CS 4110 – Programming Languages and Logics
Lectures #21: Advanced Types

1 Overview

In this lecture we will extend the simply-typed λ-calculus with several features we saw earlier in the course, including products, sums, and references, as well as one new one.

1.1 Products

We have previously seen how to encode *products* into untyped λ-calculus.

$$\begin{aligned} e &::= \cdots \mid (e_1, e_2) \mid \#1\ e \mid \#2\ e \\ v &::= \cdots \mid (v_1, v_2) \end{aligned}$$

We defined congruence rules that determine the order of evaluation, using the following evaluation contexts.

$$E ::= \cdots \mid (E, e) \mid (v, E) \mid \#1\ E \mid \#2\ E$$

We also defined two computation rules that determin how the pairing constructor and destructors interact.

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

In simply-typed λ-calculus, 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 as follows:

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

Note the similarities between these rules and the proof rules for conjunction in natural deduction. We will examine this relationship closely later in the course.

1.2 Sums

The next example, *sums*, are dual to products. Intuitively, a product holds two values, one of type τ_1, and one of type τ_2, while 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 sums, corresponding to whether we are constructing a sum with a value of τ_1 or a value of τ_2.

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

There are congruence rules that determine the order of evaluation, as defined by the following evaluation contexts.

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

There are also two computation rules that that show how the constructors and destructors interact.

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

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.

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

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

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

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 $\mathbf{int} \to \mathbf{int}$. We destruct 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. In this instance, we end up applying f to a value of type $\mathbf{int} \to \mathbf{int}$ (i.e., a value injected into the right type of the sum), so the entire program ends up evaluating to 42.

1.3 References

Next we consider mutable references. Recall the syntax and semantics for references.

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

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

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

To extend the type system, we add a new type, τ **ref**, to stand for 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.

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

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

Note that there is no typing rule for location values. What should the type of a location value ℓ be? 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 this would not 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.)

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

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 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 $\cdot, \Sigma \vdash e : \tau$ and $\cdot, \Sigma \vdash \sigma$ and $\langle e, \sigma \rangle \rightarrow^* \langle e', \sigma' \rangle$ then either e' is a value, or there exists e'' and σ'' 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: using the preservation and progress lemmas. 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 $\langle e, \sigma \rangle \rightarrow \langle e', \sigma' \rangle$ 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 $\ell \in dom(\Sigma)$ we have $\Sigma(\ell) = \Sigma'(\ell)$. This makes sense if we think of partial functions as sets of pairs: $\Sigma \equiv \{(\ell, v) \mid \ell \in dom(\Sigma) \land \Sigma(\ell) = v\}$. 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 either be Σ, or Σ extended with a newly allocated location.

Interestingly, references are enough to recover Turing completeness. For example, to implement a recursive function f we can initialize a reference cell containing a dummy value for f and then "backpatch" it with the actual definition. For example, here is an implementation of the familiar factorial function, written using **let** expressions, conditionals, and natural numbers for clarity.

$$\begin{array}{l}\textbf{let}\ r = \mathsf{ref}\ \lambda x.\, 0\ \textbf{in}\\ r := \lambda x\!:\!\textbf{int}.\, \textbf{if}\ x = 0\ \textbf{then}\ 1\ \textbf{else}\ x \times\ !r\ (x-1)\end{array}$$

This trick is known as "Landin's knot" after its inventor.

1.4 Fixed Points

Another way to obtain fixed points in the simply-typed lambda calculus is to simply add a new primitive fix to the language. The evaluation rules for the new primitive mimic the behavior of the fixed-point combinators we saw previously.

We extend the syntax with the new primitive operator. Intuitively, $\mathsf{fix}\ e$ is the fixed-point of the function e. Note that $\mathsf{fix}\ v$ is *not* a value.

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

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

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

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

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

The typing rule for fix is left as an exercise.

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

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

Note that we can write non-terminating computations for any type: the expression $\mathsf{fix}\ \lambda x\!:\!\tau.\, x$ 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.

$$\begin{array}{l}\mathsf{fix}\ \lambda x\!:\!(\textbf{int} \to \textbf{int}) \times (\textbf{int} \to \textbf{int}).\, (\lambda n\!:\!\textbf{int}.\ \mathsf{if}\ n = 0\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ (\#2\ x)\ (n-1),\\ \qquad\qquad\qquad\qquad\qquad\qquad \lambda n\!:\!\textbf{int}.\ \mathsf{if}\ n = 0\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ (\#1\ x)\ (n-1))\end{array}$$

This expression defines 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.

1.5 Exceptions

Many programming languages provide support for throwing and catching exceptions. We can model an extremely simple form of exceptions by extending the simply-typed λ-calculus with a single exception representing an error. We first extend the syntax of the language:

$$e ::= \cdots \mid \textbf{error} \mid \textbf{try } e_1 \textbf{ with } e_2$$

We do not add **try** expressions to our evaluation contexts—doing so would allow exceptions to "jump over" handlers. Instead, we add a special propagation rule for **try**:

$$\frac{e_1 \to e_1'}{\textbf{try } e_1 \textbf{ with } e_2 \to \textbf{try } e_1' \textbf{ with } e_2}$$

and rules for propagating and catching exceptions:

$$\frac{}{E[\textbf{error}] \to \textbf{error}} \qquad \frac{}{\textbf{try error with } e \to e} \qquad \frac{}{\textbf{try } v \textbf{ with } e \to v}$$

The typing rule for exceptions allows them to take *any* type, while the typing rule for try-with expressions requires both sub-expressions to have the same type:

$$\frac{}{\Gamma \vdash \textbf{error} : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \textbf{try } e_1 \textbf{ with } e_2 : \tau}$$

The first typing rule is extremely flexible, allowing errors to be thrown anywhere in a program. However, it is not hard to see that it causes the progress lemma to become false: the expression **error** is not a value but is stuck. Fortunately, we can prove the following weaker version, which is still strong enough to prove a useful form of type soundness.

Lemma (Progress)**.** *If $\vdash e : \tau$ then e is a value or e is **error** or there exists e' such that $e \to e'$.*

The preservation theorem remains unchanged.

The actual soundness theorem is as follows:

Theorem 1 (Soundness)**.** *If $\vdash e : \tau$ and $e \to^* e'$ and $e' \not\to$ then either e is a value or e is **error**.*