The simply-typed λ-calculus: functions

Alex Kavvos

Reading: PFPL §8.2

The term $z : \mathsf{Num} \vdash \mathsf{plus}(z; z) : \mathsf{Num}$ expresses the idea of doubling a number. Should we wish to use this term, we must first substitute a number—e.g. $\mathsf{num}[57]$—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	τ	$::=$	$\ldots$	
			$\tau_1 \to \tau_2$	function type
pre-terms	e	$::=$	$\ldots$	
			$\lambda x : \tau.\, e$	abstraction
			$e_1(e_2)$	application

The typing is given by the following two rules.

$$\textsc{Lam}\ \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x : \sigma.\, e : \sigma \to \tau} \qquad \textsc{App}\ \frac{\Gamma \vdash e_1 : \sigma \to \tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1(e_2) : \tau}$$

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

$$\vdash \lambda z : \mathsf{Num}.\, \mathsf{plus}(z; z) : \mathsf{Num} \to \mathsf{Num}$$

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.

$$\textsc{Val-Lam}\ \frac{}{\lambda x : \tau.\, e\ \mathsf{val}} \qquad \textsc{D-App-1}\ \frac{e_1 \longmapsto e_1'}{e_1(e_2) \longmapsto e_1'(e_2)} \qquad \textsc{D-Beta}\ \frac{}{(\lambda x : \tau.\, e_1)(e_2) \longmapsto e_1[e_2/x]}$$

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

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

Every λ-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 λ-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.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \mathsf{Num}, y : \mathsf{Num} \vdash x : \mathsf{Num}}\textsc{Var} \quad \dfrac{}{x : \mathsf{Num}, y : \mathsf{Num} \vdash y : \mathsf{Num}}\textsc{Var}}{x : \mathsf{Num}, y : \mathsf{Num} \vdash \mathsf{plus}(x; y) : \mathsf{Num}}\textsc{Plus}}{x : \mathsf{Num} \vdash \lambda y : \mathsf{Num}.\, \mathsf{plus}(x; y) : \mathsf{Num} \to \mathsf{Num}}\textsc{Lam}}{\vdash \underbrace{\lambda x : \mathsf{Num}.\, \lambda y : \mathsf{Num}.\, \mathsf{plus}(x; y)}_{\mathsf{add}} : \mathsf{Num} \to (\mathsf{Num} \to \mathsf{Num})}\textsc{Lam}$$

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.

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

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f : \mathsf{Num} \to \mathsf{Num}}\textsc{Var} \quad \dfrac{}{\Gamma \vdash x : \mathsf{Num}}\textsc{Var}}{\Gamma \vdash f(x) : \mathsf{Num}}\textsc{App}}{\Gamma \vdash f : \mathsf{Num} \to \mathsf{Num}} \quad \dfrac{}{\Gamma \vdash f(x) : \mathsf{Num}}\textsc{App}}{f : \mathsf{Num} \to \mathsf{Num}, x : \mathsf{Num} \vdash f(f(x)) : \mathsf{Num}}\textsc{Lam}}{f : \mathsf{Num} \to \mathsf{Num} \vdash \lambda x : \mathsf{Num}.\, f(f(x)) : \mathsf{Num} \to \mathsf{Num}}\textsc{Lam}}{\vdash \underbrace{\lambda f : \mathsf{Num} \to \mathsf{Num}.\, \lambda x : \mathsf{Num}.\, f(f(x))}_{\mathsf{twice}} : (\mathsf{Num} \to \mathsf{Num}) \to (\mathsf{Num} \to \mathsf{Num})}\textsc{Lam}$$

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: $\mathsf{twice}(\mathsf{add}(\mathsf{num}[2]))(\mathsf{num}[0]) \longmapsto^* \mathsf{num}[4]$.

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 λ-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. $\mathsf{plus}(-;-)$ and $\mathsf{cat}(-;-)$.

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