Jens Palsberg Sep 24, 1999

Lecture Note: Types

Contents

1 Introduction

What is a Type?

A type is an *invariant.* For example, the Java declaration

```
int v;
```

specifies that `v` may only contain integer values in a certain range.

Why Types?

An untyped program may be:

- Unreadable.

  Types provide documentation;

  "Well-typed programs are more readable"

- Unreliable.

  Types provide a safety guarantee;

  "Well-typed programs cannot go wrong"

- Inefficient.

  Types enable optimizations;

  "Well-typed programs are faster"

2 Simple Types

Our example language is a λ-calculus where the only two kinds of data are functions and integers. The language is generated by the following grammar:

$$\begin{array}{lll} e & \in & \mathit{Expression} \\ e & ::= & x \mid \lambda x.e \mid e_1 e_2 \mid c \mid \mathsf{succ}\ e \\ x & \in & \mathit{Var} \quad \text{(infinite set of variables)} \\ c & \in & \mathit{IntegerConstant} \end{array}$$

We will use the standard convention that when writing $\lambda x.e$, then e is everything until the next ")" (or until the end of the whole λ term). The slogan is, "the body of a λ extends as long as possible." We will also use the standard convention that $e_1 e_2 e_3$ should be grouped as $(e_1 e_2) e_3$. An expression is *closed* if it does not contain free variables.

We need two kinds of type: function types and an integer type. Types are generated from the grammar:

$$t \ ::= \ t_1 \to t_2 \mid \mathsf{Int}$$

Such types are called *simple types*. The basic idea is that we can assign natural types to expressions, for example:

$$\begin{array}{rcl} 0 & : & \mathsf{Int} \\ \lambda x.(\mathsf{succ}\ x) & : & \mathsf{Int} \to \mathsf{Int}\ . \end{array}$$

Notice that there are infinitely many types. Notice also that each type can be viewed as a tree: the syntax tree of the type.

A type environment is a partial function with finite domain which maps elements of Var to types. We use A to range over type environments. We use $\emptyset$ to denote the type environment with empty domain. We use $A[x : t]$ to denote a partial function which maps x to t, and maps y, where $y \neq x$, to $A(y)$.

Let us now consider a formal system for assigning types to λ-terms. If e is a λ-term, t is a type, and A is a type environment, then the judgment

$$A \vdash e : t$$

means that e has the type t in the environment A. Formally, this holds when the judgment is derivable by a finite derivation tree using the following five rules:

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

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

$$\frac{A \vdash e_1 : s \to t \quad A \vdash e_2 : s}{A \vdash e_1 e_2 : t} \tag{3}$$

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

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

An expression e is *well typed* if there exist A, t such that $A \vdash e : t$ is derivable.

Notice that there is one rule for each of the five constructs in the language. The hypotheses, if any, are written above the line, and the conclusion is written below the line. Two of the rules have no hypotheses so we call them axioms.

We can now use these rules to derive the typings of $\mathsf{0}$ and $\lambda x.(\mathsf{succ}\ x)$ from above. The first case is easy:

$$\emptyset \vdash \mathsf{0} : \mathsf{Int}$$

The judgment is indeed an instance of the axiom for $\mathsf{0}$. The second case requires the use of the rules for abstraction, succ, and variable:

$$\cfrac{\cfrac{\emptyset[x : \mathsf{Int}] \vdash x : \mathsf{Int}}{\emptyset[x : \mathsf{Int}] \vdash \mathsf{succ}\ x : \mathsf{Int}}}{\emptyset \vdash \lambda x.\mathsf{succ}\ x : \mathsf{Int} \to \mathsf{Int}}$$

Here are some more examples of the use of the five rules. The identity function:

$$\frac{\emptyset[x : \mathsf{Int}] \vdash x : \mathsf{Int}}{\emptyset \vdash \lambda x.x : \mathsf{Int} \to \mathsf{Int}}$$

The apply function:

$$\cfrac{\cfrac{\cfrac{\emptyset[f : s \to t][x : s] \vdash f : s \to t \qquad \emptyset[f : s \to t][x : s] \vdash x : s}{\emptyset[f : s \to t][x : s] \vdash fx : t}}{\emptyset[f : s \to t] \vdash \lambda x.fx : s \to t}}{\emptyset \vdash \lambda f.\lambda x.fx : (s \to t) \to (s \to t)}$$

The K combinator:

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

The following λ-term does not have a simple type:

$\mathsf{succ}\ (\lambda x.e)$

The problem is that $\lambda x.e$ is not an integer. If we try to build a type derivation, then may start with Rule (5):

$$\dfrac{\emptyset \vdash \lambda x.e : \mathsf{Int}}{\emptyset \vdash \mathsf{succ}\ (\lambda x.e) : \mathsf{Int}}$$

There is no rule with which we can derive the hypothesis $\emptyset \vdash \lambda x.e : \mathsf{Int}$, so we conclude that $\mathsf{succ}\ (\lambda x.e)$ does not have a simple type.

3 Type Soundness

A type system for a programming language is *sound* if well-typed programs cannot cause type errors.

A *program* is a closed expression. A *value* is either a λ-abstraction $\lambda x.e$ or an integer constant c. We use v to range over values. We use $\lceil c \rceil$ to denote the integer represented by an integer constant c.

A small-step call-by-value operational semantics for the language is given by the reflexive, transitive closure of the relation $\rightarrow_V$:

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

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

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

$$\frac{e_2 \rightarrow_V e_2'}{v\ e_2 \rightarrow_V v\ e_2'} \tag{8}$$

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

$$\frac{e_1 \rightarrow_V e_2}{\mathsf{succ}\ e_1 \rightarrow_V \mathsf{succ}\ e_2} \tag{10}$$

The notation $e[x := M]$ denotes e with M substituted for every free occurrence of x.

$$\begin{array}{rcl}
x[x := M] & \equiv & M \\
y[x := M] & \equiv & y \quad (x \not\equiv y) \\
(\lambda x.e_1)[x := M] & \equiv & \lambda x.e_1 \\
(\lambda y.e_1)[x := M] & \equiv & \lambda z.((e_1[y := z])[x := M]) \quad (x \not\equiv y \text{ and } z \text{ fresh}) \\
(e_1\ e_2)[x := M] & \equiv & (e_1[x := M])\ (e_2[x := M]) \\
c[x := M] & \equiv & c \\
(\mathsf{succ}\ e_1)[x := M] & \equiv & \mathsf{succ}\ (e_1[x := M])
\end{array}$$

An expression e is *stuck* if it is not a value and there is no expression e' such that $e \rightarrow_V e'$. Intuitively, a stuck expression is just about to produce a run-time error. A program e *goes wrong* if $e \rightarrow_V^* e'$ and e' is stuck.

Examples of stuck expressions include cv and $\mathsf{succ}\ (\lambda x.e)$. Intuitively, these expressions are stuck because c is not a function, and succ cannot be applied to functions.

We will now prove that well-typed programs cannot go wrong (Corollary 3.7). The proof technique is standard. For example, several research groups have produced a similar theorem and proof for a considerable subset of Java, for example, Flatt, Krishnamurthi and Felleisen, “A Programmer’s Reduction Semantics for Classes and Mixins ”, Rice University, Department of Computer Science, TR 97–293, revised June 1999. Many such big type soundness proofs have been checked by automatic proof checkers.

Lemma 3.1 (Useless Assumption) *If $A[x : s] \vdash e : t$, and x does not occur free in e, then $A \vdash e : t$.*

Proof. Left to the reader! □

Lemma 3.2 (Substitution) *If $A[x : s] \vdash e : t$ and $A \vdash M : s$, then $A \vdash e[x := M] : t$.*

Proof. We proceed by induction on the structure of the size of e. There are now five subcases depending on which one of Rules (1)–(5) was the last one used in the derivation of $A[x : s] \vdash e : t$.

- Rule (1). We have $e \equiv y$. There are 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 $A \vdash M : t$.
  - $x \not\equiv y$. 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 $A \vdash y : t$.
- Rule (2). We have $e \equiv \lambda y.e_1$. There are 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 $\lambda y.e_1$, we can from Lemma 3.1 and the derivation of $A[x : s] \vdash \lambda y.e_1 : t$ produce a derivation of $A \vdash \lambda y.e_1 : t$.

  - $x \not\equiv y$. We have $(\lambda y.e_1)[x := M] \equiv \lambda z.((e_1[y := z])[x := M])$. The last step in the derivation of $A[x : s] \vdash e : t$ is of the form

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

    From $A[x : s][y : t_2] \vdash e_1 : t_1$ and a renaming of y to z, we have $A[x : s][z : t_2] \vdash e_1[y := z] : t_1$. Notice that the expressions e_1 and $e_1[y := z]$ have the same size. 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 $A \vdash \lambda z.((e_1[y := z])[x := M]) : t_2 \to t_1$.

- 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 $A[x : s] \vdash e : t$ is of the form

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

  From the induction hypothesis we have $A \vdash e_1[x := M] : t_2 \to t$ and $A \vdash e_2[x := M] : t_2$, and from 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

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

  and from Rule (4) we have $A \vdash c : \mathsf{Int}$.

- Rule (5). This case is similar to the case of Rule (3).

□

Theorem 3.3 (Type Preservation) *If $A \vdash e : t$ and $e \to_V e'$, then $A \vdash e' : t$.*

Proof. We proceed by induction on the structure of the derivation of $A \vdash e : t$. There are now five subcases depending on which one of Rules (1)–(5) was the last one used in the derivation of $A \vdash e : t$.

- Rule (1). We have $e \equiv x$, so $e \rightarrow_V e'$ is not possible.
- Rule (2). We have $e \equiv \lambda x.e_1$, so $e \rightarrow_V e'$ is not possible.
- Rule (3). We have $e \equiv e_1 e_2$. There are now three subcases depending on which one of Rules (6)–(8) was the last one used in the derivation of $e \rightarrow_V e'$.
  - Rule (6). We have $e \equiv (\lambda x.e_1)v$ and $e' \equiv e_1[x := v]$. The last part of the derivation of $A \vdash e : t$ is of the form

    $$\frac{\dfrac{A[x:s] \vdash e_1 : t}{A \vdash \lambda x.e_1 : s \rightarrow t} \quad A \vdash v : s}{A \vdash (\lambda x.e_1)v : t}$$

    From Lemma 3.2, $A[x : s] \vdash e_1 : t$, and $A \vdash v : s$ we get $A \vdash e_1[x := v] : t$.
  - Rules (7)–(8). In each case $A \vdash e' : t$ follows from the induction hypothesis, and Rule (3).
- Rule (4). We have $e \equiv c$, so $e \rightarrow_V e'$ is not possible.
- Rule (5). We have $e \equiv \mathsf{succ}\ e_1$. There are now two subcases depending on which one of Rules (9)–(10) was the last one used in the derivation of $e \rightarrow_V e'$.
  - Rule (9). We have $e \equiv \mathsf{succ}\ c_1$ and $e' \equiv c_2$, where $\lceil c_2 \rceil = \lceil c_1 \rceil + 1$. The last judgment in the derivation of $A \vdash e : t$ is of the form $A \vdash \mathsf{succ}\ c_1 : \mathsf{Int}$, and from Rule (4) we have $A \vdash c_2 : \mathsf{Int}$.
  - Rule (10). We have $e \equiv \mathsf{succ}\ e_1$ and $e' \equiv \mathsf{succ}\ e_2$, and $e_1 \rightarrow_V e_2$. The last part of the derivation of $A \vdash e : t$ is of the form

    $$\frac{A \vdash e_1 : \mathsf{Int}}{A \vdash \mathsf{succ}\ e_1 : \mathsf{Int}}$$

    From the induction hypothesis we have $A \vdash e_2 : \mathsf{Int}$, and from Rule (10) we derive $A \vdash \mathsf{succ}\ e_2 : \mathsf{Int}$.

$\square$

Lemma 3.4 (Typable Value) *If $A \vdash v : \mathsf{Int}$, then v is of the form c. If $A \vdash v : s \rightarrow t$, then v is of the form $\lambda x.e$.*

Proof. Immediate from Rule (2) and Rule (4). □

The following lemma states that a well-typed program is not stuck.

Lemma 3.5 (Progress) *If e is a closed expression, and $A \vdash e : t$, then either e is a value, or there exists e' such that $e \rightarrow_V e'$.*

Proof. We proceed by induction on the structure of the derivation of $A \vdash e : t$. There are now five subcases depending on which one of Rules (1)–(5) was the last one used in the derivation of $A \vdash e : t$.

- Rule (1). We have $e \equiv x$, and x is not closed.
- Rule (2). We have $e \equiv \lambda x.e_1$, so e is a value.
- Rule (3). We have $e \equiv e_1 e_2$. We have that e is closed, so also e_1, e_2 are closed. The last step in the derivation of $A \vdash e : t$ is of the form

  $$\frac{A \vdash e_1 : s \rightarrow t \quad A \vdash e_2 : s}{A \vdash e_1 e_2 : t}$$

  From the induction hypothesis we have that (1) either e_1 is a value, or there exists e_1' such that $e_1 \rightarrow_V e_1'$ and (2) either e_2 is a value, or there exists e_2' such that $e_2 \rightarrow_V e_2'$. We proceed by case analysis.

  – If there exists e_1' such that $e_1 \rightarrow_V e_1'$, then $e_1 e_2 \rightarrow_V e_1' e_2$ by Rule (7).

  – If e_1 is a value, and there exists e_2' such that $e_2 \rightarrow_V e_2'$, then $e_1 e_2 \rightarrow_V e_1 e_2'$ by Rule (8).

  – If e_1, e_2 are values, then from $A \vdash e_1 : s \rightarrow t$ and Lemma 3.4 we have that e_1 is of the form $\lambda x.e_3$, and hence $e_1 e_2 \rightarrow_V e_3[x := e_2]$ by Rule (6).
- Rule (4). We have $e \equiv c$, so e is a value.
- Rule (5). We have $e \equiv \mathsf{succ}\ e_1$. We have that e is closed, so also e_1 is closed. The last step in the derivation of $A \vdash e : t$ is of the form

  $$\frac{A \vdash e_1 : \mathsf{Int}}{A \vdash \mathsf{succ}\ e_1 : \mathsf{Int}}$$

From the induction hypothesis we have that either e_1 is a value, or there exists e_1' such that $e_1 \rightarrow_V e_1'$. We proceed by case analysis.

- If e_1 is a value, then from $A \vdash e_1 : \mathsf{Int}$ and Lemma 3.4 we have that e_1 is of the form c_1, and hence $\mathsf{succ}\ c_1 \rightarrow_V c_2$ where $\lceil c_2 \rceil = \lceil c_1 \rceil + 1$ by Rule (9).
- If there exists e_1' such that $e_1 \rightarrow_V e_1'$, then $\mathsf{succ}\ e_1 \rightarrow_V \mathsf{succ}\ e_1'$ by Rule (10).

$\square$

Lemma 3.6 (Closedness Preservation) *If e is closed, and $e \rightarrow_V e'$, then e' is closed.*

Proof. Left to the reader! $\square$

Corollary 3.7 *Well-typed programs cannot go wrong.*

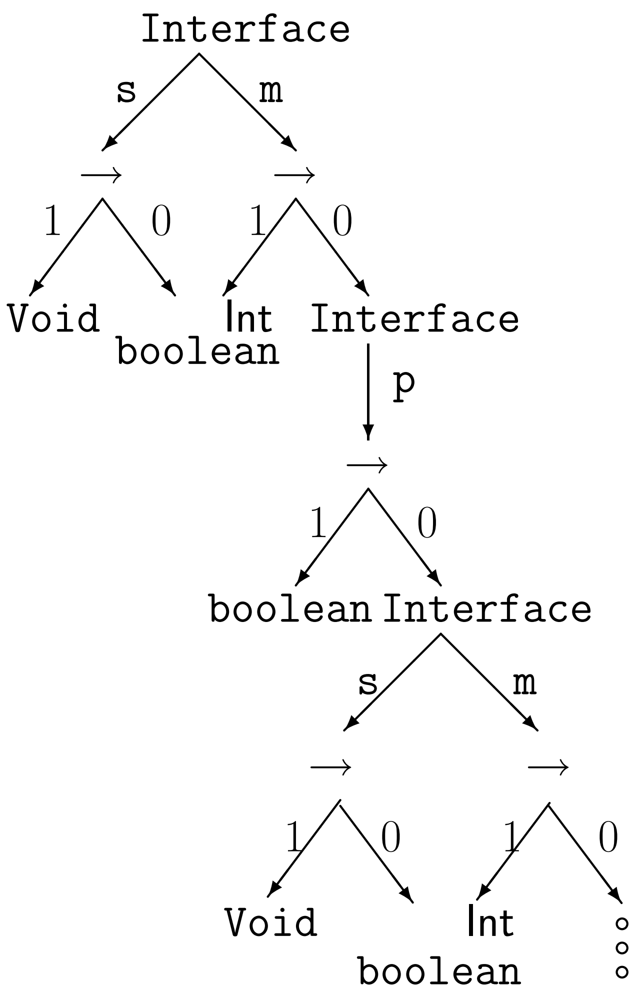
Proof. Suppose we have a well-typed program e, that is, e is closed and we have A, t such that $A \vdash e : t$. Suppose also that e can go wrong, that is, there exists a stuck expression e' such that $e \rightarrow_V^* e'$. From Lemma 3.6 we have that e' is closed. From Theorem 3.3 we have that $A \vdash e' : t$. From Lemma 3.5 we have that e' is not stuck, a contradiction. We conclude that e' does not exist so e cannot go wrong. $\square$

4 Recursive Types

Simple types are finite. When we draw a simple type as a tree, it is finite. This is in contrast to Java where the definitions of interfaces and classes can be mutually recursive. If we *unfold* a Java class or interface by keeping replacing occurrences of interface names and class names by their definition, then we may eventually end up with an infinite tree. For example,

```
interface I {
  void s(boolean a);
  int m(J a);
}

interface J {
  boolean p(I a);
}
```



In Java, two types are considered equal if and only if they have the same name. Similarly, subtyping between interfaces and classes is defined in terms of the names of the interfaces and classes. The usual terminology is that type equality and subtyping in Java are *name based*. By using the idea of unfolding, we can abstract away from the names and get *structure-based* definitions of type equality and subtyping. This is useful for sending objects and types from one name space to another. In this section we will look at the details of unfolding types into infinite trees, and in the following section we will define a structural notion of subtyping on such infinite trees.

Let us now extend the grammar for simple types such that types are allowed to be infinite.

$$t \ ::= \ t_1 \to t_2 \mid \mathsf{Int} \mid \alpha \mid \mu\alpha.(t_1 \to t_2)$$

where α is a variable that ranges over types, and $\mu\alpha.t$ is a *recursive type* which allow the unfolding:

$$\mu\alpha.t \quad = \quad t[\alpha := (\mu\alpha.t)]$$

For example, let u be the type $\mu\alpha.(\alpha \to \mathsf{Int})$. By unfolding once, we get that $u = u \to \mathsf{Int}$. By unfolding twice, we get that $u = (u \to \mathsf{Int}) \to \mathsf{Int}$. If we unfold infinitely, we get an infinite tree. That tree is the meaning of u.

In summary, if we generate something from the grammar above, then we can unfold into an infinite tree where all the μs have disappeared. That infinite tree is the type. Such a type will always have only finitely many distinct subtrees. Trees of that kind are called *regular trees*. Conversely, any regular tree can be written as a finite expression with μs.

Let us now return to the λ-calculus. Consider the λ-term $\lambda x.xx$. With the type $u = \mu\alpha.(\alpha \to \mathsf{Int})$ it is easy to build a type derivation:

$$\dfrac{\dfrac{\emptyset[x : u] \vdash x : u \to \mathsf{Int} \qquad\qquad \emptyset[x : u] \vdash x : u}{\emptyset[x : u] \vdash xx : \mathsf{Int}}}{\emptyset \vdash \lambda x.xx : u \to \mathsf{Int}}$$

In the left part of the tree we have $\emptyset[x : u] \vdash u \to \mathsf{Int}$ and this is okay because $u = u \to \mathsf{Int}$.

Let us consider another application of the type $u = \mu\alpha.(\alpha \to \mathsf{Int})$. There is a famous λ-term which is known as the "paradoxical combinator." This λ-term has the property that when applied to a function g, then it produces a fixed point of g. The λ-term is:

$$Y \quad = \quad \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Notice that for any λ-term E we have

$$\begin{aligned} Y(E) &\rightsquigarrow_\beta (\lambda x.E(xx))(\lambda x.E(xx)) \\ &\rightsquigarrow_\beta E((\lambda x.E(xx))(\lambda x.E(xx))) \\ &=_\beta E(Y(E)) \end{aligned}$$

So, $Y(E)$ is a fixed point of E!

Can we type Y? Not with simple types. But it is easy with recursive types. Let us again use the type $u = \mu\alpha.(\alpha \to \mathsf{Int})$. Here is part of the type derivation:

$$\frac{\emptyset[f : \mathsf{Int} \to \mathsf{Int}] \vdash (\lambda x.f(xx))(\lambda x.f(xx)) : \mathsf{Int}}{\emptyset \vdash \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) : (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}}$$

It is easy to see that if $\lambda x.f(xx)$ has type u, then we can use $u = u \to \mathsf{Int}$ to get the hypothesis of the above derivation. So we must derive

$$\emptyset[f : \mathsf{Int} \to \mathsf{Int}] \vdash \lambda x.f(xx) : u$$

We will give x the type $u \to \mathsf{Int}$, which is actually u, and we must then show that $f(xx)$ has type Int. This in turn mean that xx must have type Int. This is easy to achieve because $u = u \to \mathsf{Int}$. Here is most of the type derivation:

$$\frac{\dfrac{\emptyset[f : \mathsf{Int} \to \mathsf{Int}][x : u] \vdash f : \mathsf{Int} \to \mathsf{Int} \qquad \emptyset[f : \mathsf{Int} \to \mathsf{Int}][x : u] \vdash xx : \mathsf{Int}}{\emptyset[f : \mathsf{Int} \to \mathsf{Int}][x : u] \vdash f(xx) : \mathsf{Int}}}{\emptyset[f : \mathsf{Int} \to \mathsf{Int}] \vdash \lambda x.f(xx) : u}$$

It is easy to see that we can replace Int with any other type and still have a valid type derivation.

Not all λ-terms can be typed with recursive types. For example,

$$\lambda x.x(\mathsf{succ}\ x)$$

is a quite weird λ-term that we cannot type even with recursive types.

Corollary 3.7 also holds for recursive types: same type rules, and recursive types instead of simple types.

Types as Functions

We have seen that we can view a type as a possibly infinite tree. We can represent such a tree by a function which maps *paths* to *labels* from a set Σ. Such functions are called *terms*. For example, for the types in Sections 3–4, all paths are in the set $\{0,1\}^*$, where we use 0 to denote Left, and 1 to denote Right, and all labels are from the set $\Sigma = \{\mathsf{Int}, \rightarrow\}$. A term t over Σ is a partial function

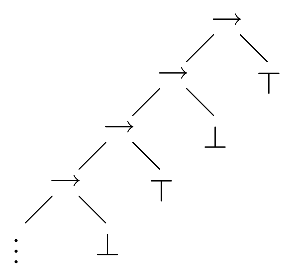
$$t : \{0,1\}^* \quad \rightarrow \quad \Sigma$$

with domain $\mathcal{D}(t)$ satisfying

- $\mathcal{D}(t)$ is nonempty and prefix-closed;
- if $t(\alpha) = \rightarrow$, then $\alpha 0, \alpha 1 \in \mathcal{D}(t)$.

The set of all terms over Σ is denoted T_Σ.

For example, the type



can be represented by the term t:

$$\begin{aligned} t(0^n) &= \rightarrow \\ t(0^{2n}1) &= \top \\ t(0^{2n+1}1) &= \bot \end{aligned}$$

A term t is *finite* if its domain $\mathcal{D}(t)$ is a finite set. We denote the set of finite terms over Σ by F_Σ.

Subterms: $t{\downarrow}\alpha(\beta) = t(\alpha\beta)$.

A term t is *regular* if it has only finitely many distinct subterms; *i.e.*, if $\{t{\downarrow}\alpha \mid \alpha \in \omega^*\}$ is a finite set.

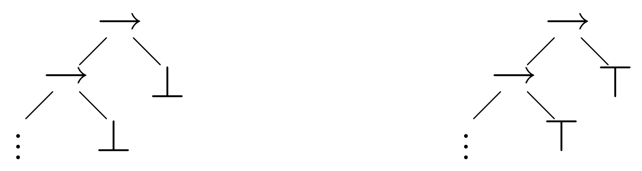
Types as Automata

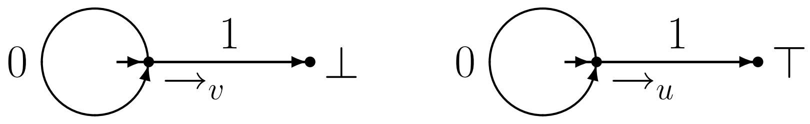
If t is a term, then the following are equivalent:

(i) t is regular;

(ii) t is representable by a term automaton;

(iii) t is describable by a type expression involving μ.





5 Subtyping

Motivation: the extends keyword in Java.

In general, we will work with an ordering $\leq$ of the types, the subtype order.

One extra typing rule, known as "subsumption":

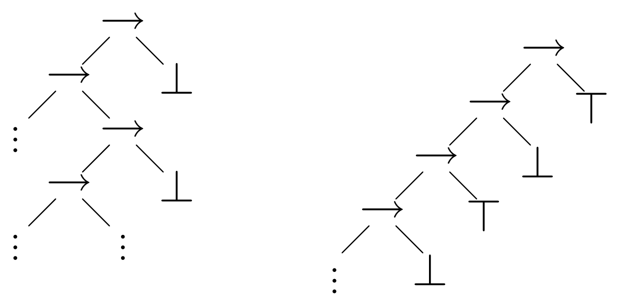
$$\frac{A \vdash e : t \quad t \leq t'}{A \vdash e : t'}$$

Consider this set of types:

$$t \ ::= \ v \mid \bot \mid \top \mid t \to t \mid \mu v.(t \to t) \qquad \text{(types)}$$

Representing types as labeled trees:

$$\mu u.((u \to u) \to \bot) \ \leq \ \mu v.((v \to \bot) \to \top)$$



The order $\leq_{\mathrm{FIN}}$ is the smallest binary relation on F_Σ, with $\Sigma = \{\bot, \to, \top\}$, such that

(i) $\bot \leq_{\mathrm{FIN}} t \leq_{\mathrm{FIN}} \top$ for all finite t;

(ii) if $s' \leq_{\mathrm{FIN}} s$ and $t \leq_{\mathrm{FIN}} t'$ then $s \to t \leq_{\mathrm{FIN}} s' \to t'$.

Parity

The *parity* of $\alpha \in \{0,1\}^*$ is the number mod 2 of 0's in α. The parity of α is denoted $\pi\alpha$. A string α is said to be *even* if $\pi\alpha = 0$ and *odd* if $\pi\alpha = 1$.

Let $\leq_0$ be the linear order

$$\perp \;\leq_0\; \rightarrow \;\leq_0\; \top$$

on Σ, and let $\leq_1$ be its reverse

$$\top \;\leq_1\; \rightarrow \;\leq_1\; \perp .$$

The Type Ordering

For $s, t \in T_\Sigma$, define $s \leq t$ iff $s(\alpha) \leq_{\pi\alpha} t(\alpha)$ for all $\alpha \in \mathcal{D}(s) \cap \mathcal{D}(t)$.

A *counterexample* to $s \leq t$ is a path $\alpha \in \mathcal{D}(s) \cap \mathcal{D}(t)$ for which $s(\alpha) \not\leq_{\pi\alpha} t(\alpha)$.

Slogan: Two trees are ordered if no common path detects a counterexample.

Basic Result: The relation $\leq$ is a partial order on T_Σ, and agrees with $\leq_{\text{FIN}}$ on F_Σ.

6 Decision Procedure for Subtyping

There is an efficient algorithm that decides if one recursive type is a subtype of another.

$$t \ ::= \ v \mid \bot \mid \top \mid t \to t \mid \mu v.(t \to t) \qquad \text{(types)}$$

Given types s, t, a certain product automaton accepts all counterexamples to $s \leq t$, that is, the set

$$\{\alpha \in \mathcal{D}(s) \cap \mathcal{D}(t) \mid s(\alpha) \not\leq_{\pi\alpha} t(\alpha)\}.$$

Algorithm

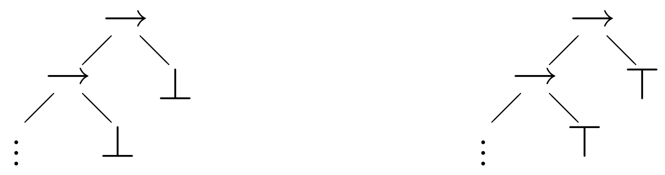
Input: Two types s, t.
Output: Is $s \leq t$?

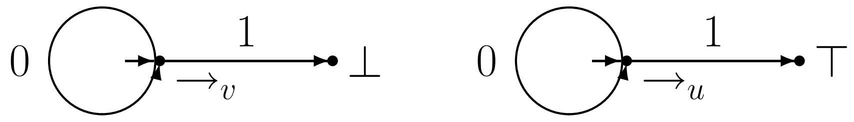
1. Construct the term automata for s and t (linear time).
2. Construct the product automaton (size $O(n^2)$).
3. Decide, using depth first search, if the product automaton accepts the empty set (linear time).
4. If so then $s \leq t$ else $s \not\leq t$.

Time complexity: $O(n^2)$.

Example

$$\mu v.(v \to \bot) \not\leq \mu u.(u \to \top)$$





Product automaton:

$$\begin{tikzcd}
 & {} \arrow[d] & \\
 & (\to_v, \to_u, 0) \arrow[dl, bend left, "0"] \arrow[dr, "1"] & \\
(\to_v, \to_u, 1) \arrow[ur, bend left, "0"] \arrow[dr, "1"] & & (\bot, \top, 0) \\
 & ((\bot, \top, 1)) &
\end{tikzcd}$$

7 First-Order Unification

Let us consider equations in which the variables range over finite trees. The problem of first-order unification is to decide whether a set of such equations has a solution. Moreover, if there is a solution, then we want the most general solution. One can solve this unification problem in $O(n)$ time, where n is the size of the equation set (M. S. Paterson and M. N. Wegman, "Linear Unification", Journal of Computer and System Sciences, 16:158–167, 1978.) Here, we will not cover this linear-time algorithm. Rather we will present a simpler algorithm with worse worst-case complexity.

For simplicity of notation, we will focus on equations where each side is a simple-type expression generated from the grammar

$$t ::= t \to t \mid \mathsf{Int} \mid \alpha$$

where α ranges over a set of type variables. A substitution is of the form

$$[\alpha_1 := s_1, \ldots, \alpha_m := s_m]$$

where $\alpha_1, \ldots, \alpha_m$ are type variables, and $s_1, \ldots, s_m$ are type expressions. When a substitution is applied to a type expression, $\alpha_1, \ldots, \alpha_m$ are replaced simultaneously. For example

$$\begin{array}{rcl}
((\mathsf{Int} \to \alpha) \to \beta)[\alpha := \mathsf{Int}, \beta := (\mathsf{Int} \to \mathsf{Int})] & = & (\mathsf{Int} \to \mathsf{Int}) \to (\mathsf{Int} \to \mathsf{Int}) \\
((\mathsf{Int} \to \alpha) \to \beta)[\alpha := \mathsf{Int}, \beta := (\mathsf{Int} \to \alpha)] & = & (\mathsf{Int} \to \mathsf{Int}) \to (\mathsf{Int} \to \alpha) \\
((\mathsf{Int} \to \alpha) \to \gamma)[\alpha := \mathsf{Int}, \beta := (\mathsf{Int} \to \alpha)] & = & (\mathsf{Int} \to \mathsf{Int}) \to \gamma.
\end{array}$$

Let G be the set of equations

$$\begin{array}{rcl}
s_1 & = & t_1 \\
\ldots & & \\
s_n & = & t_n.
\end{array}$$

If σ is a substitution, then we denote by $G\sigma$ the set of equations

$$\begin{array}{rcl}
s_1\sigma & = & t_1\sigma \\
\ldots & & \\
s_n\sigma & = & t_n\sigma.
\end{array}$$

If for every $i \in 1..n$ we have that $s_i\sigma$ is identical to $t_i\sigma$, then we say that G has solution σ. For example, the two equations

$$\begin{array}{rcl} \alpha &=& \beta \rightarrow \mathsf{Int} \\ \beta &=& \mathsf{Int} \rightarrow \mathsf{Int}\ , \end{array}$$

have the solution σ where

$$\sigma \quad = \quad [\alpha := ((\mathsf{Int} \rightarrow \mathsf{Int}) \rightarrow \mathsf{Int}),\ \beta := (\mathsf{Int} \rightarrow \mathsf{Int})].$$

The equation

$$\alpha \quad = \quad \alpha \rightarrow \mathsf{Int}\ ,$$

has no solution because of the cycle. In general, the forms of equations we consider can be unsolvable for two reasons:

1. we encounter a *cycle*, that is, an equation of one of the forms

   $$\begin{array}{rcl} \alpha &=& (\ldots\ \alpha\ \ldots) \rightarrow \ldots \qquad \text{or} \\ \alpha &=& \ldots\ \rightarrow (\ldots\ \alpha\ \ldots)\ , \end{array}$$

   or

2. we encounter an equation where neither of the two sides are variables, and where the two sides have *different root symbols*, that is, the equation is of the form $\mathsf{Int} = s \rightarrow t$.

Note that although equations with cycles and/or different root symbols may not be immediately present in an equation set, we may encounter such equations after replacing equals by equals.

If we have an equation $s = t$, and a substitution σ such that $s\sigma = t\sigma$, then σ is called a *unifier* of s and t, and we say that σ *unifies* s and t, that σ *solves* $s = t$, and that σ *satisfies* $s = t$. If for type expressions s, t, there exists a unifier σ of s and t, then we say that s and t are *unifiable*.

We will denote the composition of two substitutions σ, θ by $\sigma \circ \theta$, and this notation means

$$t(\sigma \circ \theta) \quad = \quad (t\sigma)\theta.$$

A substitution σ which solves an equation system is called a *most general solution* of that equation system provided that for any other solution θ, there exists a substitution τ such that $\theta = \sigma \circ \tau$.

A Unification Algorithm

We now present a first-order unification algorithm which either decides that an equation system has no solution, or produces the most general solution. The algorithm uses three variables G, $Failure$, σ:

- G ranges over sets of equations, and it is initially the input set of equations,
- $Failure$ is a boolean variable which is initially false, and
- σ is a substitution which is initially empty.

After initialization, the algorithm proceeds as follows:

```
while $G \neq \emptyset$ and (not Failure) do
        Choose and remove an equation $e$ from $G$, and let $(s = t)$ be $(e\sigma)$.
        There are now six cases depending on the form of $(s = t)$.
            1. If $s, t$ are identical variables, or $s, t$ are both $\mathsf{Int}$,
               then do nothing.
            2. If $s = s_1 \rightarrow s_2$ and $t = t_1 \rightarrow t_2$,
               then $G = G \cup \{s_1 = t_1, s_2 = t_2\}$.
            3. If $(s = \mathsf{Int}$ and $t = t_1 \rightarrow t_2)$, or $(s = s_1 \rightarrow s_2$ and $t = \mathsf{Int})$,
               then $Failure = true$.
            4. If $s$ is a variable that does not occur in $t$,
               then $\sigma = \sigma \circ [s := t]$.
            5. If $t$ is a variable that does not occur in $s$,
               then $\sigma = \sigma \circ [t := s]$.
            6. If $s \neq t$, and either $s$ is a variable that occurs in $t$, or $t$ is a
               variable that occurs in $s$,
               then $Failure = true$.
end while

If $Failure$, then output "no solution", else output $\sigma$.

END of algorithm
```

The loop invariant is:

If θ is the most general solution of G, then $\sigma \circ \theta$ is the most general solution of the input set of equations.

In case (4) and (5), the phrase “does not occur in” guards against cycles. This check is expensive computationally; it is widely known as the “occurs-check.”

Examples

The first example has no solution because of a cycle, and the second has a solution.

Example 1

Equations:

$$\begin{array}{rcl} \alpha & = & \mathsf{Int} \to \beta \\ \beta & = & \alpha \to \mathsf{Int}. \end{array}$$

Initially we have

$$\begin{array}{rcl} G & = & \{(\alpha = \mathsf{Int} \to \beta), (\beta = \alpha \to \mathsf{Int})\} \\ Failure & = & false \\ \sigma & = & \emptyset. \end{array}$$

Choose and remove, say, $(\alpha = \mathsf{Int} \to \beta)$ from G. We have $(\alpha = \mathsf{Int} \to \beta)\sigma \equiv (\alpha = \mathsf{Int} \to \beta)$, so we are in case (4). Change σ to $[\alpha := (\mathsf{Int} \to \beta)]$. We now have $G = \{(\beta = \alpha \to \mathsf{Int})\}$. Choose(!) and remove $(\beta = \alpha \to \mathsf{Int})$ from G. We have $(\beta = \alpha \to \mathsf{Int})\sigma \equiv (\beta = (\mathsf{Int} \to \beta) \to \mathsf{Int})$, so there is a cycle and we are in case (6), and the algorithm outputs that there is no solution.

Example 2

Equations:

$$\begin{array}{rcl} \alpha & = & \beta \to \mathsf{Int} \\ \beta & = & \mathsf{Int} \to \mathsf{Int}. \end{array}$$

Initially we have

$$\begin{array}{rcl} G & = & \{(\alpha = \beta \to \mathsf{Int}), (\beta = \mathsf{Int} \to \mathsf{Int})\} \\ \mathit{Failure} & = & \mathit{false} \\ \sigma & = & \emptyset. \end{array}$$

Choose and remove, say, $(\alpha = \beta \to \mathsf{Int})$ from G. We have $(\alpha = \beta \to \mathsf{Int})\sigma \equiv (\alpha = \beta \to \mathsf{Int})$, so we are in case (4). Change σ to $[\alpha := (\beta \to \mathsf{Int})]$. We now have $G = \{(\beta = \mathsf{Int} \to \mathsf{Int})\}$. Choose(!) and remove $(\beta = \mathsf{Int} \to \mathsf{Int})$ from G. We have $(\beta = \mathsf{Int} \to \mathsf{Int})\sigma \equiv (\beta = \mathsf{Int} \to \mathsf{Int})$, so we are in case (4). Change σ to

$$\begin{array}{l} [\alpha := (\beta \to \mathsf{Int})] \circ [\beta := (\mathsf{Int} \to \mathsf{Int})] = \\ [\alpha := (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int},\ \ \beta := (\mathsf{Int} \to \mathsf{Int})]. \end{array}$$

We now have $G = \emptyset$, so the algorithm outputs σ as a most general solution of the initial equations.

8 Typed Assembly Language

Here is the grammar for a typed assembly language. We use c to range over integer constants, and we use l to range over labels.

$$\begin{array}{lcll}
v & ::= & c \mid l & \text{(values)} \\
t & ::= & \mathsf{Int} \mid \Gamma & \text{(types)} \\
e & ::= & \text{mov } r_d, r_s; e \mid \text{set } r_d, v; e \mid \text{inc } r; e \mid \text{jmp } r & \text{(code)} \\
R & ::= & [r \mapsto v, \ldots] & \text{(register file)} \\
\Gamma & ::= & [r : t, \ldots] & \text{(register file type)} \\
H & ::= & [l \mapsto (\Gamma, e), \ldots] & \text{(code heap)} \\
\Psi & ::= & [l : \Gamma, \ldots] & \text{(heap type)} \\
\mathrm{s} & ::= & (H, R, e) & \text{(program state)}
\end{array}$$

We use $\lceil c \rceil$ to denote the integer represented by an integer constant c. A small-step operational semantics for the language is given by the reflexive, transitive closure of the relation $\rightarrow$:

$$\rightarrow \subseteq \text{ program state} \times \text{program state}$$

$$(H, R, \text{mov } r_d, r_s; e) \rightarrow (H, R[r_d \mapsto R(r_s)], e) \tag{11}$$

$$(H, R, \text{set } r_d, v; e) \rightarrow (H, R[r_d \mapsto v)], e) \tag{12}$$

$$(H, R, \text{inc } r; e) \rightarrow (H, R[r \mapsto c')], e) \tag{13}$$

$$\text{where } \lceil c' \rceil = \lceil c \rceil + 1, \text{ provided } R(r) = c.$$

$$(H, R, \text{jmp } r) \rightarrow (H, R, e) \tag{14}$$

$$\text{provided } R(r) = l \text{ and } H(l) = (\Gamma, e).$$

A program state s is *stuck* if there is no program state s' such that $s \rightarrow s'$. A program state s *goes wrong* if $\exists s' : s \rightarrow^* s'$ and s' is stuck.

We will use five forms of judgments and an ordering of register file types to describe what it means for a program state to be well typed.

Values:

$$\Psi \vdash c : \mathsf{Int} \tag{15}$$

$$\Psi \vdash l : \Gamma \quad (\Psi(l) = \Gamma) \tag{16}$$

Code:

$$\frac{\Psi, \Gamma[r_d : \Gamma(r_s)] \vdash e}{\Psi, \Gamma \vdash \text{mov } r_d, r_s; e} \tag{17}$$

$$\frac{\Psi \vdash v : t \quad \Psi, \Gamma[r_d : t] \vdash e}{\Psi, \Gamma \vdash \text{set } r_d, v; e} \tag{18}$$

$$\frac{\Gamma(r) = \mathsf{Int} \quad \Psi, \Gamma \vdash e}{\Psi, \Gamma \vdash \text{inc } r; e} \tag{19}$$

$$\frac{\Gamma(r) = \Gamma' \quad \Gamma \leq \Gamma'}{\Psi, \Gamma \vdash \text{jmp } r} \tag{20}$$

Register files:

$$\frac{\Psi \vdash v : t \quad \ldots}{\Psi, \Gamma \vdash [r \mapsto v, \ldots]} \quad (\Gamma(r) = t, \ldots) \tag{21}$$

Ordering of register file types:

$$[r_1 : t_1, \ldots, r_n : t_n, \ldots, r_{n+m} : t_{n+m}] \leq [r_1 : t_1, \ldots, r_n : t_n] \tag{22}$$

where $m \geq 0$.

Code heaps:

$$\frac{\Psi, \Gamma \vdash e \quad \ldots}{\Psi \vdash [l \mapsto (\Gamma, e), \ldots]} \quad (\Psi(l) = \Gamma, \ldots) \tag{23}$$

Program states:

$$\frac{\Psi \vdash H \quad \Psi, \Gamma \vdash R \quad \Psi, \Gamma \vdash e}{\vdash (H, R, e)} \tag{24}$$

A program state s is *well typed* if and only if $\vdash s$.
Exercise: prove that a well-typed program state cannot go wrong.