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:

\((e_1, e_2)\)

Operational semantic rules:

\[\#1 \ (v_1, v_2) \rightarrow v_1\]

\[\#2 \ (v_1, v_2) \rightarrow v_2\]
Typing of Products

Product type: $\tau_1 \times \tau_2$

Typing rules:

\[
\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2
\]

\[
\Gamma \vdash e : \tau_1 \times \tau_2 \\
\Gamma \vdash \#1\ e : \tau_1
\]

\[
\Gamma \vdash e : \tau_1 \times \tau_2 \\
\Gamma \vdash \#2\ e : \tau_2
\]
Sums

Syntax:

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

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

Context:

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

Operational rules:

\[ \text{case inl}_{\tau_1 + \tau_2} v \text{ of } e_2 | e_3 \longrightarrow e_2 \ v \]

\[ \text{case inr}_{\tau_1 + \tau_2} v \text{ of } e_2 | e_3 \longrightarrow e_3 \ v \]
Typing of Sums

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

$$
\Gamma \vdash e : \tau_1 \\
\Gamma \vdash \text{inl}_{\tau_1 + \tau_2} e : \tau_1 + \tau_2
$$

$$
\Gamma \vdash e : \tau_2 \\
\Gamma \vdash \text{inr}_{\tau_1 + \tau_2} e : \tau_1 + \tau_2
$$

$$
\Gamma \vdash e : \tau_1 + \tau_2 \\
\Gamma \vdash e_1 : \tau_1 \rightarrow \tau \\
\Gamma \vdash e_2 : \tau_2 \rightarrow \tau \\
\Gamma \vdash \text{case } e \text{ of } e_1 \mid e_2 : \tau
$$
let $f: (\texttt{int} + (\texttt{int} \rightarrow \texttt{int})) \rightarrow \texttt{int} =$

$\lambda a: \texttt{int} + (\texttt{int} \rightarrow \texttt{int})$. 

case $a$ of $\lambda y. y + 1 \mid \lambda g. g 35 \text{ in}$

let $h: \texttt{int} \rightarrow \texttt{int} = \lambda x: \texttt{int}. x + 7$ in

$f \ (\text{inr}_{\texttt{int} + (\texttt{int} \rightarrow \texttt{int})} \ h)$
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.

\[
\mu x : \tau. \ e \rightarrow e\{(\mu x : \tau. \ e)/x\}
\]

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

\[
\Gamma \vdash \mu x : \tau. \; e : \tau
\]

\[
\Gamma \vdash e : \tau
\]
Example Program

\[ FACT \triangleq \mu f : \text{int} \rightarrow \text{int}. \]
\[ \lambda n : \text{int}. \text{if } n = 0 \text{ then } 1 \text{ else } n \times (f (n - 1)) \]

letrec fact : \text{int} \rightarrow \text{int}
\[ = \lambda n : \text{int}. \text{if } n = 0 \text{ then } 1 \text{ else } n \times (\text{fact } (n - 1)) \]
in \ldots
Non-termination?

Recall operational semantics:

\[
\mu x: \tau. \ e \longrightarrow e\{(\mu x: \tau. \ e)/x\}
\]

Recall typing:

\[
\Gamma \vdash e : \tau \\
\Gamma \vdash \mu x: \tau. \ e : \tau
\]
Non-termination

We can write non-terminating computations for any type: the expression $\mu x : \tau. x$ has type $\tau$, 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.

$$\mu x : (\texttt{int} \rightarrow \texttt{bool}) \times (\texttt{int} \rightarrow \texttt{bool}).$$

$$(\lambda n : \texttt{int}. \text{if } n = 0 \text{ then true else } ((\#2 x) \ (n - 1)), \lambda n : \texttt{int}. \text{if } n = 0 \text{ then false else } ((\#1 x) \ (n - 1)))$$

This expression has type $(\texttt{int} \rightarrow \texttt{bool}) \times (\texttt{int} \rightarrow \texttt{bool})$—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

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

\[
v ::= \cdots | \ell
\]

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

\[
\text{ALLOC} \quad \frac{\quad}{\quad} \quad \ell \notin \text{dom}(\sigma)
\]

\[
< \text{ref } v, \sigma > \quad \quad \quad \rightarrow \quad \quad \quad < \ell, \sigma[\ell \mapsto v] >
\]

\[
\text{DEREF} \quad \frac{\quad}{\quad} \quad \sigma(\ell) = v
\]

\[
< !\ell, \sigma > \quad \quad \quad \rightarrow \quad \quad \quad < v, \sigma >
\]

\[
\text{ASSIGN} \quad \frac{\quad}{\quad} \quad \lambda \notin \text{dom}(\sigma)
\]

\[
< \ell := v, \sigma > \quad \quad \rightarrow \quad \quad \quad < v, \sigma[\ell \mapsto v] >
\]

16 / 28
Reference Type $\tau \text{ ref}$

- 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 ref 7 has type $\text{int ref}$, 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 $\tau$. 
References: Typing

\[ \tau ::= \cdots \mid \tau \text{ ref} \]

\[ \Gamma \vdash e : \tau \]

\[ \Gamma \vdash \text{ref } e : \tau \text{ ref} \]

\[ \Gamma \vdash e : \tau \text{ ref} \]

\[ \Gamma \vdash !e : \tau \]

\[ \Gamma \vdash e_1 : \tau \text{ ref} \]

\[ \Gamma \vdash e_2 : \tau \]

\[ \Gamma \vdash e_1 := e_2 : \tau \]
References: Typing

How do we type locations?
Noticeable 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, examining 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 $\Sigma$. 
References: Typing

\[
\begin{align*}
\Gamma, \Sigma \vdash e : \tau & \quad \Gamma, \Sigma \vdash e : \tau \text{ ref} \\
\Gamma, \Sigma \vdash \text{ref } e : \tau \text{ ref} & \quad \Gamma, \Sigma \vdash !e : \tau \\
\Gamma, \Sigma \vdash e_1 : \tau \text{ ref} & \quad \Gamma, \Sigma \vdash e_2 : \tau \\
\Gamma, \Sigma \vdash e_1 := e_2 : \tau & \\
\Gamma, \Sigma \vdash \ell : \tau \text{ ref} & \quad \Sigma(\ell) = \tau
\end{align*}
\]
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.
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 $\Gamma, \Sigma \vdash \sigma(\ell) : \tau$ where $\Sigma(\ell) = \tau$. 
If $\emptyset, \Sigma \vdash e : \tau$ and $\emptyset, \Sigma \vdash \sigma$ and
$\langle e, \sigma \rangle \rightarrow^* \langle e', \sigma' \rangle$ then either $e'$ is a value, or
there exists $e''$ and $\sigma''$ such that
$\langle e', \sigma' \rangle \rightarrow \langle e'', \sigma'' \rangle$.  

References: Soundness Theorem
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 \texttt{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.
If $\emptyset, \Sigma \vdash e : \tau$ and $\emptyset, \Sigma \vdash \sigma$ and $< e, \sigma > \rightarrow < e', \sigma'>$ then there exists some $\Sigma' \supseteq \Sigma$ such that $\emptyset, \Sigma' \vdash e' : \tau$ and $\emptyset, \Sigma' \vdash \sigma'$.