CS6848 - Principles of Programming Languages

Principles of Programming Languages

V. Krishna Nandivada

IIT Madras

Types are Ubiquitous

- Q: Write a function to print an Array of integers?

```
void printArr(int A[]){
    for (int i=0;i<A.length;++i){
        System.out.println(A[i]);
    }
}
```

What is a Type?

- A type is an invariant.
- For example, in Java

```
int v;
```

  specifies that `v` may only contain integer values in a certain range.
- Invariant on what?
- About what?

Why Types?

Advantages with programs with types – three (tall?) claims:

- Readable : Types provide documentation;
  "Well-typed programs are more readable".

  Example: bool equal(String s1, String s2);
- Efficient: Types enable optimizations;
  "Well-typed programs are faster".

  Example: c = a + b
- Reliable: Types provide a safety guarantee;
  "Well-typed programs cannot go wrong".

Programs with no-type information can be **unreadable, inefficient, and unreliable.**

Example language

- A λ-calculus.
- Admits only two kinds of data: integers and functions.
- Grammar of the language:

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

Simply typed lambda calculus

- Types: integer types and function types.
- Grammar for the types:

$$\tau ::= \text{Int} | \tau_1 \rightarrow \tau_2$$

- Extend the signature of a lambda: $\lambda x : \tau.e$ – every function specifies the type of its argument.
- Examples: $\begin{cases} 0 & : \text{Int} \\ \lambda x : \text{Int}\ .(succ\ x) & : \text{Int} \rightarrow \text{Int} \\ \lambda x : \text{Int}\ .succ\ \lambda y : \text{Int}\ .x + y & : \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \end{cases}$
- These are simple types - each type can be viewed as a finite tree. ~~polymorphic types, dependent types~~
- Infinitely many types.

Type environment

- Type environment A : $Var \rightarrow types$.
- A type environment is a partial function which maps variables to types.
- ϕ denotes the type environment with empty domain.
- Extending an environment A with (x,t) - given by $A[x:t]$
- Application - $A(y)$ gives the type of the variable y.
- Type Evaluation: $A \vdash e : t$ — e has type t in environment A.
- Q: How to do type evaluation?

Type rules

- The judgement $A \vdash e : t$ holds, when it is derivable by a finite derivation tree using the following type rules.

$$A \vdash x : t (A(x) = t) \quad (1)$$

$$\frac{A[x:s] \vdash e : t}{A \vdash \lambda x : s.e : s \rightarrow t} \quad (2)$$

$$\frac{A \vdash e_1 : s \rightarrow t,\ A \vdash e_2 : s}{A \vdash e_1 e_2 : t} \quad (3)$$

$$A \vdash 0 : \text{Int} \quad (4)$$

$$\frac{A \vdash e : \text{Int}}{A \vdash \text{succ}\ e : \text{Int}} \quad (5)$$

- Exactly one rule for each construct in the language. Also note the axioms
- An expression e is well typed if there exist A, t such that $A \vdash e : t$ is derivable.

Type rules

$$A \vdash x : t (A(x) = t) \quad (1)$$

$$\frac{A[x:s] \vdash e : t}{A \vdash \lambda x : s.e : s \to t} \quad (2)$$

$$\frac{A \vdash e_1 : s \to t,\ A \vdash e_2 : s}{A \vdash e_1 e_2 : t} \quad (3)$$

$$A \vdash 0 : \text{Int} \quad (4)$$

$$\frac{A \vdash e : \text{Int}}{A \vdash \text{succ } e : \text{Int}} \quad (5)$$

Return1 Return2 Return3

Example type derivations

- $\phi \vdash 0 : \text{Int}$
- succ

$$\frac{\phi[x : \text{Int}\ ] \vdash x : \text{Int}}{\dfrac{\phi[x : \text{Int}\ ] \vdash \text{succ } x : \text{Int}}{\phi \vdash \lambda x : \text{Int}\ .\text{succ } x : \text{Int} \ \to \text{Int}}}$$

Examples of type rules

- Identity function:

$$\frac{\phi[x : \text{Int}\ ] \vdash x : \text{Int}}{\phi \vdash \lambda x : \text{Int}\ .x : \text{Int} \ \to \text{Int}}$$

- Apply

$$\frac{\dfrac{\dfrac{\phi[f : s \to t][x : s] \vdash f : s \to t \qquad \phi[f : s \to t][x : s] \vdash x : s}{\phi[f : s \to t][x : s] \vdash f\ x : t}}{\phi[f : s \to t] \vdash \lambda x : s.f\ x : s \to t}}{\phi \vdash \lambda f : s \to t.\lambda x : s.f\ x : (s \to t) \to (s \to t)}$$

Type derivation for SKI

- I-combinator - identity function.
- K-combinator - **K** , when applied to any argument x returns a constant function **K** x, which when applied to any argument y returns x.
  K $xy = x$

$$\frac{\phi[x : s][y : t] \vdash x : s}{\dfrac{\phi[x : s] \vdash \lambda y : t.x : t \to s}{\phi \vdash \lambda x : s.\lambda y : t.x : s \to (t \to s)}}$$

- S-combinator, for substitution: **S** $xyz = xz(yz)$
  Useless assignment - derive the type derivation for **S** combinator.
- SKI is turing complete. Actually, SK itself is turing complete.
- Self study.

Example underivable term

- succ $(\lambda x : t.e)$
  (Recall Rule 5):

$$\frac{A \vdash e : \text{Int}}{A \vdash \text{succ } e : \text{Int}}$$

$$\text{underivable}\ \frac{A \vdash \lambda x : t.e : \text{Int}}{A \vdash \text{succ } (\lambda x : t.e) : \text{Int}}$$

- No rule to derive the hypothesis $\phi \vdash \lambda x : t.e$.
- succ $(\lambda x : t.e)$ has no simple type.

Type soundness

A type system for a programming language is sound
if well-typed programs cannot go wrong.

- Program = a closed expression.
- A value is a either a lambda or an integer constant.
- A program goes wrong if it does not evaluate to a value.
- An expression is in normal form if it cannot be further reduced.

Language semantics

$$\rightarrow_V \subseteq \mathit{Expression} \times \mathit{Expression}$$

$$(\lambda x.e)v \rightarrow_V e[x := v] \quad (6)$$

$$\frac{e_1 \rightarrow_V e_1'}{e_1 e_2 \rightarrow_V e_1' e_2} \quad (7)$$

$$\frac{e_2 \rightarrow_V e_2'}{v e_2 \rightarrow_V v e_2'} \quad (8)$$

$$\text{succ } c_1 \rightarrow_V c_2(\lceil c_2 \rceil = \lceil c_1 \rceil + 1) \quad (9)$$

$$\frac{e \rightarrow_V e'}{\text{succ } e \rightarrow_V \mathit{succ} e'} \quad (10)$$

Example:

$\lambda x.(\text{succ } \lambda y.(x+y))$ 2 3
Rule 6 : succ $\lambda y.(2+y)$ 3
Rule 8, 6 : (succ (2+3))
Rule 8 : succ 5
Rule 9 : 6

Recall from prior lecture on substitution

$$\begin{array}{lcl}
x[x := M] & \equiv & M \\
y[x := M] & \equiv & y \ (x \neq y) \\
(\lambda x.e_1)[x := M] & \equiv & (\lambda x.e_1) \\
(\lambda y.e_1)[x := M] & \equiv & \lambda z.((e_1[y := z])[x := M]) \\
 & & \text{(where } x \neq y \text{ and } z \text{ does not} \\
 & & \text{occur free in } e_1 \text{ or } M\text{).} \\
(e_1 e_2)[x := M] & \equiv & (e_1[x := M])(e_2[x := M]) \\
c[x := M] & \equiv & c \\
(\mathit{succ}\ e_1)[x := M] & \equiv & \mathit{succ}\ (e_1[x := M])
\end{array}$$

Wrong program

- An expession e is stuck if it is not a value and there is no expression e' such that $e \rightarrow_V e'$.
- Stuck expression $\Rightarrow$ runtime error.
- A program e goes wrong if $e \rightarrow_V *e'$ and e' is stuck.
- Example of stuck program: cv, and succ $\lambda x.e$

Theorem: Well typed programs cannot go wrong.

Proof of theorem in steps

Lemma

1. Useless Assumption: *If $A[x:s] \vdash e : t$, and x does not occur free in e then $A \vdash e : t$.*

Proof : Useless assignment

Proof of theorem in steps

Lemma

2. Substitution: *If $A[x:s] \vdash e : t$, and $A \vdash M : s$ then $A \vdash e[x := M] : t$.*

Proof: By induction on the structure of e. Five subcases (depending on the type rules (1) - (5)) ◂ Type rules ◂ Substitution

- Rule (1) $e \equiv y$: Two subcases:
  - $x \equiv y$. We have $y[x := M] \equiv M$.
    From $A[x:s] \vdash e : t$,
    $e \equiv y$, $x \equiv y$ and Rule (1)
    We have $(A[x:s])(x) = t$, so $s = t$.
    From $A \vdash M : s$ and $s = t$, we conclude that $A \vdash M : t$.
  - $x \not\equiv y$. We have $y[x := M] \equiv y$.
    From $A[x:s] \vdash e : t$,
    $e \equiv y$, and
    Rule (1), it follows that $A(y) = t$, so we conclude
    $A \vdash y : t$.

Proof of theorem in steps

◂ Type rules ◂ Substitution

- Rule (2). We have $e \equiv \lambda y.e_1$. Two subcases.
  - $x \equiv y$. We have $\lambda y.e_1[x := M] \equiv \lambda y.e_1$.
    Since x does not occur free in $\lambda y.e_1$,
    From Lemma 1 and the derivation $A[x:s] \vdash \lambda y.e_1 : t$ produces a derivation $A \vdash \lambda y.e_1 : t$.
  - $x \not\equiv y$. We have $\lambda y.e_1[x := M] \equiv \lambda z.((e_1[y := z])[x := M])$.
    The last step in the type derivation is of the form

$$\frac{A[x:s][y:t_2] \vdash e_1 : t_1}{A[x:s] \vdash \lambda y : t_2 : e_1 : t_2 \rightarrow t_1}$$

    From the premise of this rule and renaming of y to z, we have
    $A[x:s][z:t_2] \vdash e_1[y := z]$
    From the induction hypothesis, we have
    $A[z:t_2] \vdash ((e_1[y := z])[x := M]) : t_1$.
    So from Rule (2) we can derive
    $A \vdash \lambda z.((e_1[y := z])[x := M]) : t_2 \rightarrow t_1$.

Proof of theorem in steps

◂ Type rules ◂ Substitution

- Rule (3). We have $e \equiv e_1 e_2$, and $(e_1 e_2)[x := M] \equiv (e_1[x := M])(e_2[x := M])$.
  The last step in the derivation of the type rule is of the form:

$$\frac{A[x:s] \vdash e_1 : t_2 \to t,\ A[x:s] \vdash e_2 : t_2}{A[x:s] \vdash e_1 e_2 : t}$$

  From the induction hypothesis, we have $A \vdash e_1[x := M] : t_2 \to t$ and $A \vdash e_2[x := M] : t_2$.Using type derivation rule (3), we get $A \vdash e_1[x := M] e_2[x := M] : t$.
- Rule (4). We have $e \equiv c$, and $c[x := M] \equiv c$.
  The entire derivation of $A[x : s] \vdash e : t$ is of the form

$$A[x:s] \vdash c : \text{Int}$$

  Thus from rule (4) we have $A \vdash c : \text{Int}$.
- Rule (5): similar to Rule (3).

Lemma

3. Type preservation: *If $A \vdash e : t$, and $e \to_V e'$ then $A \vdash e' : t$.*

Lemma

4. Typable Value: *If $A \vdash v : \text{Int}$ then v is of the form c. If $A \vdash v : s \to t$ then v is of the form $\lambda x.e$.*

Lemma

5. Progress: *If e is a closed expression, and $A \vdash e : t$ then either e is a value or there exists e' such that $e \to_V e'$.*

Lemma

6. Closedness Preservation: *If e is closed, and $e \to_V e'$ then either e' is also closed.*

Type soundness

Corollary

Well typed programs cannot go wrong.

Proof

- Say we have a well typed program e. That is, e is closed and we have A, t, such that $A \vdash e : t$.
- Proof by contradiction. Say, e can go wrong.
- $\Leftrightarrow \exists$ a stuck expression e' such that $e \to_V^* e'$.
- From Lemma 6, e' is closed.
- From Lemma 3,we have $A \vdash e' : t$.
- From Lemma 5, we have e' is not stuck. Hence a contradiction.

□

Recap

- Type rules.
- Simply typed lambda calculus.
- Type soundness proof.