CS6848 - Principles of Programming Languages

Principles of Programming Languages

V. Krishna Nandivada

IIT Madras

Recap

- Structural subtyping
- Unification algorithm

Recall

$$
\begin{array}{lcl}
e & ::= & x \mid \lambda x.e \mid e_1 e_2 \mid c \mid succ\ e \\
x & \in & \text{Identifier (infinite set of variables)} \\
c & \in & \text{Integer} \\
v & ::= & c | \lambda x.e \\
t & ::= & \text{Int } | t \rightarrow t
\end{array}
$$

Extending a language

- Extend the language grammar that will lead to new terms.
- Extend the allowed values.
- Extend the types.
- New operational semantics.
- New typing rules.

Pairs

- Expressions
$$e ::= \cdots |(e_1, e_2)|e.1|e.2$$
- Values
$$v ::= \cdots |(v_1, v_2)$$
- Types
$$t ::= \cdots |t_1 \times t_2$$

New operational semantics

- First element:
$$(Pair\ \beta 1)(v_1, v_2).1 \to v_1$$
- Second element:
$$(Pair\ \beta 2)(v_1, v_2).2 \to v_2$$
- $$\text{Projection 1}\ \frac{e \to e'}{e.1 \to e'.1}$$
- $$\text{Projection 2}\ \frac{e \to e'}{e.2 \to e'.2}$$
- $$\text{Pair Evaluation 1}\ \frac{e_1 \to e_1'}{(e_1, e_2) \to (e_1', e_2)}$$
- $$\text{Pair Evaluation 2}\ \frac{e_2 \to e_2'}{(v_1, e_2) \to (v_1, e_2')}$$

Typing rules for pairs

- $$\text{Pair}\ \frac{A \vdash e_1 : t_1 \quad A \vdash e_2 : t_2}{A \vdash (e_1, e_2) : t_1 \times t_2}$$
- $$\text{Projection 1}\ \frac{A \vdash e : t_1 \times t_2}{A \vdash e.1 : t_1}$$
- $$\text{Projection 2}\ \frac{A \vdash e : t_1 \times t_2}{A \vdash e.2 : t_2}$$

Properties of pairs

- The components are evaluated left to right.
- The pair must be fully evaluated to get the components.
- A pair that is passed as an argument will be fully evaluated, before the function starts executing(in call by value semantics).

Tuples

- Expressions

$$e ::= \cdots | (e_i^{i \in 1..n}) | e.i$$

- Values

$$v ::= \cdots | (v_i^{i \in 1..n})$$

- Types

$$t ::= \cdots | (t_i^{i \in 1..n})$$

New operational semantics

- Element j:

$$(\beta)(v_i^{i \in 1..n}).j \rightarrow v_j$$

-

$$\text{Projection 1}\ \frac{e \rightarrow e'}{e.i \rightarrow e'.i}$$

-

$$\text{Tuple Evaluation}\ \frac{e_j \rightarrow e'_j}{(v_i^{i \in 1..j-1}, e_j, e_k^{k \in j+1..n}) \rightarrow (v_i^{i \in 1..j-1}, e'_j, e_k^{k \in j+1..n})}$$

Typing rules for tuples

-

$$\text{Tuple}\ \frac{A \vdash \forall i\ e_i : t_i}{A \vdash (e_i^{i \in 1..n}) : (t_i^{i \in 1..n})}$$

-

$$\text{Projection}\ \frac{A \vdash e : (t^{i \in 1..n})}{A \vdash e.j : t_j}$$

Records

- Expressions

$$e ::= \cdots | (l_i = e_i^{i \in 1..n}) | e.l$$

- Values

$$v ::= \cdots | (l_i = v_i^{i \in 1..n})$$

- Types

$$t ::= \cdots | (l_i : t_i^{i \in 1..n})$$

New operational semantics

- Element j:
$$(\beta)(l_i : v_i^{i\in 1..n}).l_j \rightarrow v_j$$
- Projection 1 $$\frac{e \rightarrow e'}{e.l \rightarrow e'.l}$$
- Record Evaluation $$\frac{e_j \rightarrow e'_j}{\begin{array}{l}(l_i = v_i^{i\in 1..j-1}, l_j = e_j, l_k = e_k^{k\in j+1..n}) \rightarrow \\ (l_i = v_i^{i\in 1..j-1}, l_j = e'_j, l_k = e_k^{k\in j+1..n})\end{array}}$$

Typing rules for tuples

- Tuple $$\frac{A \vdash \forall i\ e_i : t_i}{A \vdash (l_i = e_i^{i\in 1..n}) : (l_i : t_i^{i\in 1..n})}$$
- Projection $$\frac{A \vdash e : (l_i : t^{i\in 1..n})}{A \vdash e.l_j : t_j}$$

Polymorphism

- AppTwiceInt = $\lambda f.$Int $\rightarrow$ Int $.\lambda x :$ Int $.f\ (f\ x)$
  AppTwiceRcd = $\lambda f.(l :$ Int $) \rightarrow (l :$ Int $).\lambda x : (l :$ Int $).f\ (f\ x)$
  AppTwiceOther =
  $\lambda f.$(Int $\rightarrow$ Int) $\rightarrow$ (Int $\rightarrow$ Int)$.\lambda x :$ (Int $\rightarrow$ Int)$.f\ (f\ x)$
- Each significant piece of functionality in a program should be implemented in just one place in the source code.
- Type systems allow single piece of code to be used with multiple types are collectively known as polymorphic systems.
  - Parametric polymorphism: Single piece of code to be typed generically.
    - Example: Java generics, C++ templates.
    - Uses variables in places of actual types and may instantiate with actual types if needed.
  - Ad-hoc polymorphism - allows a polymorphic value to exhibit different behaviors when viewed using different types.
    - Example: Overloading
  - subtype polymorphism: A single term may get many types using subsumption.