CS 4110 – Programming Languages and Logics
Lectures #16: More Types

Based on material by Stephen Chong, Greg Morrisett, Andrew Myers, George Necula, and Radu Rugina

1 More types

We previously explored the dynamic semantics of a number of language features. Today we consider how to extend the type system of lambda calculus for some of the language features we saw previously, as well as 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.

$$\frac{}{\#1\ (v_1, v_2) \rightarrow v_1} \qquad \frac{}{\#2\ (v_1, v_2) \rightarrow v_2}$$

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.

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \#1\ e : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \#2\ e : \tau_2}$$

We introduce *sums*, which are dual to products. Intuitively, a product holds two values, one of type τ_1, and one of type τ_2. By contrast, a sum holds a single value that is either of type τ_1 or of type τ_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 τ_1 or a value of τ_2.

$$\begin{aligned} e &::= \cdots \mid \mathsf{inl}_{\tau_1+\tau_2}\ e \mid \mathsf{inr}_{\tau_1+\tau_2}\ e \mid \mathsf{case}\ e_1\ \mathsf{of}\ e_2 \mid e_3 \\ v &::= \cdots \mid \mathsf{inl}_{\tau_1+\tau_2}\ v \mid \mathsf{inr}_{\tau_1+\tau_2}\ v \end{aligned}$$

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

$$E ::= \cdots \mid \mathsf{inl}_{\tau_1+\tau_2}\ E \mid \mathsf{inr}_{\tau_1+\tau_2}\ E \mid \mathsf{case}\ E\ \mathsf{of}\ e_2 \mid e_3$$

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

$$\frac{}{\textsf{case inl}_{\tau_1+\tau_2}\ v\ \textsf{of}\ e_2 \mid e_3 \rightarrow e_2\ v} \qquad \frac{}{\textsf{case inr}_{\tau_1+\tau_2}\ v\ \textsf{of}\ e_2 \mid e_3 \rightarrow e_3\ v}$$

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.

$$\frac{\Gamma \vdash e\!:\!\tau_1}{\Gamma \vdash \textsf{inl}_{\tau_1+\tau_2}\ e\!:\!\tau_1+\tau_2} \qquad \frac{\Gamma \vdash e\!:\!\tau_2}{\Gamma \vdash \textsf{inr}_{\tau_1+\tau_2}\ e\!:\!\tau_1+\tau_2} \qquad \frac{\Gamma \vdash e\!:\!\tau_1+\tau_2 \quad \Gamma \vdash e_1\!:\!\tau_1 \rightarrow \tau \quad \Gamma \vdash e_2\!:\!\tau_2 \rightarrow \tau}{\Gamma \vdash \textsf{case}\ e\ \textsf{of}\ e_1 \mid e_2\!:\!\tau}$$

Let's see an example of a program that uses sum types.

$$\begin{aligned}
&\textsf{let}\ f = \lambda a\!:\!\textbf{int} + (\textbf{int} \rightarrow \textbf{int}).\, \textsf{case}\ a\ \textsf{of}\ (\lambda y.\, y+1) \mid (\lambda g.\, g\ 35)\ \textsf{in}\\
&\textsf{let}\ h = \lambda x\!:\!\textbf{int}.\, x + 7\ \textsf{in}\\
&f\ (\textsf{inr}_{\textbf{int}+(\textbf{int}\rightarrow\textbf{int})}\ h)
\end{aligned}$$

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 fix 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, fix e is the fixed-point of the function e. Note that fix v is *not* a value.

$$e ::= \cdots \mid \textsf{fix}\ e$$

We extend the operational semantics for the new operator. There is a new evaluation context, and a new axiom.

$$E ::= \cdots \mid \textsf{fix}\ E \qquad \frac{}{\textsf{fix}\ \lambda x\!:\!\tau.\, e \rightarrow e\{(\textsf{fix}\ \lambda x\!:\!\tau.\, e)/x\}}$$

Note that we can define the letrec $x\!:\!\tau = e_1$ in e_2 construct in terms of the fix operator.

$$\textsf{letrec}\ x\!:\!\tau = e_1\ \textsf{in}\ e_2 \triangleq \textsf{let}\ x = \textsf{fix}\ \lambda x\!:\!\tau.\, e_1\ \textsf{in}\ e_2$$

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

$$\frac{\Gamma \vdash e\!:\!\tau \rightarrow \tau}{\Gamma \vdash \textsf{fix}\ e\!:\!\tau}$$

Returning to our trusty factorial example, the following program implements the factorial function using the fix operator.

$$\mathsf{FACT} \triangleq \mathsf{fix}\ \lambda f\!:\!\textbf{int} \to \textbf{int}.\ \lambda n\!:\!\textbf{int}.\ \mathsf{if}\ n = 0\ \mathsf{then}\ 0\ \mathsf{else}\ n \times (f\ (n-1))$$

We can write non-terminating computations for any type: the expression fix $\lambda x : \tau.\, x$ has type τ, and does not terminate.

Although the fix operator is normally used to define recursive functions, it can be used to find fixed points of any type. For example, consider the following expression.

$$\begin{aligned}\mathsf{fix}\ \lambda x\!:\!(\textbf{int} \to \textbf{int}) \times (\textbf{int} \to \textbf{int}).\ (&\lambda n\!:\!\textbf{int}.\ \mathsf{if}\ n = 0\ \mathsf{then\ true\ else\ not}\ ((\#2\ x)\ (n-1)),\\ &\lambda n\!:\!\textbf{int}.\ \mathsf{if}\ n = 0\ \mathsf{then\ false\ else\ not}\ ((\#1\ x)\ (n-1)))\end{aligned}$$

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

1.3 References

Recall the syntax and semantics for references.

$$\begin{aligned} e &::= \cdots \mid \mathsf{ref}\ e \mid\ !e \mid e_1 := e_2 \mid \ell\\ v &::= \cdots \mid \ell\\ E &::= \cdots \mid \mathsf{ref}\ E \mid\ !E \mid E := e \mid v := E\end{aligned}$$

$$\textsc{Alloc}\ \frac{}{\langle \mathsf{ref}\ v, \sigma\rangle \longrightarrow \langle \ell, \sigma[\ell \mapsto v]\rangle}\ \ell \notin dom(\sigma) \qquad \textsc{Deref}\ \frac{}{\langle\ !\ell, \sigma\rangle \longrightarrow \langle v, \sigma\rangle}\ \sigma(\ell) = v$$

$$\textsc{Assign}\ \frac{}{\langle \ell := v, \sigma\rangle \longrightarrow \langle v, \sigma[\ell \mapsto v]\rangle}$$

We add a new type for references: type τ **ref** is the type of a location that contains a value of type τ. For example the expression ref 7 has type **int ref**, since it evaluates to a location that contains a value of type **int**. Dereferencing a location of type τ **ref** results in a value of type τ, so $!e$ has type τ if e has type τ **ref**. And for assignment $e_1 := e_2$, if e_1 has type τ **ref**, then e_2 must have type τ.

$$\tau ::= \cdots \mid \tau\ \textbf{ref}$$

$$\frac{\Gamma \vdash e\!:\!\tau}{\Gamma \vdash \mathsf{ref}\ e\!:\!\tau\ \textbf{ref}} \qquad \frac{\Gamma \vdash e\!:\!\tau\ \textbf{ref}}{\Gamma \vdash\ !e\!:\!\tau} \qquad \frac{\Gamma \vdash e_1\!:\!\tau\ \textbf{ref} \quad \Gamma \vdash e_2\!:\!\tau}{\Gamma \vdash e_1 := e_2\!:\!\tau}$$

Noticeable by its absence is a typing rule for location values. What is the type of a location value ℓ? Clearly, it should be of type τ **ref**, where τ is the type of the value contained in location ℓ. But how do we know what value is contained in location ℓ? We could directly examine the store, but that would be inefficient. In addition, examining the store directly may not give us a conclusive answer! Consider, for example, a store σ and location ℓ where $\sigma(\ell) = \ell$; what is the type of ℓ?

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 Σ 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 τ under typing context Γ and store typing Σ.

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.)

$$\frac{\Gamma, \Sigma \vdash e : \tau}{\Gamma, \Sigma \vdash \textsf{ref}\ e : \tau\ \textbf{ref}} \qquad \frac{\Gamma, \Sigma \vdash e : \tau\ \textbf{ref}}{\Gamma, \Sigma \vdash !e : \tau} \qquad \frac{\Gamma, \Sigma \vdash e_1 : \tau\ \textbf{ref} \quad \Gamma, \Sigma \vdash e_2 : \tau}{\Gamma, \Sigma \vdash e_1 := e_2 : \tau} \qquad \frac{}{\Gamma, \Sigma \vdash \ell : \tau\ \textbf{ref}}\ \Sigma(\ell) = \tau$$

So, how do we state type soundness? Our type soundness theorem for simply-typed lambda calculus said that if $\Gamma \vdash e : \tau$ and $e \longrightarrow^* e'$ 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 σ is *well-typed* with respect to typing context Γ and store typing Σ, written $\Gamma, \Sigma \vdash \sigma$, if $dom(\sigma) = dom(\Sigma)$ and for all $\ell \in dom(\sigma)$ 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 $\langle e, \sigma \rangle \longrightarrow^* \langle e', \sigma' \rangle$ then either e' is a value, or there exists e'' and σ'' such that $\langle e', \sigma' \rangle \longrightarrow \langle e'', \sigma'' \rangle$.*

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 ALLOC extends the store σ with a fresh location ℓ, producing store σ'. Since $dom(\Sigma) = dom(\sigma) \neq dom(\sigma')$, it means that we will not have σ' well-typed with respect to typing store Σ.

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 $\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle$ 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 $\ell \in dom(\Sigma)$ we have $\Sigma(\ell) = \Sigma'(\ell)$. This makes sense if we think of partial functions as sets of pairs: $\Sigma \equiv \{(\ell, v) \mid \ell \in dom(\Sigma) \wedge \Sigma(\ell) = v\}$.

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, Σ' will wither be Σ, or Σ extended on a single, newly allocated, location.