[\![B]\!] + E$. Hence

$$[\![\Gamma]\!] \xrightarrow{[\![M]\!]} [\![B]\!] + E$$

Now let's think about $[\![A \to B]\!]$. A closed value of type $A \to B$ is a λ-abstraction $\lambda \mathtt{x}_A.M$. This can be applied to a closed *value* V of type A, and gives a closed term $M[V/\mathtt{x}]$ of type B. So we define

$$[\![A \to B]\!] = [\![A]\!] \to ([\![B]\!] + E)$$

We can easily write out the semantics of terms now.

1.3 Substitution Lemma

According to what we have said, a value $\Gamma \vdash V : A$ denotes a function

$$[\![\Gamma]\!] \xrightarrow{[\![V]\!]} [\![A]\!] + E$$

To formulate a substitution lemma, we *also* want V to denote a function

$$[\![\Gamma]\!] \xrightarrow{[\![V]\!]^{\mathsf{val}}} [\![A]\!]$$

and $[\![V]\!]^{\mathsf{val}}$ should be related to $[\![V]\!]$ by

$$[\![V]\!]\rho = \mathsf{inl}\ [\![V]\!]^{\mathsf{val}}\rho \tag{1}$$

or as a diagram:

$$\begin{tikzcd}
[\![\Gamma]\!] \arrow[r, "{[\![V]\!]^{\mathsf{val}}}"] \arrow[dr, "{[\![V]\!]}"'] & {[\![A]\!]} \arrow[d, "\mathsf{inl}"] \\
 & {[\![A]\!] + E}
\end{tikzcd}$$

We define $[\![V]\!]^{\mathsf{val}}$ and verify (1) by induction on V.

Given a substitution $\Gamma \xrightarrow{k} \Delta$, we obtain a function $[\![\Delta]\!] \xrightarrow{[\![k]\!]} [\![\Gamma]\!]$. It maps $\rho \in [\![\Delta]\!]$ to the semantic environment for Γ that takes each identifier $\mathtt{x} : A$ in Γ+ to $[\![k(x)]\!]^{\mathsf{val}}\rho$.

Now we can formulate two substitution lemmas: one for substitution into terms, and one for substitution into values.

Proposition 1. *Let $\Gamma \xrightarrow{k} \Delta$ be a substitution, and let ρ be a semantic environment for Δ.*

1. *For any term $\Gamma \vdash M : B$, we have $[\![k^* M]\!]\rho = [\![M]\!]([\![k]\!]\rho)$, or as a diagram:*

$$\begin{tikzcd}
{[\![\Delta]\!]} \arrow[d, "{[\![k]\!]}"'] \arrow[dr, "{[\![k^* M]\!]}"] & \\
{[\![\Gamma]\!]} \arrow[r, "{[\![M]\!]}"'] & {[\![B]\!] + E}
\end{tikzcd}$$

2. *For any value $\Gamma \vdash V : B$, we have $[\![k^* V]\!]^{\mathsf{val}}\rho = [\![V]\!]^{\mathsf{val}}([\![k]\!]\rho)$, or as a diagram:*

$$\begin{tikzcd}
{[\![\Delta]\!]} \arrow[d, "{[\![k]\!]^{\mathsf{val}}}"'] \arrow[dr, "{[\![k^* V]\!]}"] & \\
{[\![\Gamma]\!]} \arrow[r, "{[\![V]\!]^{\mathsf{val}}}"'] & {[\![B]\!] + E}
\end{tikzcd}$$

As usual we first prove this for renamings (or at least weakening).

1.4 Computational Adequacy

It is all very well to define a denotational semantics, but it's no good if it doesn't agree with the way the language was defined (the operational semantics).

Proposition 2. *Let M be a closed term.*

1. *If $M \Downarrow V$, then $[\![M]\!] = \mathsf{inl}\ [\![V]\!]^{\mathsf{val}}$.*
2. *If $M \lightning e$, then $[\![M]\!] = \mathsf{inr}\ e$.*

We prove this by induction on $\Downarrow$ and $\lightning$.

Typed λ-calculus: Further Questions

P. B. Levy

University of Birmingham

The exam for this course consists of all exercises in Handouts 2–4, if you haven't done them already, and two additional questions below.

1 Answers To "Concepts and Syntax" Exercises

Here are the answers to the exercises in Section 8 of Handout 1.

Question What integer is

$$\begin{array}{l} \mathsf{let}\ 3\ \mathsf{be}\ x. \\ \mathsf{let}\ \mathsf{inl}\ \lambda y_{\mathbb{Z}}.(x+y)\ \mathsf{be}\ u. \\ \mathsf{let}\ 4\ \mathsf{be}\ x. \\ \mathsf{match}\ u\ \mathsf{as}\ \{\mathsf{inl}\ f.f2, \mathsf{inr}\ f.0\} \end{array}$$

?

Correct answer 5

Plausible but incorrect answer 6

Question What integer is

$$\begin{array}{l} \mathsf{let}\ \lambda x_{\mathbb{Z}}.\ \mathsf{inl}\ \lambda y_{\mathbb{Z}}.(x+y)\ \mathsf{be}\ f. \\ \mathsf{let}\ f0\ \mathsf{be}\ u. \\ \mathsf{match}\ u\ \mathsf{as}\ \{ \\ \quad \mathsf{inl}\ g.\ \mathsf{let}\ f1\ \mathsf{be}\ v.\ \mathsf{match}\ v\ \mathsf{as}\ \{\mathsf{inl}\ h.\ g3, \mathsf{inr}\ h.\ 0\}, \\ \quad \mathsf{inr}\ g.\ 0 \\ \} \end{array}$$

?

Correct answer 3

Plausible but incorrect answer 4. Both these exercises illustrate the idea of *static binding*, meaning that bindings cannot be changed. The incorrect answers, 6 and 4, are not in accordance with our definition of the notation. Unfortunately, Emacs Lisp would give you these answers. That is because it uses *dynamic binding*, meaning that a binding of `x` overwrites any previous binding of `x`.

Question (variant record type) For sets A, B, C, D, E, we define $\alpha(A, B, C, D, E)$ to be the set of tuples

$$\{\langle \#\mathsf{left}, x, y\rangle | x \in A, y \in B\} \cup \{\langle \#\mathsf{right}, x, y, z\rangle | x \in C, y \in D, z \in E\}$$

Now think of α as an operation on types. Inventing a reasonable syntax, given 2 introduction rules and 1 elimination rule for $\alpha(A, B, C, D, E)$.

Answer We invent the syntax

$$\langle \#\mathsf{left}, M, N\rangle \qquad\qquad \langle \#\mathsf{right}, M, N, P\rangle$$

for terms of type $\alpha(A, B, C, D, E)$. And we invent the syntax

$$\mathtt{case}\ M\ \mathtt{of}\ \{\langle \#\mathsf{left}, \mathsf{x}, \mathsf{y}\rangle.\ N, \langle \#\mathsf{right}, \mathsf{x}, \mathsf{y}, \mathsf{z}\rangle.\ N'\}$$

for pattern-matching a term M of type $\alpha(A, B, C, D, E)$.
The introduction rules are

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \langle \#\mathsf{left}, M, N\rangle : \alpha(A, B, C, D, E)}$$

$$\frac{\Gamma \vdash M : C \quad \Gamma \vdash N : D \quad \Gamma \vdash P : E}{\Gamma \vdash \langle \#\mathsf{right}, M, N, P\rangle : \alpha(A, B, C, D, E)}$$

The elimination rule is

$$\frac{\begin{array}{l}\Gamma \vdash M : \alpha(A, B, C, D, E) \\ \qquad\qquad \Gamma, \mathsf{x} : A, \mathsf{y} : B \vdash N : F \quad \Gamma, \mathsf{x} : C, \mathsf{y} : D, \mathsf{z} : E \vdash N' : F\end{array}}{\Gamma \vdash \mathtt{case}\ M\ \mathtt{of}\ \{\langle \#\mathsf{left}, \mathsf{x}, \mathsf{y}\rangle.\ N, \langle \#\mathsf{right}, \mathsf{x}, \mathsf{y}, \mathsf{z}\rangle.\ N'\} : F}$$

Question For sets A, B, C, D, E, F, G, we define $\beta(A, B, C, D, E, F, G)$ to be the set of functions that take

- a sequence of arguments $(\#\mathsf{left}, x, y)$, where $x \in A$ and $y \in B$, to an element of C
- a sequence of arguments $(\#\mathsf{right}, x, y, z)$, where $x \in D$ and $y \in E$ and $z \in F$, to an element of G.

Thus the first argument is always a tag, indicating how many other arguments there are, what their type is, and what the type of the result should be.
Now think of β as an operation on types. Inventing a reasonable syntax, give 1 introduction rule and 2 elimination rules for $\beta(A, B, C, D, E, F, G)$.

Answer We invent the syntax

$$\lambda\{(\#\mathsf{left},\mathtt{x},\mathtt{y})\ .M,(\#\mathsf{right},\mathtt{x},\mathtt{y},\mathtt{z}).\ M'\}$$

for something of type $\beta(A,B,C,D,E,F,G)$. And we invent the syntax

$$M(\#\mathsf{left},N,N') \qquad\qquad M(\#\mathsf{right},N,N',N'')$$

for applying a term M of type $\beta(A,B,C,D,E,F,G)$.
The introduction rule is

$$\frac{\Gamma,\mathtt{x}:A,\mathtt{y}:B\vdash M:C \quad \Gamma,\mathtt{x}:D,\mathtt{y}:E,\mathtt{z}:F\vdash M':G}{\Gamma\vdash\lambda\{(\#\mathsf{left},\mathtt{x},\mathtt{y})\ .M,(\#\mathsf{right},\mathtt{x},\mathtt{y},\mathtt{z}).\ M'\}:\beta(A,B,C,D,E,F,G)}$$

The elimination rules are

$$\frac{\Gamma\vdash M:\beta(A,B,C,D,E,F,G) \quad \Gamma\vdash N:A \quad \Gamma\vdash N':B}{\Gamma\vdash M(\#\mathsf{left},N,N'):C}$$

$$\frac{\Gamma\vdash M:\beta(A,B,C,D,E,F,G) \quad \Gamma\vdash N:D \quad \Gamma\vdash N':E \quad \Gamma\vdash N'':F}{\Gamma\vdash M(\#\mathsf{right},N,N',N''):G}$$

2 Question on Pure λ-calculus

This question is about the pure (i.e. no imperative features) simply typed λ-calculus.

In this language, a *syntactic isomorphism* from A to B consists of a term $\mathtt{x}:A\vdash M:B$ and a term $\mathtt{y}:B\vdash N:A$ such that the equations

$$\begin{aligned}\mathtt{x}:A&\vdash N[M/\mathtt{x}]=\mathtt{x}:B\\ \mathtt{y}:B&\vdash M[N/\mathtt{x}]=\mathtt{y}:A\end{aligned}$$

are provable in the equational theory. (NB This definition, as it stands, is not suitable for λ-calculus with imperative features.) Construct syntactic isomorphisms

$$\begin{aligned}(A+B)+C&\cong A+(B+C)\\ (A\times B)\to C&\cong A\to(B\to C)\\ (A+B)\to C&\cong(A\to C)\times(B\to C)\end{aligned}$$

3 Question on λ-calculus with Imperative Features

CELL is a storage cell (piece of computer memory) that stores an integer.

Consider call-by-value λ-calculus, without divergence or printing, but with the facility to write to and read from CELL.

- `CELL` $:= M.\ N$, where M is an integer expression. To evaluate this, first evaluate M, the put the answer in CELL (overwriting whatever was there previously), then evaluate N.
- `read CELL as x`. N. To evaluate this, define `x` to be whatever is currently in CELL, then evaluate N.

We write $s, M \Downarrow s', T$ to mean that if M (a closed term) is evaluated at a time when CELL contains s (an integer), then it evaluates to T (a terminal term, with the same type as M) with CELL then containing s' (an integer). Give an inductive definition of the relation $\Downarrow$.