

# THE SIMPLY-TYPED $\lambda$ -CALCULUS: FUNCTIONS

Alex Kavvos

Reading: PFPL §8.2

The term  $z : \text{Num} \vdash \text{plus}(z; z) : \text{Num}$  expresses the idea of doubling a number. Should we wish to use this term, we must first substitute a number—e.g. `num[57]`—for the free variable  $z$ . Instead, we would like our programming language to be able express doubling as a concept itself. That will be achieved by adding **functions**.

## 1 Statics

We extend the syntax chart with the following constructs:

types	$\tau ::= \dots$	
	$\tau_1 \rightarrow \tau_2$	function type
pre-terms	$e ::= \dots$	
	$\lambda x : \tau. e$	abstraction
	$e_1(e_2)$	application

The typing is given by the following two rules.

$$\frac{\text{LAM} \quad \Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x : \sigma. e : \sigma \rightarrow \tau}$$

$$\frac{\text{APP} \quad \Gamma \vdash e_1 : \sigma \rightarrow \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1(e_2) : \tau}$$

The first rule creates  $\lambda$ -**abstractions**: it discharges a free variable  $x : \sigma$ , thereby creating a function which accepts an argument of type  $\sigma$  and returns a result of type  $\tau$ . Hence, we may express the concept of doubling by

$$\vdash \lambda z : \text{Num}. \text{plus}(z; z) : \text{Num} \rightarrow \text{Num}$$

which is a term of **function type**.

The second rule is known as **application**, and allows the application of a function to a compatible argument.

## 2 Dynamics

The dynamics of function types are given by the following rules.

$$\frac{\text{VAL-LAM}}{\lambda x : \tau. e \text{ val}}$$

$$\frac{\text{D-APP-1} \quad e_1 \mapsto e'_1}{e_1(e_2) \mapsto e'_1(e_2)}$$

$$\frac{\text{D-APP-2} \quad e_1 \text{ val} \quad e_2 \mapsto e'_2}{e_1(e_2) \mapsto e_1(e'_2)}$$

$$\frac{\text{D-BETA}}{(\lambda x : \tau. e_1)(e_2) \mapsto e_1[e_2/x]}$$

The definition of substitution is the same as before, but extended with the clauses

$$(\lambda x : \tau. u)[e/x] \stackrel{\text{def}}{=} \lambda x : \tau. u[e/x] \quad (e_1(e_2))[e/x] \stackrel{\text{def}}{=} (e_1[e/x])(e_2[e/x])$$

Every  $\lambda$ -abstraction is a value: its body is ‘frozen’ until an argument is provided.

The rule **D-BETA** encapsulates the meaning of functions. If we have a function  $\lambda x. e_1$  is applied to an argument  $e_2$ , then we must evaluate the **body**  $e_1$  of the function with the **argument**  $e_2$  substituted for the variable  $x$ . This accords with our mathematical experience: if  $f(x) \stackrel{\text{def}}{=} x^2$  then  $f(5) = (x^2)[5/x] = 5^2$ . However, we shall now write the definition using  $\lambda$ -notation, viz. as  $f \stackrel{\text{def}}{=} \lambda x. x^2$ .

### 3 Examples

Our typing rule is the most obvious solution to adding functions. However, it is worth noting that we have perhaps obtained more than we asked: our language now has **higher-order functions**.

For example, we have the following typing derivation.

$$\begin{array}{c}
 \frac{}{x : \text{Num}, y : \text{Num} \vdash x : \text{Num}} \text{VAR} \quad \frac{}{x : \text{Num}, y : \text{Num} \vdash y : \text{Num}} \text{VAR} \\
 \hline
 \frac{}{x : \text{Num}, y : \text{Num} \vdash \text{plus}(x; y) : \text{Num}} \text{PLUS} \\
 \hline
 \frac{}{x : \text{Num} \vdash \lambda y : \text{Num}. \text{plus}(x; y) : \text{Num} \rightarrow \text{Num}} \text{LAM} \\
 \hline
 \frac{}{\vdash \underbrace{\lambda x : \text{Num}. \lambda y : \text{Num}. \text{plus}(x; y)}_{\text{add}} : \text{Num} \rightarrow (\text{Num} \rightarrow \text{Num})} \text{LAM}
 \end{array}$$

This is a function that returns a function. It corresponds to the Haskell definition

```
add :: Integer -> Integer -> Integer
add x y = x + y
```

which can also be written as

```
add :: Integer -> Integer -> Integer
add = \x -> \y -> x + y
```

This definition gives rise to the following transition sequence.

$$\begin{aligned}
 \text{add}(\text{num}[1])(\text{num}[2]) &\mapsto (\lambda y : \text{Num}. \text{plus}(\text{num}[1]; y))(\text{num}[2]) \\
 &\mapsto \text{plus}(\text{num}[1]; \text{num}[2]) \\
 &\mapsto \text{num}[3]
 \end{aligned}$$

The following is also a valid derivation, where  $\Gamma \stackrel{\text{def}}{=} f : \text{Num} \rightarrow \text{Num}, x : \text{Num}$ .

$$\begin{array}{c}
 \frac{}{\Gamma \vdash f : \text{Num} \rightarrow \text{Num}} \text{VAR} \quad \frac{}{\Gamma \vdash x : \text{Num}} \text{VAR} \\
 \hline
 \frac{}{\Gamma \vdash f(x) : \text{Num}} \text{APP} \\
 \hline
 \frac{}{f : \text{Num} \rightarrow \text{Num}, x : \text{Num} \vdash f(f(x)) : \text{Num}} \text{APP} \\
 \hline
 \frac{}{f : \text{Num} \rightarrow \text{Num} \vdash \lambda x : \text{Num}. f(f(x)) : \text{Num} \rightarrow \text{Num}} \text{LAM} \\
 \hline
 \frac{}{\vdash \underbrace{\lambda f : \text{Num} \rightarrow \text{Num}. \lambda x : \text{Num}. f(f(x))}_{\text{twice}} : (\text{Num} \rightarrow \text{Num}) \rightarrow (\text{Num} \rightarrow \text{Num})} \text{LAM}
 \end{array}$$

This is a function that both takes in and returns a function. It corresponds to the Haskell definition

```
twice :: (Num -> Num) -> Num -> Num
twice f x = f (f x)
```

This gives rise to the multi-step transition:  $\text{twice}(\text{add}(\text{num}[2]))(\text{num}[0]) \mapsto^* \text{num}[4]$ .

It is possible to obtain only first-order functions, but it requires additional effort: see PFPL §8.1.

### 4 Properties

We have completed a presentation of

**the simply-typed  $\lambda$ -calculus (STLC)** = product types + sum types + function types (+ constants)

The optional constants referred to above amount to the the basic language of numbers and strings, which consists of some **base types**—e.g. `Num` and `Str`—as well as some **primitive functions**, e.g. `plus(-; -)` and `cat(-; -)`.

The STLC satisfies the usual properties of type safety, namely progress and preservation.