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:

The typing is given by the following two rules.

$$\begin{array}{c} \text{Lam} & \text{App} \\ \hline \Gamma, x : \sigma \vdash e : \tau & \hline \Gamma \vdash \lambda x : \sigma . e : \sigma \to \tau & \hline \Gamma \vdash e_1 : \sigma \to \tau & \Gamma \vdash e_2 : \sigma \\ \hline \hline \Gamma \vdash \lambda x : \sigma . e : \sigma \to \tau & \hline \end{array}$$

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.

$$\begin{array}{c} \text{Val-Lam} & \begin{array}{c} \text{D-App-1} \\ e_1 \longmapsto e_1' \\ \hline \lambda x : \tau. \ e \ \text{val} \end{array} & \begin{array}{c} \text{D-Beta} \\ \hline (\lambda x : \tau. \ e_1)(e_2) \longmapsto e_1[e_2/x] \end{array}$$

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

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

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 λx . 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.

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

```
add :: Integer \rightarrow Integer \rightarrow Integer add x y = x + y
```

which can also be written as

add :: Integer
$$\rightarrow$$
 Integer \rightarrow Integer add = $\xspace x \rightarrow \xspace y \rightarrow x + y$

This definition gives rise to the following transition sequence.

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

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

$$\frac{\frac{\Gamma \vdash f : \mathsf{Num} \to \mathsf{Num}}{\Gamma \vdash f : \mathsf{Num} \to \mathsf{Num}} \frac{\mathsf{Var}}{\Gamma \vdash x : \mathsf{Num}} \frac{\mathsf{Var}}{\Gamma \vdash x : \mathsf{Num}} \frac{\mathsf{Var}}{\mathsf{App}}}{\frac{f : \mathsf{Num} \to \mathsf{Num}, x : \mathsf{Num} \vdash f(f(x)) : \mathsf{Num}}{f : \mathsf{Num} \to \mathsf{Num} \vdash \lambda x : \mathsf{Num}. f(f(x)) : \mathsf{Num} \to \mathsf{Num}}} \frac{\mathsf{App}}{\mathsf{Lam}}}{\mathsf{Lam}}}$$

$$\vdash \underbrace{\lambda f : \mathsf{Num} \to \mathsf{Num}. \lambda x : \mathsf{Num}. f(f(x))}_{\mathsf{bulks}} : (\mathsf{Num} \to \mathsf{Num}) \to (\mathsf{Num} \to \mathsf{Num})}_{\mathsf{bulks}}}$$

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

```
twice :: (Num \rightarrow Num) \rightarrow Num \rightarrow Num
twice f x = f (f x)
```

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

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.