Typed Lambda Calculi

Extensions to the Simply-Typed Lambda Calculus

CS 6520, Spring 2000

1 Numbers and Primitives

We can add numbers and primitives to our language in an ISWIM-like way.

New syntax:

$$
\begin{array}{lcl}
t & = & \ldots \mid n \mid (o\ t\ \ldots) \\
v & = & \ldots \mid n \\
T & = & \ldots \mid \texttt{Num} \\
n & = & \text{a number} \\
o & = & \text{a primitive operation}
\end{array}
$$

New reductions:

$$(o\ v_1\ \ldots\ v_n) \quad \longrightarrow \quad v \quad \text{where } v = \delta(o, v_1, \ldots, v_n)$$

New type rules:

$$\Gamma \vdash n : \texttt{Num}$$

$$\frac{\Gamma \vdash t_1 : T_1 \quad \ldots \quad \Gamma \vdash t_n : T_n}{\Gamma \vdash (o\ t_1\ \ldots\ t_n) : T}\ \Gamma_\delta(o) = T_1 \ldots T_n \to T$$

For the following example, define $\Gamma = f : \texttt{Num} \to \texttt{Num}, n : \texttt{Num}$:

$$
\dfrac{
\dfrac{
\dfrac{\Gamma \vdash n : \texttt{Num}}{\Gamma \vdash (\texttt{iszero}\ n) : \texttt{Bool}}
\quad
\Gamma \vdash 0 : \texttt{Num}
\quad
\dfrac{
\Gamma \vdash n : \texttt{Num}
\quad
\dfrac{
\Gamma \vdash f : \texttt{Num} \to \texttt{Num}
\quad
\dfrac{\Gamma \vdash n : \texttt{Num} \quad \Gamma \vdash 1 : \texttt{Num}}{\Gamma \vdash (-\ n\ 1) : \texttt{Num}}
}{\Gamma \vdash (f\ (-\ n\ 1)) : \texttt{Num}}
}{\Gamma \vdash (+\ n\ (f\ (-\ n\ 1))) : \texttt{Num}}
}{
\dfrac{f : \texttt{Num} \to \texttt{Num}, n : \texttt{Num} \vdash \texttt{if}\ (\texttt{iszero}\ n)\ 0\ (+\ n\ (f\ (-\ n\ 1))) : \texttt{Num}}{f : \texttt{Num} \to \texttt{Num} \vdash \lambda n : \texttt{Num}\ .\ \texttt{if}\ (\texttt{iszero}\ n)\ 0\ (+\ n\ (f\ (-\ n\ 1))) : \texttt{Num} \to \texttt{Num}}
}
}{\vdash \lambda f : \texttt{Num} \to \texttt{Num}\ .\ \lambda n : \texttt{Num}\ .\ \texttt{if}\ (\texttt{iszero}\ n)\ 0\ (+\ n\ (f\ (-\ n\ 1))) : (\texttt{Num} \to \texttt{Num}) \to \texttt{Num} \to \texttt{Num}}
$$

where $\Gamma_\delta(+) = \texttt{Num}\ \texttt{Num} \to \texttt{Num}$, and $\Gamma_\delta(-) = \texttt{Num}\ \texttt{Num} \to \texttt{Num}$.

2 Pairs

Suppose we want general functions to combine two values into one. We could try to use our old encoding:

$$\textbf{mkpair} = \lambda l.\lambda r.\lambda s.(s\ l\ r)$$

$$\textbf{fst} = \lambda p.(p\ (\lambda l.\lambda r.l))$$

$$\textbf{snd} = \lambda p.(p\ (\lambda l.\lambda r.r))$$

but it won't work with the type system we have currently. First of all, we can't write a **mkpair** function that works on any pair of values. Instead, for each pair of types T and T' that we want to join, we'd need a type-specific pairing function, $\textbf{mkpair}_{T\times T'}$. Even then, the type won't work out:

$$\textbf{mkpair}_{T\times T'} = \lambda l : T\ .\ \lambda r : T'\ .\ \lambda s : T \to T' \to \underline{?}\ .\ (s\ l\ r)$$

$$\textbf{fst}_{T\times T'} = \lambda p : (T \to T' \to T) \to T\ .\ (p\ (\lambda l : T\ .\ \lambda r : T'\ .\ l))$$

$$\textbf{snd}_{T\times T'} = \lambda p : (T \to T' \to T') \to T'\ .\ (p\ (\lambda l : T\ .\ \lambda r : T'\ .\ r))$$

The problem is that **fst** and **snd** must return different types of values, but these two functions flow into the same s in **mkpair**. There is no good type to put in place of $\underline{?}$ in the definition on **mkpair**.

Our solution (for now) is to introduce special syntax and rules for pairs.

New syntax:

$$\begin{array}{rcl} t & = & \ldots \mid \langle t, t\rangle \mid \texttt{fst}\ t \mid \texttt{snd}\ t \\ v & = & \ldots \mid \langle v, v\rangle \\ T & = & \ldots \mid T \times T \end{array}$$

New reductions:

$$\begin{array}{rcl} \texttt{fst}\ \langle v_1, v_2\rangle & \longrightarrow & v_1 \\ \texttt{snd}\ \langle v_1, v_2\rangle & \longrightarrow & v_2 \end{array}$$

$$\frac{t \longrightarrow t'}{\texttt{fst}\ t \longrightarrow \texttt{fst}\ t'} \qquad \frac{t \longrightarrow t'}{\texttt{snd}\ t \longrightarrow \texttt{snd}\ t'}$$

$$\frac{t_1 \longrightarrow t_1'}{\langle t_1, t_2\rangle \longrightarrow \langle t_1', t_2\rangle} \qquad \frac{t_2 \longrightarrow t_2'}{\langle v, t_2\rangle \longrightarrow \langle v, t_2'\rangle}$$

New type rules:

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash \langle t_1, t_2\rangle : T_1 \times T_2} \qquad \frac{\Gamma \vdash t : T_1 \times T_2}{\Gamma \vdash \texttt{fst}\ t : T_1} \qquad \frac{\Gamma \vdash t : T_1 \times T_2}{\Gamma \vdash \texttt{snd}\ t : T_2}$$

---

3 Variants

Sometimes, a function needs to accept multiple types of values, which implement variants of an abstract datatype. For example, a list-processing function might accept both **null**, a constant representing the empty list, and a cons cell (**cons** v l), where v is the first element in the list and l is the rest of the list. The function must somehow be able to distinguish the two different shapes of lists.

We implemented such variants in the untyped lambda calculus by attaching a different tag to each variant:

$$\textbf{null} = \langle true, false\rangle$$

$$\textbf{cons} = \lambda v.\lambda l.\langle false, \langle v, l\rangle\rangle$$

$$\textbf{isnull} = \lambda l.\texttt{fst}\ l$$

where the **isnull** function distinguishes empty lists from non-empty lists.

From a typing point of view, however, the above encoding does not solve the problem. The **null** constant has the type $\texttt{Bool} \times \texttt{Bool}$, whereas a cons cell has type $\texttt{Bool} \times (T \times T')$ for some T and T'. These types are incompatible, so we cannot write the **isnull** function, which is supposed take an argument of either shape.

Once again, we solve the problem by extending our type language. We will extend it with a notion of *variants* (also called *disjoint sums*), represented by $T + T'$ for variant types T and T'. A value of type $T + T'$ encapsulates either a value of type T or a value of type T'.

The syntax $\mathsf{L}_{T+T'}$ takes a value of type T and creates a value of type $T + T'$ (i.e., it *injects* the value into the sum). Similarly, the syntax $\mathsf{R}_{T+T'}$ takes a value of type T' and creates a value of type $T + T'$.

Finally, the syntax $\mathsf{M}_{(T+T')\to T''}$ takes three values. The first value is a value of type $T + T'$. If it encapsulates a T value, the second argument of $\mathsf{M}_{(T+T')\to T''}$ is called with the value. If it encapsulates a T' value, the third argument of $\mathsf{M}_{(T+T')\to T''}$ is called with the value. Both the second and third arguments must be functions that return a value of type T''.

Given this new syntax, we can make a start at encoding lists. For now, we can define **null** and a **list1** function that creates a list containg one number:

$$\textbf{null} = \mathsf{L}_{\texttt{Bool+Num}}\ \texttt{false}$$

$$\textbf{list1} = \lambda v : \texttt{Num}\ .\ \mathsf{R}_{\texttt{Bool+Num}}\ v$$

$$\textbf{isnull} = \lambda l : \texttt{Bool} + \texttt{Num}\ .\ \mathsf{M}_{(\texttt{Bool+Num})\to Bool}\ l\ (\lambda x : \texttt{Bool}\ .\ \texttt{true})\ (\lambda x : \texttt{Num}\ .\ \texttt{false})$$

New syntax:

$$\begin{array}{rcl} t & = & \ldots \mid \mathsf{L}_{T+T}\ t \mid \mathsf{R}_{T+T}\ t \mid \mathsf{M}_{(T+T)\to T}\ t\ t\ t \\ v & = & \ldots \mid \mathsf{L}_{T+T}\ v \mid \mathsf{R}_{T+T}\ v \\ T & = & \ldots \mid T + T \end{array}$$

New reductions:

$$\begin{array}{rcl} \mathsf{M}_{(T_1+T_2)\to T_3}\ (\mathsf{L}_{T_1+T_2}\ v)\ (\lambda x : T_1\ .\ t_1)\ (\lambda x : T_2\ .\ t_2) & \longrightarrow & (\lambda x : T_1\ .\ t_1)\ v \\ \mathsf{M}_{(T_1+T_2)\to T_3}\ (\mathsf{R}_{T_1+T_2}\ v)\ (\lambda x : T_1\ .\ t_1)\ (\lambda x : T_2\ .\ t_2) & \longrightarrow & (\lambda x : T_2\ .\ t_2)\ v \end{array}$$

$$\frac{t \longrightarrow t'}{\mathsf{L}_{T+T'}\ t \longrightarrow \mathsf{L}_{T+T'}\ t'} \qquad \frac{t \longrightarrow t'}{\mathsf{R}_{T+T'}\ t \longrightarrow \mathsf{R}_{T+T'}\ t'}$$

$$\frac{t_1 \longrightarrow t_1'}{\mathsf{M}_{(T_1+T_2)\to T_3}\ t_1\ t_2\ t_3 \longrightarrow \mathsf{M}_{(T_1+T_2)\to T_3}\ t_1'\ t_2\ t_3} \qquad \frac{t_2 \longrightarrow t_2'}{\mathsf{M}_{(T_1+T_2)\to T_3}\ v_1\ t_2\ t_3 \longrightarrow \mathsf{M}_{(T_1+T_2)\to T_3}\ v_1\ t_2'\ t_3}$$

$$\frac{t_3 \longrightarrow t_3'}{\mathsf{M}_{(T_1+T_2)\to T_3}\ v_1\ v_2\ t_3 \longrightarrow \mathsf{M}_{(T_1+T_2)\to T_3}\ v_1\ v_2\ t_3'}$$

New type rules:

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \mathsf{L}_{T+T'}\ t : T + T'} \qquad \frac{\Gamma \vdash t : T'}{\Gamma \vdash \mathsf{R}_{T+T'}\ t : T + T'} \qquad \frac{\Gamma \vdash t_1 : T_1 + T_2 \quad \Gamma \vdash t_2 : T_1 \to T_3 \quad \Gamma \vdash t_3 : T_2 \to T_3}{\Gamma \vdash \mathsf{M}_{(T_1+T_2)\to T_3}\ t_1\ t_2\ t_3 : T_3}$$

---

4 Recursion

As we saw in the first section, we can easily define our old **mksum** function

$$\lambda f.\lambda n.\texttt{if}\ (\texttt{iszero}\ n)\ 0\ (+\ n\ (f\ (-\ n\ 1)))$$

in our typed language:

$$\lambda f : \texttt{Num} \to \texttt{Num}\ .\ \lambda n : \texttt{Num}\ .\ \texttt{if}\ (\texttt{iszero}\ n)\ 0\ (+\ n\ (f\ (-\ n\ 1)))$$

But we also know that the simply-typed lambda calculus doesn't allow recursion, so we should expect that

$$\mathbf{sum} = \mathbf{Y}\ \mathbf{mksum}$$

does not work, for some reason. Is the reason that $\mathbf{Y}$ has no type? Partly. Certainly $\mathbf{Y}$ can't work on arbitrary functions with arbitrary types. So we need a $\mathbf{Y}$ whose type is suitable for use with **mksum**:

$$\mathbf{sum} = \mathbf{Y}_{\texttt{Num}\to\texttt{Num}}\ \mathbf{mksum}$$

The notation $\mathbf{Y}_{\texttt{Num}\to\texttt{Num}}$ means the fixpoint operator that finds fixpoints of type $\texttt{Num} \to \texttt{Num}$. Thus,

$$\mathbf{Y}_{\texttt{Num}\to\texttt{Num}} : ((\texttt{Num} \to \texttt{Num}) \to (\texttt{Num} \to \texttt{Num})) \to (\texttt{Num} \to \texttt{Num})$$

Since **mksum** has type $(\texttt{Num} \to \texttt{Num}) \to (\texttt{Num} \to \texttt{Num})$, the application $\mathbf{Y}_{\texttt{Num}\to\texttt{Num}}$ **mksum** is well-typed, right?

Almost. The problem is that $\mathbf{Y}_{\texttt{Num}\to\texttt{Num}}$ is supposed to be an abbrevation. If we replace it with its expansion, we get an expression that cannot be typed in the simply-typed lambda calculus (for the same reason that Ω has no type).

However, we *can* add a family of fixpoint operators, $\mathbf{Y}_T$ for each T, as primitives in our language, similar to $+$ and $-$. Equivalently, we can add a `fix` syntactic construct to the language, which acts like $\mathbf{Y}_T$ for an apporpriate T.

New syntax:

$$t \quad = \quad \ldots \mid \texttt{fix}\ t$$

New reductions:

$$\texttt{fix}\ (\lambda x : T\ .\ t) \quad \longrightarrow \quad \{x \mapsto (\texttt{fix}\ (\lambda x : T\ .\ t))\}t$$

$$\frac{t \longrightarrow t'}{\texttt{fix}\ t \longrightarrow \texttt{fix}\ t'}$$

New type rules:

$$\frac{\Gamma \vdash t : T \to T}{\Gamma \vdash \texttt{fix}\ t : T}$$

With this extension, we can define **sum** as

$$\mathbf{sum} = \texttt{fix}\ \mathbf{mksum}$$

Our extended language is still sound in a useful sense, even though we can have programs that never terminate. To recognize such non-termination, our soundness theorem must be modified:

> **Soundness with a fixpoint operation:** If $\vdash t : T$, then either
>
> - $t \longrightarrow^* v$ and $\vdash v : T$ for some v; or
> - for all t' such that $t \longrightarrow^* t'$, there exists a t'' such that $t' \longrightarrow t''$.
>
> In other words, the evaluation of t cannot get stuck.