CS6848 - Principles of Programming Languages

Principles of Programming Languages

V. Krishna Nandivada

IIT Madras

Recap

- Structural subtyping
- Unification algorithm

Recall

e	::=	$x \mid \lambda x.e \mid e_1 e_2 \mid c \mid succ\ e$
x	$\in$	Identifier (infinite set of variables)
c	$\in$	Integer
v	:: =	$c \mid \lambda x.e$
t	:: =	Int $\mid t \rightarrow t$

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

Typing rules for pairs

- Pair $$\frac{A \vdash e_1 : t_1 \qquad A \vdash e_2 : t_2}{A \vdash (e_1, e_2) : t_1 \times t_2}$$
- Projection 1 $$\frac{A \vdash e : t_1 \times t_2}{A \vdash e.1 : t_1}$$
- 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
- Values
- Types

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

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

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

New operational semantics

- Element j:

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

- Projection 1 $\dfrac{e \rightarrow e'}{e.i \rightarrow e'.i}$

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

Typing rules for tuples

- Tuple $\dfrac{A \vdash \forall i \; e_i : t_i}{A \vdash (e_i^{i \in 1..n}) : (t_i^{i \in 1..n})}$

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

Records

- Expressions
- Values
- Types

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

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

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

New operational semantics

- Element j:

$$\beta \text{ reduction} \quad (l_i : v_i^{i \in 1..n}).l_j \rightarrow v_j$$

- $$\text{Projection 1} \quad \frac{e \rightarrow e'}{e.l \rightarrow e'.l}$$

- $$\text{Record Evaluation} \quad \frac{e_j \rightarrow e'_j}{(l_i = v_i^{i \in 1..j-1}, l_j = e_j, l_k = e_k^{k \in j+1..n}) \rightarrow (l_i = v_i^{i \in 1..j-1}, l_j = e'_j, l_k = e_k^{k \in j+1..n})}$$

Typing rules for tuples

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

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

Polymorphism - motivation

- AppTwiceInt = λf : Int $\rightarrow$ Int .λx : Int .f (f x)
  AppTwiceRcd = λf : (l : Int) $\rightarrow$ (l : Int).λx : (l : Int).f (f x)
  AppTwiceOther =
  λf : (Int $\rightarrow$ Int) $\rightarrow$ (Int $\rightarrow$ Int).λx : (Int $\rightarrow$ Int).f (f x)
- Breaks the idea of abstraction: Each significant piece of functionality in a program should be implemented in just one place in the source code.

Polymorphism - variations

- Type systems allow single piece of code to be used with multiple types are collectively known as *polymorphic* systems.
- Variations:
  - Parametric polymorphism: Single piece of code to be typed generically (also known as, let polymorphism, first-class polymorphism, or ML-style polymorphic).
    - Restricts polymorphism to top-level `let` bindings.
    - Disallows functions from taking polymorphic values as arguments.
    - Uses variables in places of actual types and may instantiate with actual types if needed.
    - Example: ML, Java Generics

```
(let ((apply  lambda f. lambda a (f a)))
  (let ((a (apply succ 3)))
    (let ((b (apply zero? 3))) ...
```

  - Ad-hoc polymorphism - allows a polymorphic value to exhibit different behaviors when viewed using different types.
    - Example: function Overloading, Java `instanceof` operator.
  - subtype polymorphism: A single term may get many types using subsumption.

polymorphism.