

# Typed $\lambda$ -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  $\llbracket A \rrbracket$ . This is by induction on  $A$ .

$$\begin{aligned}\llbracket \text{int} \rrbracket &= \mathbb{Z} \\ \llbracket \text{bool} \rrbracket &= \mathbb{B} \\ \llbracket A + B \rrbracket &= \llbracket A \rrbracket + \llbracket B \rrbracket \\ \llbracket 0 \rrbracket &= 0 \\ \llbracket A \times B \rrbracket &= \llbracket A \rrbracket \times \llbracket B \rrbracket \\ \llbracket 1 \rrbracket &= 1 \\ \llbracket A \rightarrow B \rrbracket &= \llbracket A \rrbracket \rightarrow \llbracket B \rrbracket\end{aligned}$$

Recall that a *context*  $\Gamma$  is a list of distinct identifiers with types e.g.  $x : A, y : B$ .

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

A *semantic environment* for  $\Gamma$  provides, for each identifier  $x : A$  in  $\Gamma$ , an element of  $\llbracket A \rrbracket$ .

For example

$$x : \text{int} \rightarrow \text{int}, y : \text{bool}$$

is a context.

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

is a syntactic environment.

$$\begin{aligned} \mathbf{x} &\mapsto \lambda x.(x + 1) \\ \mathbf{y} &\mapsto \text{true} \end{aligned}$$

is a semantic environment.

We define  $\llbracket \Gamma \rrbracket$  to be the set of semantic environments for  $\Gamma$ . (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  $\rho \in \llbracket \Gamma \rrbracket$ , an element  $\llbracket M \rrbracket \rho \in \llbracket B \rrbracket$ . So we can say

$$\llbracket \Gamma \rrbracket \xrightarrow{\llbracket M \rrbracket} \llbracket B \rrbracket$$

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

$$\begin{aligned} &\llbracket \text{match } M \text{ as } \{\text{inl } \mathbf{x}.N, \text{inr } \mathbf{y}.N'\} \rrbracket \rho \\ &= \\ &\text{match } \llbracket M \rrbracket \rho \text{ as } \{\text{inl } x.\llbracket N \rrbracket(\rho, \mathbf{x} \mapsto x), \text{inr } y.\llbracket N' \rrbracket(\rho, \mathbf{y} \mapsto y)\} \end{aligned}$$

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

We use this to formulate a *substitution lemma*. For  $\rho \in \llbracket \Gamma \rrbracket$ ,

$$\llbracket k^* M \rrbracket \rho = \llbracket M \rrbracket (\llbracket k \rrbracket \rho)$$

As always, this must be proved in two stages, first for renaming and then for general substitution.

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

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  $\llbracket \Gamma \vdash B \rrbracket$  to mean the set of functions from  $\llbracket \Gamma \rrbracket$  to  $\llbracket B \rrbracket$ . Our denotational semantics provides a function from  $[\Gamma \vdash B]$  to  $\llbracket \Gamma \vdash B \rrbracket$ .

## 2 Reversible Rules

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

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

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

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

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

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 `#left` to  $M$  and `#right` to  $N$ . Thus  $\pi M$  is  $M$  applied to `#left`, whereas  $\pi' M$  is  $M$  applied to `#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  $\alpha$  and  $\beta$ , you should be able to see how to do both kinds of product.

### 3 Something Imperative

So far we have seen simply typed  $\lambda$ -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.

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

Here  $\mathcal{A}$  is the set of characters that can be printed.

2. You can cause the computer to halt with an error message

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

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.

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

Indeed, it is an annoying consequence of computability theory 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 1.** *Let  $f : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$  be a computable partial function. (Think:  $f$  is an interpreter for the programming language. The first argument encodes a program of type `int`  $\rightarrow$  `int`, and  $f(m, n)$  applies the program that  $m$  encodes to  $n$ .) Suppose that, for every total computable function  $g : \mathbb{Z} \rightarrow \mathbb{Z}$ , 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

```
print "hello".  $\lambda x_{\mathbb{Z}}. 3$ 
```

```
 $\lambda x_{\mathbf{bool}}. \mathbf{match} \ x \ \mathbf{as} \ \{\mathbf{true}. 3, \mathbf{false}. \mathbf{error} \ \mathbf{CRASH}\}$ 
```

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

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

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  $\lambda$ -abstractions. It seems silly to evaluate under  $\lambda\mathbf{x}$  when we don't know what  $\mathbf{x}$  is.

Having made these decisions, several questions remain.

- To evaluate `let M be x. N`, do we
  1. evaluate  $M$  to a terminal term  $T$ , and then evaluate  $N[T/\mathbf{x}]$
  2. or just substitute  $M$ , unevaluated, for  $\mathbf{x}$ ?
- To evaluate  $MN$ , we certainly have to evaluate  $M$  to a  $\lambda$ -abstraction  $\lambda\mathbf{x}.P$ . 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  $\mathbf{x}$ ?

- To evaluate `inl M`, do we
  1. evaluate  $M$ —so `inl T` is terminal only if  $T$  is
  2. stop straight away—so `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

$$\text{match } M \text{ as } \{\text{inl } x.N, \text{inr } y.N'\}$$

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

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

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

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

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

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**.
- $\lambda x.M$ , return  $\lambda x.M$ .
- **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  $N[T/x]$ .
- **match**  $M$  **as**  $\{\mathbf{true}.N, \mathbf{false}.N'\}$ , evaluate  $M$ . If it returns **true**, evaluate  $N$ , but if it returns **false**, evaluate  $N'$ .
- **match**  $M$  **as**  $\{\mathbf{inl}\ x.N, \mathbf{inr}\ x.N'\}$ , evaluate  $M$ . If it returns **inl**  $T$ , evaluate  $N[T/x]$ , but if it returns **inr**  $T$ , evaluate  $N'[T/x]$ .
- $MN$ , evaluate  $M$ . If it returns  $\lambda x.P$ , evaluate  $N$ . If that returns  $T$ , evaluate  $P[T/x]$ .
- **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**.
- $\lambda x.M$ , return  $\lambda x.M$ .
- **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  $\underline{m + n}$ .
- **let**  $M$  **be**  $x$ .  $N$ , evaluate  $N[M/x]$ .
- **match**  $M$  **as**  $\{\mathbf{true}.N, \mathbf{false}.N'\}$ , evaluate  $M$ . If it returns **true**, evaluate  $N$ , but if it returns **false**, evaluate  $N'$ .
- **match**  $M$  **as**  $\{\mathbf{inl}\ x.N, \mathbf{inr}\ x.N'\}$ , evaluate  $M$ . If it returns **inl**  $P$ , evaluate  $N[P/x]$ , but if it returns **inr**  $P$ , evaluate  $N'[P/x]$ .

- $MN$ , evaluate  $M$ . If it returns  $\lambda x.P$ , evaluate  $P[N/x]$ .
- **print**  $c$ .  $M$ , print  $c$  and then evaluate  $M$ .
- **error**  $e$ , halt with error message  $e$ .
- **diverge**, diverge.

Note that we only ever substitute unevaluated terms.

*Exercise 1.* 1. Evaluate

```
let error CRASH be x. 5
```

in CBV and CBN

2. Evaluate

```
(λx.(x + x))(print "hello". 4)
```

in CBV and CBN.

3. Evaluate

```
match (print "hello". inr error CRASH) as
  {inl x. x + 1, inr y. 5}
```

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  $M \Downarrow e$  to mean that  $M$  halts with error message  $e$ .

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

$$\begin{array}{c}
 \frac{}{\lambda x.M \Downarrow \lambda x.M} \qquad \frac{}{\text{error } e \Downarrow e} \\
 \\
 \frac{M \Downarrow \lambda x.P \quad N \Downarrow T \quad P[T/x] \Downarrow T'}{MN \Downarrow T'} \qquad \frac{M \Downarrow e}{MN \Downarrow e} \\
 \\
 \frac{M \Downarrow \lambda x.P \quad N \Downarrow e}{MN \Downarrow e} \qquad \frac{M \Downarrow \lambda x.P \quad N \Downarrow T \quad P[T/x] \Downarrow e}{MN \Downarrow e}
 \end{array}$$

Evaluation always terminates:

**Proposition 2.** *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  $M \Downarrow e$ , or
- $M \Downarrow e$  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  $\Downarrow$  for CBN, and Prop. 2 holds for these predicates.

## 5 Observational Equivalence

With the pure  $\lambda$ -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

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

This, the  $\beta$ -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

$$z : \text{bool} \vdash 3 \simeq \text{match } z \text{ as } \{\text{true}.3, \text{false}.3\} : \text{int}$$

This is an instance of the  $\eta$ -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. [·]`.

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

Next, consider the equivalence

$$\vdash \lambda x_{\text{int}}. \text{error } e \simeq \text{error } e : \text{int} \rightarrow \text{int}$$

This seems unlikely: the LHS terminates whereas the RHS raises an error. It fails in CBV: take the program context `let [·] 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

$$\vdash \lambda x_{\text{int}}. \text{print } c. M \simeq \text{print } c. \lambda x_{\text{int}}. M : \text{int} \rightarrow \text{int}$$

Again, this fails in CBV but holds in CBN.

## 6 Exercises

1. Find a context to show that

$$\begin{aligned} z : \text{bool} \vdash \\ & \text{match } z \text{ as } \{\text{true}. \text{match } z \text{ as } \{\text{true}.3, \text{false}.3\}, \text{false}.3\} \\ & \simeq \text{match } z \text{ as } \{\text{true}.3, \text{false}.3\} : \text{int} \end{aligned}$$

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  $\beta(A, B, C, D, E, F, G)$ .