Simply-Typed λ-Calculus Extensions

CS 565
Lecture 11
3/20/06
Static Semantics for

Product Types: $F_1^X$

Extend the semantics with (binary) tuples:

$e ::= \ldots | (e_1, e_2) | \text{fst } e | \text{snd } e$

$\tau ::= \ldots | \tau_1 \times \tau_2$

Same typing judgment $\Gamma \vdash e : \tau$

$\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2$

$\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2$

$\Gamma \vdash e : \tau_1 \times \tau_2$

$\Gamma \vdash \text{fst } e : \tau_1$

$\Gamma \vdash \text{snd } e : \tau_2$
Dynamic Semantics for Product Types

- New form of values:
  \( v ::= \ldots \mid (v_1, v_2) \)

- New (big-step) evaluation rules:
  \[
  \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{(e_1,e_2) \Downarrow (v_1,v_2)}
  \]

\[
\frac{e \Downarrow (v_1,v_2)}{\text{fst } e \Downarrow v_1}
\]

\[
\frac{e \Downarrow (v_1,v_2)}{\text{snd } e \Downarrow v_2}
\]
Contextual Semantics

- New contexts:
  \[ E ::= [] \mid ... \mid (E, e) \mid (v, E) \mid \text{fst } E \mid \text{snd } E \]

- New redexes:
  \[ \text{fst } (v_1, v_2) \rightarrow v_1 \]
  \[ \text{snd } (v_1, v_2) \rightarrow v_2 \]

- Type soundness as before
Records

- Records are like tuples with labels
- New form of expressions:
  \[ e ::= \ldots \mid \{ L_1 = e_1, \ldots, L_n = e_n \} \mid e \cdot L \]
- New form of values:
  \[ v ::= \{ L_1 = v_1, \ldots, L_n = v_n \} \]
- New form of types:
  \[ \tau ::= \ldots \mid \{ L_1 : \tau_1, \ldots, L_n : \tau_n \} \]
- Follows the same basic structure as \( F_1^\times \)
Sum Types (System F₁⁺)

- Consider types of the form:
  - either an int or a bool
  - either int or a pointer
  - either a function or an int
- These types are called disjoint union types
- Introduce new form of expressions and types:
  \[ e ::= \ldots \mid \text{injl } e \mid \text{injr } e \mid \text{case } e \text{ of } \text{injl } x \Rightarrow e_1 \mid \text{injr } y \Rightarrow e_2 \]
  \[ \tau ::= \ldots \mid \tau_1 + \tau_2 \]
  - A value of type \( \tau_1 + \tau_2 \) is either a \( \tau_1 \) or a \( \tau_2 \)
  - Similar to unions in C or Pascal, but safe.
- Distinguishing between components is under compiler control
- Case is a binding operator: \( x \) is bound in \( e_1 \) and \( y \) is bound in \( e_2 \)
Examples

- Consider the type "unit" with a single element called "*"
- The type optional integer defined as "unit + int"
  - Similar to option types in ML
  - No argument: injl *
  - Argument is 5: injr 5
- To use the argument, must test its kind:
  ```
  case arg of
    injl * → 0
    injr y → y + 3
  ```
- injl and injr are value-carrying tags and case does tag checking
Sum Types

☐ Can think of bool as a unit type:

- bool = unit + unit
- true = injl *
- false = injr *
- if e then e₁ else e₂ is the same as
  case e of
    injl * ⇒ e₁
    injr * ⇒ e₂
Typing Rules for Sum Types

\[ \Gamma \vdash e : \tau_1 \quad \Gamma \vdash e : \tau_2 \]
\[ \Gamma \vdash \text{injl } e : \tau_1 + \tau_2 \quad \Gamma \vdash \text{injr } e : \tau_1 + \tau_2 \]

\[ \Gamma \vdash e_1 : \tau_1 + \tau_2 \quad \Gamma, x : \tau_1 \vdash e_1 : \tau \]
\[ \Gamma, y : \tau_2 \vdash e_2 : \tau \]
\[ \Gamma \vdash \text{case } e_1 \text{ of injl } x \Rightarrow e_1 \mid \text{injr } y \Rightarrow e_2 : \tau \]
Dynamic Semantics

- New values:
  \[ v :: = \ldots \mid \text{injl} \ v \mid \text{injr} \ v \]

- New evaluation rules:

  \[
  \frac{e \Downarrow v}{\text{injl} \ e \Downarrow \text{injl} \ v} \quad \quad \frac{e \Downarrow v}{\text{injr} \ e \Downarrow v}
  \]

  \[
  \frac{e \Downarrow \text{injl} \ v \quad [v/x]e_1 \Downarrow v'}{\text{case} \ e \ \text{of injl} \ x \Rightarrow e_1 \mid \text{injr} \ y \Rightarrow e_r \Downarrow v'}
  \]

  \[
  \frac{e \Downarrow \text{injr} \ v \quad [v/y]e_r \Downarrow v'}{\text{case} \ e \ \text{of injl} \ x \Rightarrow e_1 \mid \text{injr} \ y \Rightarrow e_r \Downarrow v'}
  \]
Type Soundness for $F_1^+$

- Type soundness still holds
- There is no way to use a value of $\tau_1 + \tau_2$ inappropriately
- The key is that the only way to use a value of $\tau_1 + \tau_2$ is with case which ensures that one does not interchange a $\tau_1$ for a $\tau_2$
- In C or Pascal, proper tag checking is the responsibility of the programmer (unsafe)
Uniqueness

- If \( e \) has type \( \tau \) then \( \text{injl} \ e \) has type \( \tau + \tau' \) for any \( \tau' \)
- Possible solutions:
  - “infer” \( \tau' \) as needed during typechecking
  - give constructors different names and only allow each name to appear in one sum type (this is the solution adopted in SML and OCaml)
  - Annotate each \( \text{injl} \) and \( \text{injr} \) with the intended sum type
Annotations

New syntactic forms:
\[ e ::= \ldots | \text{injl } e \text{ as } \tau | \text{injr } e \text{ as } \tau | \]
\[ \text{case } e \text{ of } \text{injl } x \text{ as } \tau \Rightarrow e_1 \]
\[ | \text{injr } y \text{ as } \tau \Rightarrow e_2 \]
\[ \tau ::= \ldots | \tau_1 + \tau_2 \]

New typing rules:
\[
\begin{align*}
\Gamma \vdash e : \tau_1 & \quad \Gamma \vdash e : \tau_2 \\
\Gamma \vdash \text{injl } e \text{ as } \tau_1 + \tau_2 : \tau_1 + \tau_2 & \quad \Gamma \vdash \text{injr } e \text{ as } \tau_1 + \tau_2 : \tau_1 + \tau_2
\end{align*}
\]
Recursion

- In $F_1$, all programs terminate
- An untyped term like:
  $$\lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y))$$
  which defines a fixpoint functional can’t be typed using simple types
  - How would we type $x$?
    - $x$ has type $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$
    - it also has type $\tau_1$
  - Extend the calculus with a typed fixpoint operator
Example

- \( ff = \lambda \text{isEven}: \text{int} \rightarrow \text{bool} \).
  \[
  \lambda x. \text{int}
  \]
  if (iszero x)
    then true
  else if iszero (pred x)
    then false
  else isEven (pred (pred x))

- iseven = fix \( ff \)
- \( ff \) has type \( (\text{int} \rightarrow \text{bool}) \rightarrow \text{int} \rightarrow \text{bool} \)
- \( \text{fix } ff \) has type \( (\text{int} \rightarrow \text{bool}) \)
- \( (\text{fix } ff) \) defines a fixpoint for \( ff \)
Typing and Evaluation Rules

\[ e ::= \ldots \mid \text{fix } e \]

\[
\begin{align*}
\Gamma \vdash e : \tau_1 \rightarrow \tau_2 \\
\Gamma \vdash \text{fix } e : \tau_1
\end{align*}
\]

\[
\begin{align*}
e \Downarrow \lambda x:\tau.e' \\
\text{fix } e \Downarrow [(\text{fix } \lambda x:\tau.e')/x]e'
\end{align*}
\]
Normalization

- In the pure simply-typed $\lambda$ calculus, every well-typed program is guaranteed to halt in a finite number of steps.
- The language of interest does not include support for recursion.
- Non-trivial to prove since each reduction of a term can duplicate redexes e.g.,
  \[ \lambda x. (f \times x) \]
Proof Technique

- Consider a calculus with a single base type $A$.
- Define for each type $\tau$ built using $\to$ and $A$, the set of closed terms that have type $\tau$
  - $R_A(e)$ iff $e$ halts
  - $R_{\tau_1 \to \tau_2}(e)$ iff $e$ halts and whenever $R_{\tau_1}(e_1)$ we have $R_{\tau_2}(e, e_1)$
Normalization

- Main complexity deals with function types:
  - application of a normalized function to a normalized argument defined in terms of a substitution
  - evaluation of an expression of function type must halt
  - evaluation of an expression of function type when applied to a halting argument must halt.
Logical Relations

- To prove a property $P$ of all closed terms of type $A$
  - Proceed to prove by induction on types that all terms of type $A \rightarrow A$ possess property $P$, all terms of type $(A \rightarrow A) \rightarrow (A \rightarrow A)$ possess property $A$, and so forth.
  - Define a family of predicates indexed by types:
    - for function types, the predicate requires the input type to satisfy the predicate and that the type of the value output satisfy the predicate
Lemmas

- (1) If $R_\tau(e)$ then $e$ halts
  - obvious from definition of $R$
- (2) If $e:\tau$ and $e \rightarrow e'$ then $R_\tau(e)$ iff $R_\tau(e')$
  - by induction on the structure of $\tau$
  - Need to handle the case when $\tau = \tau_1 \rightarrow \tau_2$
  - If $R_\tau(e)$ and $R_{\tau_1}(s)$, then $R_{\tau_2}(e\ s)$. We know $e\ s \rightarrow e'\ s$, so by IH $R_{\tau_2}(e'\ s)$ holds
- (3) Every term of type $\tau$ belongs to $R_\tau$.
  - by induction on typing derivations
  - for $\lambda \ x:\tau. e \in R_{\tau \rightarrow \tau'}$ we must apply the induction hypothesis to show that $e \in R_\tau$
  - But, $R_\tau$ is only defined for closed terms, but $e$ may have free variables (e.g., $x$)
  - Trick is to generalize induction hypothesis to cover all closed instances of an open term $\tau$. 
Lemmas

☐ (4) If $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e_n : \tau$ and $v_1, \ldots, v_n$ are closed values of types $\tau_1, \ldots, \tau_n$ such that $R_{\tau_i}(v_i)$ for each $i$, then $R_T([v_1/x_1, \ldots, v_n/x_n] \, \tau)$

☐ see page 151 in TAPL

☐ Theorem: If $\vdash e : \tau$ then $e$ is normalizable.

☐ Proof: $R_\tau(e)$ by above lemma and thus $e$ is normalizable by Lemma 1