Simply-Typed $\lambda$-Calculus

CS 565
Lecture 10
3/03/06
Syntax: $F_1$

Terms
\[ e ::= x \mid \lambda \ x: \tau.e \mid e_1 \; e_2 \]
\[ \mid n \mid e_1 + e_2 \mid \text{iszero} \; e \]
\[ \mid \text{true} \mid \text{false} \mid \text{not} \; e \]
\[ \mid \text{if} \; e_1 \; \text{then} \; e_2 \; \text{else} \; e_3 \]

Types
\[ \tau ::= \text{int} \mid \text{bool} \mid \tau_1 \to \tau_2 \]

→ is a function type constructor and associates to the right

Formal arguments to functions have typing annotations
Static Semantics

\(\Box\) The typing judgment

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

\(\Box\) Typing rules

\[\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\]

\[\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau . e : \tau \rightarrow \tau'}\]

\[\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 \, e_2 : \tau_2}\]
Static Semantics (cont)

- More typing rules

\[ \Gamma \vdash n : \text{int} \quad \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \]
\[ \frac{}{\Gamma \vdash e_1 + e_2 : \text{int}} \]

\[ \Gamma \vdash \text{true} : \text{bool} \]
\[ \Gamma \vdash e : \text{bool} \]
\[ \frac{}{\Gamma \vdash \text{not } e : \text{bool}} \]

\[ \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau, \Gamma \vdash e_3 : \tau \]
\[ \frac{}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \]
Typing Derivations in $\mathbb{F}_1$

Consider the term:

$$\lambda \ x: \text{int}. \ \lambda \ b: \text{bool}. \ \text{if} \ b \ \text{then} \ f \ x \ \text{else} \ x$$

in a typing environment that maps $f$ to $\text{int} \rightarrow \text{int}$

$$\Gamma \vdash f : \text{int} \rightarrow \text{int}, \Gamma \vdash x : \text{int}$$

$$\Gamma \vdash b : \text{bool} \quad \Gamma \vdash f \ x : \text{int} \quad \Gamma \vdash x : \text{int}$$

$$\Gamma \vdash \text{if} \ b \ \text{then} \ f \ x \ \text{else} \ x : \text{int}$$

$$\Gamma \vdash \lambda \ b : \text{bool}. \ \text{if} \ b \ \text{then} \ f \ x \ \text{else} \ x : \text{bool} \rightarrow \text{int}$$

$$\Gamma = f : \text{int} \rightarrow \text{int}, \ x : \text{int}, \ b : \text{bool}$$
Type Checking

- Syntax-directed
  - Derivations follow syntactic structure of the program
- Compositional
  - Understand types of terms as being built from types of subterms
- Annotated
  - All formal parameters are annotated with types
  - No need to infer types (although this wouldn’t be so hard)
- Without annotations, expressions need not have a unique type
  - $\Gamma \vdash \lambda x. x : \text{int} \to \text{int}$
  - $\Gamma \vdash \lambda x. x : \text{bool} \to \text{bool}$
Typability

- We may erase types from expressions systematically:
  - erase(x) = x
  - erase(e₁ e₂) = erase(e₁) erase(e₂)
  - erase(λ x:τ.e) = λ x. erase(e)

- Is an untyped expression typable (with respect to a given type environment)?
  - Given e, does there exist e’ and τ such that erase(e’) = e and Γ ⊢ e’ : τ
  - λ x. x is typable in the empty environment
  - There is an infinite collections of typings for this term
Operational Semantics of $F_1$

- Evaluation relation
  - $e \Downarrow v$

- Values
  - $v ::= n \mid \text{true} \mid \text{false} \mid \lambda x: \tau. e$

- Call-by-value evaluation rules
  - $\lambda x: \tau. e \Downarrow \lambda x: \tau. e$

\[
\begin{align*}
  e_1 & \Downarrow \lambda x: \tau.e'_1 \quad e_2 \Downarrow v_2 \quad [v_2/x]e'_1 \Downarrow v \\
  e_2 & \Downarrow v
\end{align*}
\]
Operational Semantics (cont)

More rules:

\[ \begin{align*}
    n & \downarrow n \\
    e_1 & \downarrow n_1 \\
    e_2 & \downarrow n_2 \\
    n & = n_1 + n_2
\end{align*} \]

\[ \begin{align*}
    e_1 & \downarrow n_1 \\
    e_2 & \downarrow n_2 \\
    e_1 + e_2 & \downarrow n
\end{align*} \]

\[ \begin{align*}
    e_1 & \downarrow \text{true} \\
    e_1 & \downarrow \text{false}
\end{align*} \]

\[ \begin{align*}
    e_1 & \downarrow \text{true} \\
    e_1 & \downarrow \text{false}
\end{align*} \]

\[ \begin{align*}
    e_1 + e_2 & \downarrow n
\end{align*} \]

\[ \begin{align*}
    \text{if } e_1 \text{ then } e_1 + e_2 \text{ else } e_1 + e_2 & \downarrow n
\end{align*} \]
Type Soundness for $F_1$

- **Theorem**
  - If $\vdash e : \tau$ and $e \downarrow v$ then $\vdash v : \tau$
  - Known as the subject reduction or type preservation theorem

- **Can we prove this theorem by induction on $e$?**
  - No, because we have $[v_2/x]e_1'$ in the evaluation of $e_1 e_2$

- **Can we prove this theorem by induction on $\tau$?**
  - No, because $e_1$ has a “bigger” type than $e_1 e_2$

- **Can we prove this by induction on $e \downarrow v$?**
  - Yes; it addresses the issue of $[v_2/x]e_1'$
  - Could also use induction on typing derivations
Soundness

Consider the case:

\[ e_1 \Downarrow \lambda x : \tau_2.e'_1, \ e_2 \Downarrow v_2, \ [v_2/x]e'_1 \Downarrow v \]
\[ e_1 e_2 \Downarrow v \]

By derivation of \( e_1 e_2 : \tau \) we know

\[ \vdash e_1 : \tau_2 \rightarrow \tau, \ \vdash e_2 : \tau_2 \]
\[ \vdash e_1 e_2 : \tau \]

From IH on \( e_1 \Downarrow ... \), we have \( x : \tau_2 \vdash e'_1 : \tau \)

From IH on \( e_2 \Downarrow ... \) we have \( \vdash v_2 : \tau_2 \)

Need to infer that \( \vdash [v_2/x]e'_1 : \tau \) and use the IH

Need a substitution lemma
Substitution Lemma

If \( \Gamma, x : \tau \vdash e : \tau' \) and \( \Gamma \vdash y : \tau \), then

\[ \Gamma \vdash [ y/x ]e : \tau' \]

Proof: by induction on the derivation of

\[ \Gamma, x : \tau \vdash e : \tau' \]

(see page 106 and 107 of the text)
Significance of Type Soundness

- The theorem says that the result of an evaluation has the same type as the initial expression.
- The theorem does not say that:
  - evaluation never gets stuck (it is not a progress theorem)
  - evaluation terminates
Small-step contextual semantics

- Define redexes:
  \[ r ::= n_1 + n_2 \mid \text{if } v \text{ then } e_1 \text{ else } e_2 \mid (\lambda x : \tau . e_1) v_2 \]

- Define contexts:
  \[ E ::= [] \mid E + e_2 \mid n_1 + E \mid \text{if } E \text{ then } e_1 \text{ else } e_2 \mid E e_2 \mid (\lambda x : \tau . e_1) E \]

- Local reduction rules:
  \[ n_1 + n_2 \rightarrow n_1 \oplus n_2 \]
  \[ \text{if true then } e_1 \text{ else } e_2 \rightarrow e_1 \]
  \[ \text{if false then } e_1 \text{ else } e_2 \rightarrow e_2 \]
  \[ (\lambda x : \tau . e_1) v_2 \rightarrow [v_2/x]e_1 \]

- Global reduction rule:
  \[ E[r] \rightarrow E[e] \text{ iff } r \rightarrow e \]
Contextual Semantics

Decomposition Lemma:

1. If $\vdash e : \tau$ and $e$ is not a value then there exists a unique $E$ and $r$ such that $e = E[r]$
   - Any well-typed term can be decomposed
   - Any well-typed non-value can make progress
2. Furthermore, there exists $\tau'$ such that $\vdash r : \tau'$
   - All redexes are well-typed
3. Furthermore, there exists $e'$ such that $r \rightarrow e'$ and $\vdash e' : \tau'$
   - Local reductions are type-preserving
4. Furthermore, for any $e'$, $\vdash e' : \tau'$ implies $\vdash E[e'] : \tau$
   - An expression keeps its type if we replace a redex by an expression of the same type
Contextual Semantics

☐ Type preservation theorem:
  ■ If \( \vdash e : \tau \) and \( e \rightarrow e' \) then \( \vdash e' : \tau \)
  ■ Follows from the decomposition lemma

☐ Progress theorem:
  ■ If \( \vdash e : \tau \) and \( e \) is not a value then there exists \( e' \) such that \( e \) can make progress: \( e \rightarrow e' \)
  ■ The progress theorem says that execution can make progress on well-typed expressions.
  ■ Furthermore, because of type preservation, we know that the execution of a well-typed expression never gets stuck.
Alternative Semantics

☐ Can obtain similar results from big-step or denotational semantics.

☐ To do so, introduce a special error value called “wrong”
  ■ Wrong does not have a typing rule
  ■ Evaluation rules yield wrong for terms that ordinarily would be stuck

☐ Main theorem: “well-typed programs don’t go wrong”