1 Announcements

Homework 3 is due today. Please turn them in. Homework 4 is available on the web and is due next Thursday (Oct 26). This is a programming homework and is reasonably substantial, so start early.

2 Introduction

We completed an important part of the class: typed lambda calculus and its type safety. Although lambda calculus is interesting on its own to study the properties of languages and language design, it is not a practical language—it lacks many features that you would expect from a practical language. In this class, we will grow typed lambda calculus with some features including base types, let bindings, product and sum types and recursion.

3 Base Types

We often want our language to have various base types such as unit type, naturals. Let’s add unit types and naturals to our language. We first have to decide the values of these types. Let denote the only value in the unit type as ⊤ and define the natural numbers as usual. We then decide what kind of primitive operations that we want to operate on these base types. For the unit type, we don’t require
any primitive operations, but for naturals, we can have addition, multiplication, comparison.

<table>
<thead>
<tr>
<th>Types</th>
<th>τ ::= unit</th>
<th>nat</th>
<th>τ₁ → τ₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numbers</td>
<td>n ::= 0 ⋯ 1 ⋯</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prim op’s</td>
<td>o ::= + ⋯ − ⋯ ×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Values</td>
<td>v ::= ⋯ n</td>
<td>λx : τ.t</td>
<td></td>
</tr>
<tr>
<td>Terms</td>
<td>t ::= x ⋯ v ⋯ o(t₁,t₂) ⋯ t t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Context</td>
<td>Γ ::= ⌀ ⋯ Γ,x : τ</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We can give the following type system for this extension of lambda calculus.

<table>
<thead>
<tr>
<th>Γ ⊢ * : unit (unit)</th>
<th>Γ ⊢ n : nat (naturals)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x : τ ∈ Γ</td>
<td>Γ ⊢ x : τ (variables)</td>
</tr>
<tr>
<td>Γ ⊢ t₁ : nat</td>
<td>Γ ⊢ t₂ : nat (prim. op’s)</td>
</tr>
<tr>
<td>Γ ⊢ o(t₁,t₂) : nat</td>
<td>Γ ⊢ o : (t₁,t₂) → o (app)</td>
</tr>
<tr>
<td>Γ,x : τ₁ ⊢ t : τ₂</td>
<td>Γ ⊢ λx : τ₁.t : τ₁ → τ₂ (lambda)</td>
</tr>
<tr>
<td>Γ ⊢ t₁ : τ₁ → τ₂</td>
<td>Γ ⊢ t₂ : τ₂</td>
</tr>
</tbody>
</table>

Similarly, we can extend the CBV (call-by-value) operational semantics for lambda calculus to support our base types. For the operational semantics, we assume that we have a primitive application denoted @ that given a primitive operation and the arguments for that operation gives us the value back. For example @(+, 2, 3) = 5.

<table>
<thead>
<tr>
<th>t₁ → t₁’</th>
<th>o(t₁,t₂) → o(t₁,t₂’)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₂ → t₂’</td>
<td>o(t₁,t₂) → o(t₁,t₂’)</td>
</tr>
<tr>
<td>t₁ → t₁’</td>
<td>o(n₁,n₂) → @o(n₁,n₂)</td>
</tr>
<tr>
<td>t₁ t₂ → t₁’ t₂</td>
<td>t₁ t₂ → t₁ t₂</td>
</tr>
<tr>
<td>(λx : τ.t)v → [v/x] t</td>
<td></td>
</tr>
</tbody>
</table>

4 Let Bindings and Derived Forms

It is often useful to be able to bind the value of an expression to a variable (e.g., SML’s let construct). We can do this by extending our language with a let binding.
\[ t ::= \ldots \mid \text{let } x : \tau_1 = t \text{ in } t \text{ end} \]

Why do we need to specify the type of the variable being bound? As it will become clear when we write the typing rule for let, we will not be able to know what type to give to first part otherwise.

\[
\frac{t_1 \rightarrow t_1'}{\text{let } x : \tau_1 = t_1 \text{ in } t_2 \text{ end} \rightarrow \text{let } x : \tau_1 = t_1' \text{ in } t_2 \text{ end}} \text{(eval-let-1)}
\]

\[
\frac{\text{let } x : \tau_1 = v \text{ in } t_2 \text{ end} \rightarrow [v/x] t_2}{v/x} \text{(eval-let-2)}
\]

\[
\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash t_2 : \tau_2}{\Gamma \vdash \text{let } x : \tau_1 = t_1 \text{ in } t_2 \text{ end} : \tau_2} \text{(type-let)}
\]

Both the evaluation rules and the typing rules look familiar to another expression that we know: application. Indeed, the let expression \( \text{let } x : \tau_1 = t_1 \text{ in } t_2 \text{ end} \) is equivalent to the application \((\lambda x : \tau_1.t_2) t_1\). In other words let bindings are derived forms—they can be derived using simply typed lambda calculus.

We can formally prove that let bindings are derived forms by supplying an elaboration function that maps the terms of the language with let bindings—call this the external language—to typed lambda calculus—call this the internal language. Formally \( \eta : t^e \mapsto t^i \), where \( t^e \) and \( t^i \) are the terms for the external and the internal languages respectively. The elaboration function \( \eta \) simply replaces let bindings with the corresponding application and leaves all other terms the same. We can then prove derivability by proving

1. \( \Gamma \vdash^e t^e : \tau \) if and only if \( \Gamma \vdash^i (\eta(t^e)) : \tau \)
2. \( t_1^i \xrightarrow{i} t_2^i \) if and only if \( \eta(t_1^i) \xrightarrow{i} \eta(t_2^i) \)

\textbf{Exercise:} Prove that let bindings is derived form by specifying the elaboration function \( \eta \) and proving the two properties.

## 5 Pairs and Product Types

Most languages provide a way to build compound data structures. Perhaps the most basic form for this is pairing. Extending typed lambda calculus to support pairs is reasonably straightforward. We first introduce a \textit{product type} for representing pairs: the type of a pair the the product of the types of its components.

\[ \tau ::= \ldots \mid \tau \times \tau \tau t ::= \ldots \mid (t, t) \mid \text{first}(t) \mid \text{second}(t) \]
The first(·) and second(·) forms project out the first and second parts of a pair. For example first((1, 2)) = 1, second((1, 2)) = 2.

The following are the typing rules.

\[
\Gamma \vdash t_1 : \tau_1 \quad \Gamma \vdash t_2 : \tau_2 \quad \frac{\text{(type-pair)}}{\Gamma \vdash \langle t_1, t_2 \rangle : \tau_1 \times \tau_2}
\]

\[
\Gamma \vdash t : \tau_1 \times \tau_2 \quad \frac{\text{(type-project-1)}}{\Gamma \vdash \text{first}(t) : \tau_1}
\]

\[
\Gamma \vdash t : \tau_1 \times \tau_2 \quad \frac{\text{(type-project-2)}}{\Gamma \vdash \text{second}(t) : \tau_2}
\]

Here are the evaluation rules for pairs.

\[
\begin{align*}
t_1 \rightarrow t_2 & \quad \frac{\text{(eval-pair/1)}}{\langle t_1, t_2 \rangle \rightarrow \langle t'_1, t'_2 \rangle} \\
t_2 \rightarrow t'_2 & \quad \frac{\text{(eval-pair/2)}}{\langle t_1, t_2 \rangle \rightarrow \langle t_1, t'_2 \rangle}
\end{align*}
\]

\[
\begin{align*}
t \rightarrow t' & \quad \frac{\text{(eval-first/1)}}{\text{first}(t) \rightarrow \text{first}(t')} \\
t \rightarrow t' & \quad \frac{\text{(eval-first/2)}}{\text{first}(\langle v_1, v_2 \rangle) \rightarrow v_1}
\end{align*}
\]

\[
\begin{align*}
t \rightarrow t' & \quad \frac{\text{(eval-second/1)}}{\text{second}(t) \rightarrow \text{second}(t')} \\
t \rightarrow t' & \quad \frac{\text{(eval-second/2)}}{\text{second}(\langle v_1, v_2 \rangle) \rightarrow v_2}
\end{align*}
\]

6 Heterogeneous Data and Sum Types

We often want to express data that has heterogeneous nature. For example, a list can empty empty (nil) or can have a head or tail (a cons cell). Similarly a tree can be empty or it can be a node consisting of two children and some data.

It is well known that many programming bugs simply result misuse of such data. For example, in the C language, any pointer is either a valid pointer or it is null. But the pointer type in C does not reflect this fact; typical C programs are full of such pointer errors (accessing null pointers, etc). It is therefore critical to ensure type safety of heterogenous data so that their misuse can be reduced.

What should the type of a heterogeneous data be? First, the type must represent all possible forms of data. Second, it should be possible to determine the form of the data by inspecting it; one way to achieve this is to tag data.

As a concrete example, suppose we want to have data that can either be of type unit or of type nat. We can write the type of such data as unit + nat to indicate that it can be either one of these types. If we think of types as sets of terms, the set of this type is the union of the set of terms of type unit unioned with the set of terms of type nat. How can we write terms of this type. Remember that we want a way to tell which form the data is. So one option is to write inl(t) for terms where t has type unit. and inr(t), where t has type nat. The tags can then tell us what to expect from the enclosed term.

Why do we need the tags? So far we have only talked about introduction forms for sums. We need the tags for the elimination form : case that allows us
to investigate the tag of a sum type and perform an operation on its contents.
For example, the following term `inspect t` and prints “star” if the term is a ⋆ or
prints the natural number.

case t of inl x ⇒ print "star" | inr x ⇒ print "natural : " x.

Let’s make this intuitive description more concrete by giving the typing and
evaluation rules.

$$\begin{align*}
\tau & ::= \ldots | \tau_1 + \tau_2 \\
v & ::= inl_{\tau_1+\tau_2}(v) | inr_{\tau_1+\tau_2}(t) \\
t & ::= inl_{\tau_1+\tau_2}(t) | inr_{\tau_1+\tau_2}(t) | (case t_1 of inl(x) ⇒ t_2 | inr(x) ⇒ t_3)
\end{align*}$$

$$\begin{align*}
\Gamma \vdash t_1 : \tau_1 & \quad \Gamma \vdash t_2 : \tau_2 \\
\Gamma \vdash inl_{\tau_1+\tau_2}(t) : \tau_1 + \tau_2 & \quad \Gamma \vdash inr_{\tau_1+\tau_2}(t) : \tau_1 + \tau_2 \\
\Gamma \vdash t_0 : \tau_1 + \tau_2 & \quad \Gamma, x_1 : \tau_1 \vdash t_1 : \tau \\
\Gamma, x_2 : \tau_2 \vdash t_2 : \tau & \quad \Gamma \vdash case t_0 of inl_{\tau_1+\tau_2}(x_1) ⇒ t_1 | inr_{\tau_1+\tau_2}(x_2) ⇒ t_2 : \tau
\end{align*}$$

Note that we have to require the programmer specify the type of a sum
type. This is important because otherwise we don’t know what type to assign
to a term. For example, `inl(1)` can have type bool + nat or unit + nat. By
requiring the programmer to specify the sum type, we can ensure that each term
has unique type.

The evaluation rules follow:

$$\begin{align*}
t & \to t' & \quad t & \to t' \\
inl \tau t & \to inl \tau, t' & \quad inr \tau t & \to inr \tau, t'
\end{align*}$$

$$\begin{align*}
t_0 & \to t'_0 \quad case t_0 of inl_{\tau_1+\tau_2}(x_1) ⇒ t_1 | inr_{\tau_1+\tau_2}(x_2) ⇒ t_2 \to case t'_0 of inl_{\tau_1+\tau_2}(x_1) ⇒ t_1 | inr_{\tau_1+\tau_2}(x_2) ⇒ t_2
\end{align*}$$

case inl_\tau v of inl_{\tau_1+\tau_2}(x_1) ⇒ t_1 | inr_{\tau_1+\tau_2}(x_2) ⇒ t_2 \to [v/x] t_1

case inr_\tau v of inl_{\tau_1+\tau_2}(x_1) ⇒ t_1 | inr_{\tau_1+\tau_2}(x_2) ⇒ t_2 \to [v/x] t_2

Exercise: Given the extension of lambda calculus with sum types, prove
that booleans are derived forms.

7 Recursion

Previously, we showed that recursion can be “simulated” in untyped lambda
calculus using the Y and Z combinators as a fixed-point operator. The Y combi-
nator worked for call-by-name semantics, whereas for call by value, we needed
a slightly more complicated version, which is known as the Z combinator. It
turns out that none of these combinators can be given a type. Note that by
definition, types must be finite. It is instructive to try to give a type for Y and
Z and see where things fail.

We therefore do not know have a way to express recursion in the typed
lambda calculus in the calculus itself. In this class, we will instead introduce
direct support for recursion by allowing the programmer express recursive func-
tions directly. In particular, we will allow the programmer write a recursive
function as \texttt{fix }f(x) : \tau \text{ is } t \text{ end}.

For example a factorial function can be written as

\begin{align*}
\text{fix fact}(x) &: \text{nat }\rightarrow \text{nat is if } x < 1 \text{ then } x \text{ else } x * \text{fact}(x - 1) \text{ end}
\end{align*}

The typing rule for the \texttt{fix} operator is very similar to that of lambda ab-
straction, except that when type-checking the body, we get to assume that the
function being defined \texttt{f} has the specified type. This allow the body of the
defined function to mention itself recursively.

\[
\Gamma, f: \tau_1 \rightarrow \tau_2, x : \tau_1 \vdash t : \tau_2 \\
\Gamma \vdash \texttt{fix } f(x) : \tau_1 \rightarrow \tau_2 \text{ is } t \text{ end}; \tau_1 \rightarrow \tau_2
\]

In the operational semantics, we change the application rule so that the
function is substituted for itself—this provides for recursion. It is instructive to
contrast this to what the Y and Z combinators achieve in the untyped setting.

\[
t_2 \rightarrow t_2' \\
(fix f(x) : \tau_1 \rightarrow \tau_2 \text{ is } t \text{ end}) t_2 \rightarrow fix f(x) : \tau_1 \rightarrow \tau_2 \text{ is } t \text{ end } t_2'
\]

\[
(fix f(x) : \tau_1 \rightarrow \tau_2 \text{ is } t \text{ end}) v \rightarrow [v/x, fix f(x) : \tau_1 \rightarrow \tau_2 \text{ is } t \text{ end }/f] t
\]