CS520, Handout 08, 26 September 2006

Extensions of the Simply Typed Lambda Calculus

Reading: [Pierce, Chapter 11]

We can organize extensions of the simply-typed lambda-calculus into two groups:

1. **Extensions that do not require new forms in the syntax of types,**
   including **ascription**, `let`**-binding**, **recursion** and `letrec`**-binding**.
   [Pierce, Sections 11.4, 11.5, and 11.11].

2. **Extensions that require new forms in the syntax of types,**
   including `unit` and **sequencing**, **pairs**, **tuples**, **records**, **sums**, **variants** and **lists**.
   [Pierce, Sections 11.2, 11.3, 11.6, 11.7, 11.8, 11.9, 11.10, and 11.12].

We start with the first group, and then continue with the second group. Different from the order in [Pierce].

Before we introduce each of the extensions, assume we are dealing with $\lambda\mathrm{NB}_{\to}$, not just $\lambda\mathrm{B}_{\to}$ as in [Pierce, Chapter 9], i.e., with the simply-typed lambda-calculus augmented with booleans and natural numbers and their primops.

Adding ascription [Pierce, Section 11.4]

Syntactic form for ascribed terms

$$t ::= \ \dots \qquad \textit{terms}$$
$$\qquad t \ \mathsf{as}\ T \qquad \textit{ascription}$$

Evaluation rules for ascribed terms

$$v \ \mathsf{as}\ T \Rightarrow v \qquad \text{(E-ASCRIBE)}$$

$$\frac{t \Rightarrow t'}{t \ \mathsf{as}\ T \Rightarrow t' \ \mathsf{as}\ T} \qquad \text{(E-ASCRIBE1)}$$

Typing rules for ascribed terms

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t \ \mathsf{as}\ T : T} \qquad \text{(T-ASCRIBE)}$$

Uses of ascribed terms

- Program **documentation**.
- Controlling **printing** of complex types, via the use of **type abbreviations**.

  $\mathtt{NN} = \mathtt{Nat} \rightarrow \mathtt{Nat};$

  $(\lambda \mathtt{f} : \mathtt{Nat} \rightarrow \mathtt{Nat}.\,\mathtt{f}\,)$ as $\mathtt{NN} \rightarrow \mathtt{NN};$
- A form of **casting**, for terms that may have more than one type.

Adding `let`-binding [Pierce, Section 11.5]

Syntactic form for `let`***-bindings***

$$t ::= \ldots \qquad \textit{terms}$$

$$\texttt{let}\ x = t\ \texttt{in}\ t \qquad \texttt{let}\ \textit{binding}$$

Evaluation rules for `let`***-bindings***

$$\texttt{let}\ x = v\ \texttt{in}\ t \Rrightarrow [x \mapsto v]t \qquad \text{(E-LetV)}$$

$$\frac{t_1 \Rrightarrow t_1'}{\texttt{let}\ x = t_1\ \texttt{in}\ t_2 \Rrightarrow \texttt{let}\ x = t_1'\ \texttt{in}\ t_2} \qquad \text{(E-Let)}$$

Typing rules for `let`***-bindings***

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 : T_2} \qquad \text{(T-Let)}$$

Adding recursion and `letrec`-binding [Pierce, Section 11.11]

Syntactic form for recursion

$$t ::= \ldots \qquad \textit{terms}$$

$$\texttt{fix}\, t \qquad \textit{fixed point of } t$$

Evaluation rules for recursion

$$\texttt{fix}\,(\lambda x : T.\, t\,) \Rrightarrow [x \mapsto (\texttt{fix}\,(\lambda x : T.\, t\,))]t \qquad \text{(E-FixBeta)}$$

$$\frac{t \Rrightarrow t'}{\texttt{fix}\, t \Rrightarrow \texttt{fix}\, t'} \qquad \text{(E-Fix)}$$

Typing rules for recursion

$$\frac{\Gamma \vdash t : T \to T}{\Gamma \vdash \texttt{fix}\, t : T} \qquad \text{(T-Fix)}$$

Derived form ("syntactic sugar")

$$\texttt{letrec}\ x : T = t_1\ \texttt{in}\ t_2 \quad \triangleq \quad \texttt{let}\ x = \texttt{fix}\,(\lambda x : T.\, t_1\,)\ \texttt{in}\ t_2$$

Example of a recursive function

```
letrec iseven : Nat -> Bool =        (* test for even numbers *)
     lambda x: Nat.
        if (iszero x) then true
        else if (iszero (pred x)) then false
             else (iseven (pred (pred x)))
in (iseven 7);
```


Example of a recursive function

```
letrec iseven : Nat -> Bool =        (* test for even numbers *)
     lambda x: Nat.
        if (iszero x) then true
        else if (iszero (pred x)) then false
             else (iseven (pred (pred x)))
in (iseven 7);
```

Desugar “letrec” to obtain:

```
ff = lambda f: Nat -> Bool.          (* ff is not recursive *)
     lambda x: Nat.
        if (iszero x) then true
        else if (iszero (pred x)) then false
             else (f (pred (pred x)));
let iseven = fix ff
in  (iseven 7);
```