1 More types

We have previously explored the dynamic semantics of a number of language features. Here, we consider how to extend the type system of lambda calculus for some of the language features we saw previously, and some new ones.

1.1 Product and sums

We have previously seen products, which are pairs of expressions. Products were constructed using the expression $(e_1, e_2)$, and destructed using projection #1 $e$ and #2 $e$.

In addition to the structural rules, there are two operational semantics rules that show how the destructors and constructor interact.

\[
\begin{align*}
#1 \ (v_1, v_2) &\rightarrow v_1 \\
#2 \ (v_1, v_2) &\rightarrow v_2 
\end{align*}
\]

The type of a product expression (or a product type) is a pair of types, written $\tau_1 \times \tau_2$. The typing rules for the product constructors and destructors are the following.

\[
\begin{align*}
\Gamma \vdash e_1 : \tau_1 & \quad \Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2 & \\
\Gamma \vdash #1 e : \tau_1 & \\
\Gamma \vdash #2 e : \tau_2 
\end{align*}
\]

We introduce sums, which are dual to products. Intuitively, a product holds two values, one of type $\tau_1$, and one of type $\tau_2$. By contrast, a sum holds a single value that is either of type $\tau_1$ or of type $\tau_2$. The type of a sum is written $\tau_1 + \tau_2$. There are two constructors for a sum, corresponding to whether we are constructing a sum with a value of $\tau_1$ or a value of $\tau_2$.

\[
e ::= \cdots | \text{inl}_{\tau_1 + \tau_2} e | \text{inr}_{\tau_1 + \tau_2} e | \text{case} \ e_1 \ \text{of} \ e_2 | e_3
\]

\[
v ::= \cdots | \text{inl}_{\tau_1 + \tau_2} v | \text{inr}_{\tau_1 + \tau_2} v
\]

Again, there are structural rules to determine the order of evaluation. In a CBV lambda calculus, the evaluation contexts are extended as follows.

\[
E ::= \cdots | \text{inl}_{\tau_1 + \tau_2} E | \text{inr}_{\tau_1 + \tau_2} E | \text{case} \ E \ \text{of} \ e_2 | e_3
\]

In addition to the structural rules, there are two operational semantics rules that show how the destructors and constructors interact.

\[
\begin{align*}
\text{case} \ \text{inl}_{\tau_1 + \tau_2} v \ \text{of} \ e_2 \ | \ e_3 &\rightarrow e_2 v \\
\text{case} \ \text{inr}_{\tau_1 + \tau_2} v \ \text{of} \ e_2 \ | \ e_3 &\rightarrow e_3 v
\end{align*}
\]

The type of a sum expression (or a sum type) is written $\tau_1 + \tau_2$. The typing rules for the sum constructors and destructor are the following.

\[
\begin{align*}
\Gamma \vdash e : \tau_1 & \\
\Gamma \vdash \text{inl}_{\tau_1 + \tau_2} e : \tau_1 + \tau_2 & \\
\Gamma \vdash \text{inr}_{\tau_1 + \tau_2} e : \tau_1 + \tau_2 & \\
\Gamma \vdash \text{case} \ e \ \text{of} \ e_1 \ | \ e_2 : \tau & \\
\end{align*}
\]
Let’s see an example of a program that uses sum types.

```plaintext
let f : (int + (int → int)) → int =
  λa : int + (int → int). case a of λy.y + 1 | λg.g 35 in

let h : int → int = λx : int. x + 7 in
f (inr int + (int → int) h)
```

Here, the function `f` takes argument `a`, which is a sum. That is, the actual argument for `a` will either be a value of type `int` or a value of type `int → int`. We destroy the sum value with a case statement, which must be prepared to take either of the two kinds of values that the sum may contain. We end up applying `f` to a value of type `int → int` (i.e., a value injected into the right type of the sum). The entire program ends up evaluating to 42.

### 1.2 Recursion

We saw in last lecture that we could not type recursive functions or fixed-point combinators in the simply-typed lambda calculus. So instead of trying (and failing) to define a fixed-point combinator in the simply-typed lambda calculus, we add a new primitive `µx : τ. e` to the language. The evaluation rules for the new primitive will mimic the behavior of fixed-point combinators.

We extend the syntax with the new primitive operator. Intuitively, `µx : τ. e` is the fixed-point of the function `λx : τ. e`. Note that `µx : τ. e` is not a value, regardless of whether `e` is a value or not.

```
e ::= ⋯ | µx : τ. e
```

We extend the operational semantics for the new operator. There is a new axiom, but no new evaluation contexts.

```
µx : τ. e → e{(µx : τ. e)/x}
```

Note that we can define the `letrec x : τ = e1 in e2` construct in terms of this new expression.

```
letrec x : τ = e1 in e2 ≜ let x : τ = µx : τ. e1 in e2
```

We add a new typing rule for the new language construct.

```
Γ ⊢ e : τ  
Γ ⊢ µx : τ. e : τ
```

Returning to our trusty factorial example, the following program implements the factorial function using the `µx : τ. e` expression.

```
FACT ≜ µf : int → int. λn : int. if n = 0 then 1 else n × (f (n − 1))
```

Or using our convenient `letrec` notation, we could define a variable `fact` as follows.

```
letrec fact : int → int = λn : int. if n = 0 then 1 else n × (fact (n − 1)) in...
```

We can write non-terminating computations for any type: the expression `µx : τ. x` has type `τ`, and does not terminate.

Although the `µx : τ. e` expression is normally used to define recursive functions, it can be used to find fixed points of any type. For example, consider the following expression.

```
µx : (int → bool) × (int → bool). (λn : int. if n = 0 then true else (((2 x) (n − 1))
  λn : int. if n = 0 then false else (((1 x) (n − 1))
```

Page 2 of 4
This expression has type \( (\text{int} \to \text{bool}) \times (\text{int} \to \text{bool}) \)—it is a pair of mutually recursive functions; the first function returns \text{true} only if its argument is even; the second function returns \text{true} only if its argument is odd.

1.3 References

Recall the syntax and semantics for references.

\[
e ::= \cdots | \text{ref}\ e \ | \ e_1 := e_2 \ | \ell
\]
\[
v ::= \cdots | \ell
\]
\[
E ::= \cdots | E \text{ref} \ | E ! E \ | e := v := E
\]

\[
\text{ALLOC} \quad \frac{(\text{ref} v, \sigma) \rightarrow (\ell, \sigma[\ell \mapsto v])}{\ell \notin \text{dom}(\sigma)}
\]
\[
\text{DEREF} \quad \frac{(\ell, \sigma) \rightarrow (v, \sigma)}{\sigma(\ell) = v}
\]
\[
\text{ASSIGN} \quad \frac{(\ell := v, \sigma) \rightarrow (v, \sigma[\ell \mapsto v])}{(\ell := v, \sigma) \rightarrow (v, \sigma[\ell \mapsto v])}
\]

We add a new type for references: type \( \tau \text{ ref} \) is the type of a location that contains a value of type \( \tau \). For example the expression \text{ref} \ 7 has type \( \text{int} \text{ ref} \), since it evaluates to a location that contains a value of type \( \text{int} \). Dereferencing a location of type \( \tau \text{ ref} \) results in a value of type \( \tau \), so \( ! e \) has type \( \tau \) if \( e \) has type \( \tau \text{ ref} \).

And for assignment \( e_1 := e_2 \), if \( e_1 \) has type \( \tau \text{ ref} \), then \( e_2 \) must have type \( \tau \).

\[
\tau ::= \cdots | \tau \text{ ref}
\]
\[
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{ref} \ e : \tau \text{ ref}}
\]
\[
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash ! e : \tau}
\]
\[
\frac{\Gamma \vdash e_1 : \tau \text{ ref} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \tau}
\]

Noticeably by its absence is a typing rule for location values. What is the type of a location value \( \ell \)? Clearly, it should be of type \( \tau \text{ ref} \), where \( \tau \) is the type of the value contained in location \( \ell \). But how do we know what value is contained in location \( \ell \)? We could directly examine the store, but that would be inefficient. In addition, examine the store directly may not give us a conclusive answer!

Consider, for example, a store \( \sigma \) and location \( \ell \) where \( \sigma(\ell) = \ell \), what is the type of \( \ell \)?

Instead, we introduce store typings to track the types of values stored in locations. Store typings are partial functions from locations to types. We use metavariable \( \Sigma \) to range over store typings. Our typing relation now becomes a relation over 4 entities: typing contexts, store typings, expressions, and types. We write \( \Gamma, \Sigma \vdash e : \tau \) when expression \( e \) has type \( \tau \) under typing context \( \Gamma \) and store typing \( \Sigma \).

Our new typing rules for references are as follows. (Typing rules for other constructs are modified to take a store typing in the obvious way.)

\[
\frac{\Gamma, \Sigma \vdash e : \tau}{\Gamma, \Sigma \vdash \text{ref} \ e : \tau \text{ ref}}
\]
\[
\frac{\Gamma, \Sigma \vdash e : \tau}{\Gamma, \Sigma \vdash ! e : \tau}
\]
\[
\frac{\Gamma, \Sigma \vdash e_1 : \tau \text{ ref} \quad \Gamma, \Sigma \vdash e_2 : \tau}{\Gamma, \Sigma \vdash e_1 := e_2 : \tau}
\]
\[
\frac{\Gamma, \Sigma \vdash e : \tau \quad \Gamma, \Sigma \vdash e_1 : \tau \text{ ref} \quad \Gamma, \Sigma \vdash e_2 : \tau}{\Gamma, \Sigma \vdash \Sigma(\ell) = \tau \text{ ref}}
\]

So, how do we state type soundness? Our type soundness theorem for simply-typed lambda calculus said that if \( \Gamma \vdash e : \tau \) and \( e \rightarrow^* e' \) then \( e' \) is not stuck. But our operational semantics for references now has a store, and our typing judgment now has a store typing in addition to a typing context. We need to adapt the definition of type soundness appropriately. To do so, we define what it means for a store to be well-typed with respect to a typing context.

**Definition.** Store \( \sigma \) is well-typed with respect to typing context \( \Gamma \) and store typing \( \Sigma \), written \( \Gamma, \Sigma \vdash \sigma \), if \( \text{dom}(\sigma) = \text{dom}(\Sigma) \) and for all \( \ell \in \text{dom}(\sigma) \) we have \( \Gamma, \Sigma \vdash \sigma(\ell) : \Sigma(\ell) \).

We can now state type soundness for our language with references.
Theorem (Type soundness). If $\Gamma, \Sigma \vdash e : \tau$ and $\Gamma, \Sigma \vdash \sigma$ and $\langle e, \sigma \rangle \rightarrow^* \langle e', \sigma' \rangle$ then either $e'$ is a value, or there exists $e''$ and $\sigma''$ such that $\langle e', \sigma' \rangle \rightarrow \langle e'', \sigma'' \rangle$.

We can prove type soundness for our language using the same strategy as for the simply-typed lambda calculus: we use preservation and progress. The progress lemma can be easily adapted for the semantics and type system for references. Adapting preservation is a little more involved, since we need to describe how the store typing changes as the store evolves. The rule $\text{ALLOC}$ extends the store $\sigma$ with a fresh location $\ell$, producing store $\sigma'$. Since $\text{dom}(\Sigma) = \text{dom}(\sigma) \neq \text{dom}(\sigma')$, it means that we will not have $\sigma'$ well-typed with respect to typing store $\Sigma$.

Since the store can increase in size during the evaluation of the program, we also need to allow the store typing to grow as well.

Lemma (Preservation). If $\Gamma, \Sigma \vdash e : \tau$ and $\Gamma, \Sigma \vdash \sigma$ and $\langle e, \sigma \rangle \rightarrow \langle e', \sigma' \rangle$ then there exists some $\Sigma' \supseteq \Sigma$ such that $\Gamma, \Sigma' \vdash e' : \tau$ and $\Gamma, \Sigma' \vdash \sigma'$.

We write $\Sigma' \supseteq \Sigma$ to mean that for all $\ell \in \text{dom}(\Sigma)$ we have $\Sigma(\ell) = \Sigma'(\ell)$. This makes sense if we think of partial functions as sets of pairs: $\Sigma \equiv \{ (\ell, v) \mid \ell \in \text{dom}(\Sigma) \wedge \Sigma(\ell) = v \}$.

Note that the preservation lemma states simply that there is some store type $\Sigma' \supseteq \Sigma$, but does not specify what exactly that store typing is. Intuitively, $\Sigma'$ will either be $\Sigma$, or $\Sigma$ extended on a single, newly allocated, location.