

CS6848 - Principles of Programming Languages

Principles of Programming Languages

V. Krishna Nandivada

IIT Madras

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?



- 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]);  
    }  
}
```



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

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

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



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?



Simply typed lambda calculus

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

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

- Extend the signature of a lambda: $\lambda x : \tau.e$ — every function specifies the type of its argument.
- Examples:

0	$:$	Int
$\lambda x : \text{Int} .(succ\ x)$	$:$	$\text{Int} \rightarrow \text{Int}$
$\lambda x : \text{Int} .succ\ \lambda y : \text{Int} .x + y$	$:$	$\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$
- These are simple types - each type can be viewed as a finite tree.
polymorphic types, dependent types
- Infinitely many types.



Type rules

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

$$A \vdash x : t(A(x) = t) \tag{1}$$

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

$$\frac{A \vdash e_1 : s \rightarrow t, A \vdash e_2 : s}{A \vdash e_1 e_2 : t} \tag{3}$$

$$A \vdash 0 : \text{Int} \tag{4}$$

$$\frac{A \vdash e : \text{Int}}{A \vdash succ\ e : \text{Int}} \tag{5}$$

- 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

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

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

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

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

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

Return1 Return2 Return3



Example type derivations

- $\phi \vdash 0 : \text{Int}$
- **SUCC**

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



Examples of type rules

- Identity function:

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

- Apply

$$\frac{\frac{\frac{\phi[f : s \rightarrow t][x : s] \vdash f : s \rightarrow t \quad \phi[f : s \rightarrow t][x : s] \vdash x : s}{\phi[f : s \rightarrow t][x : s] \vdash f x : t}}{\phi[f : s \rightarrow t] \vdash \lambda x : s. f x : s \rightarrow t}}{\phi \vdash \lambda f : s \rightarrow t. \lambda x : s. f x : (s \rightarrow t) \rightarrow (s \rightarrow t)}$$



Type derivation for SKI

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

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

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



Example underivable term

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

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

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 $\phi \vdash \lambda x : t.e$.
- $\text{succ } (\lambda x : t.e)$ 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 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 (recall small step semantics)

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

$$(\lambda x.e)v \rightarrow_V e[x := v] \tag{6}$$

$$\frac{e_1 \rightarrow_V e'_1}{e_1 e_2 \rightarrow_V e'_1 e_2} \tag{7}$$

$$\frac{e_2 \rightarrow_V e'_2}{ve_2 \rightarrow_V ve'_2} \tag{8}$$

$$\text{succ } c_1 \rightarrow_V c_2 \text{ (}[c_2] = [c_1] + 1) \tag{9}$$

$$\frac{e \rightarrow_V e'}{\text{succ } e \rightarrow_V \text{succ } e'} \tag{10}$$

Example:

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



Recall from prior lecture on substitution

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



- An expression 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 $\text{succ } \lambda x.e$

Theorem: Well typed programs cannot go wrong.



Lemma

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



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
 $A \vdash y : t$.



◀ 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:

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

From the induction hypothesis, we have $A \vdash e_1[x := M] : t_2 \rightarrow t$ 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).



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 $e \rightarrow_V^* e'$.
- 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. \square



Lemma

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

Lemma

4. Typable Value: *If $A \vdash v : \text{Int}$ then v is of the form c . If $A \vdash v : s \rightarrow t$ 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 $e \rightarrow_V e'$.*

Lemma

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



Lemmas' recollect

Lemma

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

Lemma

4. Typable Value: *If $A \vdash v : \text{Int}$ then v is of the form c . If $A \vdash v : s \rightarrow t$ 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 $e \rightarrow_V e'$.*

Lemma

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



Recap

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

