Programming Languages and Types

Homework 12

Yi Dai

February 11, 2013

1 Simply-Typed λ-Calculus

1.1 Typing Derivation

Tell whether each of the following terms in the simply-typed λ-calculus with all the extensions introduced in the lecture is well-typed in the empty typing context. If it is, give a typing derivation for it; if not, give the reason. For very large terms, you can name their sub-terms and type them individually.

1. $\mathbf{pred}\ (\mathbf{succ}\ \mathbf{false})$
2. $\lambda f : \mathbf{Nat} \to \mathbf{Nat}.\lambda n : \mathbf{Nat}.f\ (f\ (\mathbf{succ}\ n))$
3. $\mathbf{if}\ (\mathbf{iszero}\ (\mathbf{succ}\ \mathbf{0}))\ \mathbf{then}\ \mathbf{true}\ \mathbf{else}\ \mathbf{0}$
4. $\{one = \mathbf{succ}\ \mathbf{0}, tru = \mathbf{true}\}\ \mathbf{as}\ \{tru : \mathbf{Bool}, one : \mathbf{Nat}\}$
5. $\mathbf{let}\ b = \mathbf{false}\ \mathbf{in}\ (\mathbf{iszero}\ b)$
6. $\mathbf{let}\ p = (\mathbf{0}, \mathbf{succ}\ \mathbf{0})\ \mathbf{in}\ (\mathbf{snd}\ p, \mathbf{fst}\ p)$
7. $\mathbf{case}\ (\mathbf{inl}\ \mathbf{0})\ \mathbf{of}\ \mathbf{inl}\ x \Rightarrow \mathbf{false} \mid \mathbf{inr}\ x \Rightarrow \mathbf{true}$

8.

$$
\begin{aligned}
&\textbf{fix}\ (\lambda\ \mathit{fise} : \textbf{Nat} \to \textbf{Bool}\ .\\
&\qquad \lambda\ n : \textbf{Nat}\ .\\
&\qquad\quad \textbf{if}\ (\textbf{iszero}\ n)\\
&\qquad\qquad \textbf{then true}\\
&\qquad\qquad \textbf{else if}\ (\textbf{iszero}\ (\textbf{pred}\ n))\\
&\qquad\qquad\qquad \textbf{then false}\\
&\qquad\qquad\qquad \textbf{else}\ \mathit{fise}\ (\textbf{pred}\ (\textbf{pred}\ n))\ )
\end{aligned}
$$

1.2 Programming with Extensions

1. Complete the addition function *add* : **Nat** $\to$ **Nat** $\to$ **Nat** in the simply-typed λ-calculus with base type **Nat** and extension the fixed-point operator **fix**.

$$\mathit{add} = \textbf{fix}\ (\lambda\ \mathit{fadd} : \textbf{Nat} \to \textbf{Nat} \to \textbf{Nat}\ .\ ?)$$

2 System-$\mathcal{F}$

2.1 Parametric Polymorphism

1. Define a function called *twice* that applies a function to an argument twice.

2. Use the function *twice* to define a function called *thrice* that applies a function to an argument for three times.

3. Define a function called *compose* that composes two functions.

2.2 Typing Church-Encodings

Refer to the Church-encodings for numerals, booleans and lists.[1] Note that, for all exercises, you should also give the type of the whole term.

1. Define the multiplication function *cmul* for Church-numerals. Do it first using the *cadd* function already given in the slides. Then try to give a definition directly. (*Hint*: For the latter, consider how many times the product of two Church-numerals means to iterating a function.)

[1]The encodings for booleans I showed in the exercise session is kinda over-generalized. You should use the simpler one given in the slides.

2. Define the boolean-or function *cor* for Church-booleans.

3. Define the *crev* that reverses a Church-encoded list.