More types

CS 152 (Spring 2021)

Harvard University

Thursday, March 4, 2021

Today, we will learn about

- typing extensions to the simply-typed lambda-calculus

Products

Syntax:

$$(e_1, e_2)$$

$$\#1\ e$$

$$\#2\ e$$

Operational semantic rules:

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

Typing of Products

Product type: $\tau_1 \times \tau_2$
Typing rules:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \#1\ e : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \#2\ e : \tau_2}$$

Sums

Syntax:

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

$$v ::= \cdots \mid \mathsf{inl}_{\tau_1+\tau_2}\ v \mid \mathsf{inr}_{\tau_1+\tau_2}\ v$$

Context:

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

Operational rules:

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

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

Typing of Sums

Sum type: $\tau_1 + \tau_2$
Typing rules:

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{inl}_{\tau_1+\tau_2}\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{inr}_{\tau_1+\tau_2}\ e : \tau_1 + \tau_2}$$

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

Example Program

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

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 $\mu x\!:\!\tau.\ e$ to the language. The evaluation rules for the new primitive will mimic the behavior of fixed-point combinators.

Recursion: Syntax

$$e ::= \cdots \mid \mu x\!:\!\tau.\ e$$

Intuitively, $\mu x\!:\!\tau.\ e$ is the fixed-point of the function $\lambda x\!:\!\tau.\, e$.
Note that $\mu x\!:\!\tau.\ e$ is *not* a value, regardless of whether e is a value or not.

Recursion: Operational Semantics

There is a new axiom, but no new evaluation contexts.

$$\frac{}{\mu x{:}\tau.\ e \longrightarrow e\{(\mu x{:}\tau.\ e)/x\}}$$

Note that we can define the letrec $x{:}\tau = e_1$ in e_2 construct in terms of this new expression.

$$\text{letrec } x{:}\tau = e_1 \text{ in } e_2 \triangleq \text{let } x{:}\tau = \mu x{:}\tau.\ e_1 \text{ in } e_2$$

Recursion: Typing

$$\frac{\Gamma[x \mapsto \tau] \vdash e : \tau}{\Gamma \vdash \mu x : \tau.\ e : \tau}$$

Example Program

$$FACT \triangleq \mu f : \mathbf{int} \to \mathbf{int}. \\ \lambda n : \mathbf{int}. \text{ if } n = 0 \text{ then } 1 \text{ else } n \times (f\ (n - 1))$$

$$\begin{array}{l} \text{letrec } \mathit{fact} : \mathbf{int} \to \mathbf{int} \\ \ = \lambda n : \mathbf{int}. \text{ if } n = 0 \text{ then } 1 \text{ else } n \times (\mathit{fact}\ (n - 1)) \\ \ \text{in } \ldots \end{array}$$

Non-termination?

Recall operational semantics:

$$\frac{}{\mu x\!:\!\tau.\ e \longrightarrow e\{(\mu x\!:\!\tau.\ e)/x\}}$$

Recall typing:

$$\frac{\Gamma[x \mapsto \tau] \vdash e\!:\!\tau}{\Gamma \vdash \mu x\!:\!\tau.\ e\!:\!\tau}$$

Non-termination

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

Although the $\mu x : \tau.\ e$ 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.

$$\begin{aligned} \mu x : (\textbf{int} \to \textbf{bool}) \times (\textbf{int} \to \textbf{bool}). \\ (\lambda n : \textbf{int}.\ \text{if } n = 0 \text{ then true else } ((\#2\ x)\ (n-1)), \\ \lambda n : \textbf{int}.\ \text{if } n = 0 \text{ then false else } ((\#1\ x)\ (n-1))) \end{aligned}$$

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

References: Syntax and Semantics

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

$$\textsc{Alloc}\ \frac{}{< \mathsf{ref}\ v, \sigma > \longrightarrow < \ell, \sigma[\ell \mapsto v] >}\ \ell \notin \mathsf{dom}(\sigma)$$

$$\textsc{Deref}\ \frac{}{< !\ell, \sigma > \longrightarrow < v, \sigma >}\ \sigma(\ell) = v$$

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

Reference Type τ **ref**

- 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.

References: Typing

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

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathrm{ref}\ e : \tau\ \mathbf{ref}} \qquad \frac{\Gamma \vdash e : \tau\ \mathbf{ref}}{\Gamma \vdash\ !e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau\ \mathbf{ref} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \tau}$$

References: Typing

How do we type locations?

References: Typing

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?

References: Store Typings

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.

References: Typing

$$\frac{\Gamma, \Sigma \vdash e : \tau}{\Gamma, \Sigma \vdash \mathsf{ref}\ e : \tau\ \mathbf{ref}} \qquad \frac{\Gamma, \Sigma \vdash e : \tau\ \mathbf{ref}}{\Gamma, \Sigma \vdash\ !e : \tau}$$

$$\frac{\Gamma, \Sigma \vdash e_1 : \tau\ \mathbf{ref} \quad \Gamma, \Sigma \vdash e_2 : \tau}{\Gamma, \Sigma \vdash e_1 := e_2 : \tau}$$

$$\frac{}{\Gamma, \Sigma \vdash \ell : \tau\ \mathbf{ref}}\ \Sigma(\ell) = \tau$$

References: Soundness?

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.

References: Soundness Aux. Def.

Store σ is *well-typed* with respect to typing context Γ and store typing Σ, written $\Gamma, \Sigma \vdash \sigma$, if $\mathrm{dom}(\sigma) = \mathrm{dom}(\Sigma)$ and for all $\ell \in \mathrm{dom}(\sigma)$ we have $\Gamma, \Sigma \vdash \sigma(\ell) : \tau$ where $\Sigma(\ell) = \tau$.

References: Soundness Theorem

If $\emptyset, \Sigma \vdash e : \tau$ and $\emptyset, \Sigma \vdash \sigma$ and $< e, \sigma > \longrightarrow^* < e', \sigma' >$ then either e' is a value, or there exists e'' and σ'' such that $< e', \sigma' > \longrightarrow < e'', \sigma'' >$.

References: Soundness

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

References: Soundness

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.

References: Preservation Lemma

If $\emptyset, \Sigma \vdash e : \tau$ and $\emptyset, \Sigma \vdash \sigma$ and $< e, \sigma > \longrightarrow < e', \sigma' >$ then there exists some $\Sigma' \supseteq \Sigma$ such that $\emptyset, \Sigma' \vdash e' : \tau$ and $\emptyset, \Sigma' \vdash \sigma'$.