CS 565 October 6, 1997
Jens Palsberg Readings: Efficient Recursive Subtyping

Lecture Note 8:
Structural Subtyping and Recursive Types

Contents

1 Recursive Types **2**

2 Subtyping **5**

3 Efficient Recursive Subtyping **8**

4 Amadio and Cardelli's Original Algorithm **9**

5 The Fast Algorithm **10**

6 Example **11**

7 Terms **12**

8 The Type Ordering **14**

9 Basic Result **15**

10 Algorithm **16**

11 Tree Truncations **17**

12 Equivalence Result **18**

1 Recursive Types

Simple types are finite. When we draw a simple type as a tree, it is finite. One of the consequences is that we cannot type all λ-terms with simple types. For example, consider the λ-term $\lambda x.xx$. Like in Lecture Note 7, let us think of it as $\lambda x.x_1 x_2$. The constraints are:

$$
\begin{array}{rcl}
\langle x \rangle \to [\![x_1 x_2]\!] & = & [\![\lambda x.x_1 x_2]\!] \\
[\![x_1]\!] & = & [\![x_2]\!] \to [\![x_1 x_2]\!] \\
\langle x \rangle & = & [\![x_1]\!] \\
\langle x \rangle & = & [\![x_2]\!]
\end{array}
$$

Some unification gives:

$$
\begin{array}{rcl}
[\![x_1]\!] & = & [\![x_2]\!] \to [\![x_1 x_2]\!] \\
 & = & \langle x \rangle \to [\![x_1 x_2]\!] \\
 & = & [\![x_1]\!] \to [\![x_1 x_2]\!]
\end{array}
$$

In other words, the type for x_1 must be a function type which has itself as the domain type! This is not possible with a finite type, so from Theorem 1 of Lecture Note 7 we get that λ-term $\lambda x.xx$ is not typable.

Suppose now that we allow types to be infinite. We change the grammar of types to:

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

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

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

The notation $t[s/\alpha]$ is the notation for substitution that we studied in Lecture Note 3. 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. Draw it!

There is just one thing to be careful about: if a type contains a pattern of the form

$$\mu\alpha_1.\mu\alpha_2.\dots\mu\alpha_n.\alpha_1$$

then we cannot unfold it into something other than itself! Such a pattern is called a Black Hole and we will not consider it to be a type. In summary, if we generate something from the grammar above, and it does not contain a Black Hole, 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. (We are not going to study the proofs of these claims here.)

Let consider again the λ-term $\lambda x.xx$. With the type $u = \mu\alpha.(\alpha \to \mathsf{Int})$ it is easy to solve the system of constraints and to build a type derivation. Here is the type derivation:

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

In the left part of the tree we have $\{x : u\} \vdash u \to \mathsf{Int}$ 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:

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

Notice that for any λ-term E we have

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

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:

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

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

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

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:

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

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

Let us say that a λ-term with only abstractions, applications, and variables is a *pure* λ-term. Thus, a pure λ-term does not contain $\mathsf{0}$ or succ. Recursive types are sufficient to type all pure λ-terms. There is even a type which types *all* pure λ-terms! Consider the type $B = \mu\alpha.(\alpha \to \alpha)$. The tree for this type expression is a full, binary, infinite tree with $\to$ as the label in all the nodes. Draw it! By unfolding once we see that $B = B \to B$.

Let us say that a type environment is a t-environment if all the type variables in its domain are mapped to t.

Theorem 2 If A is a B-environment which is defined (at least) for all free variables in the λ-term E, then $A \vdash E : B$.

It is easy to prove this by induction on the structure of E.

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. The good news is that Theorem 1 also holds for recursive types: same type rules, same constraints, but recursive types instead of simple types.

2 Subtyping

Motivation: typing functions in object-oriented languages.

$$t \ ::= \top \mid t \to t \qquad \text{(types)}$$

Type ordering: $t \leq \top$ for any type t.

One extra typing rule: subsumption.

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

Partial Types

$$t ::= \top \mid t \to t \qquad \text{(types)}$$

$t \leq \top$ for any type t.

Add the usual rule for ordering function spaces:

$s \to t \leq s' \to t'$ if and only if $s' \leq s$ and $t \leq t'$.

Typed terms are strongly normalizing [Wand 92].

Every normal form is typable
[Palsberg, Inf. Proc. Letters].

Type inference:

- [Thatte ICALP'88]:
  Semi-decidable
- [O'Keefe, Wand ESOP'92]:
  Exponential time
- [Kozen, Palsberg, Schwartzbach FOCS'92]:
  Cubic time

Subtyping and Recursive Types

Found in Amber and Quest.

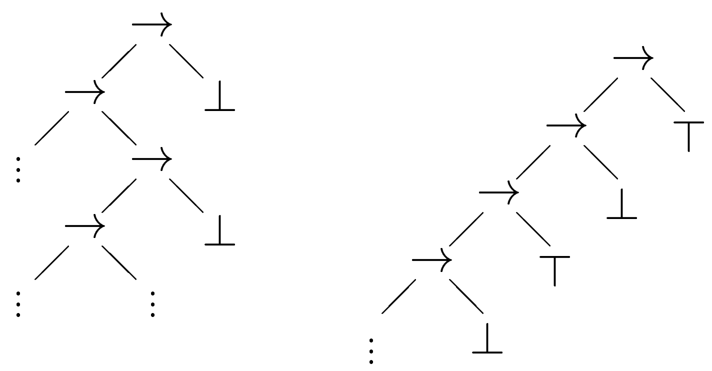
$$t \; ::= v \mid \bot \mid \top \mid t \to t \mid \mu v.t \qquad \text{(types)}$$

Studied by Amadio and Cardelli [POPL'91].
$\bot \leq t$ for any type t

$$(v \leq v' \Rightarrow t \leq t') \Rightarrow (\mu v.t \leq \mu v'.t')$$

Representing types as labeled trees:

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



3 Efficient Recursive Subtyping

The Result

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

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

Amadio and Cardelli (POPL'91): Exponential time.

Kozen, Palsberg, and Schwartzbach (POPL'93): $O(n^2)$ time.

4 Amadio and Cardelli's Original Algorithm

Problem: Given two types s and t, is $s \leq t$?

Definition
Two trees are ordered if all their finite truncations are ordered.

Concrete representation of trees involving back-pointers.

Explicit construction of a binary proof tree of polynomial depth.

Time complexity: exponential.

5 The Fast Algorithm

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

Theorem
The new definition is equivalent to Amadio and Cardelli's.

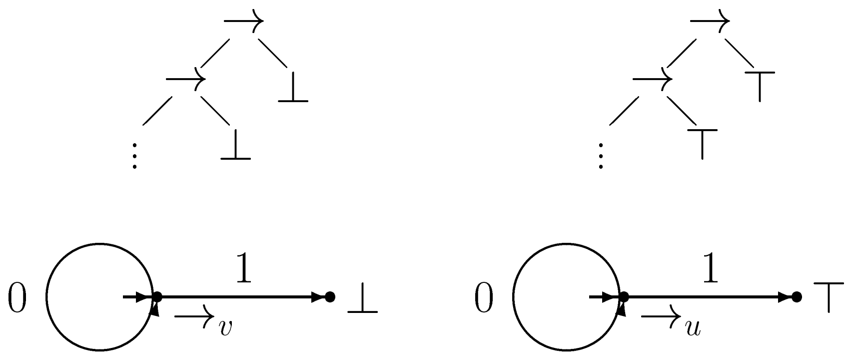
Representation of trees as *term automata*.

A certain product automaton accepts all counterexamples.

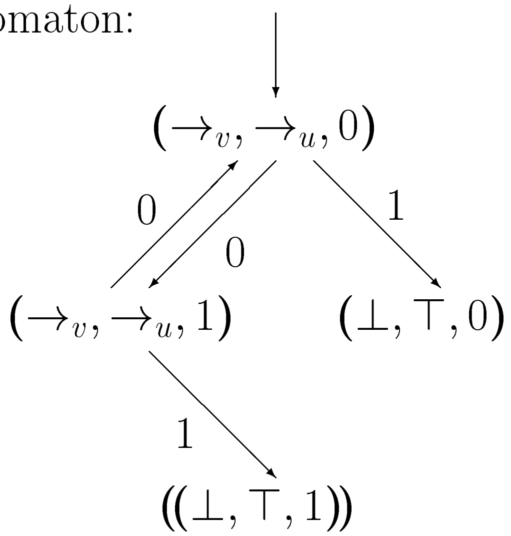
Time complexity: $O(n^2)$ time.

6 Example

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



Product automaton:



Ordering: $\bot \leq \to \leq \top$.

Perhaps reverse? 0 means "no", 1 means "yes".

7 Terms

Intuition: a term is a labeled tree.

Σ, a finite ranked alphabet.

In our application: $\Sigma = \{\bot, \rightarrow, \top\}$.

A term over Σ is a partial function

$$t : \omega^* \rightarrow \Sigma$$

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

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

The set of all terms over Σ is denoted T_Σ.

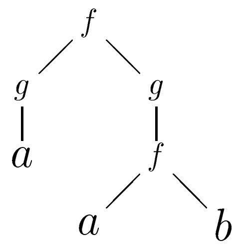
A term t is *finite* if its domain $\mathcal{D}(t)$ is a finite set. We denote the set of finite terms over Σ by F_Σ.

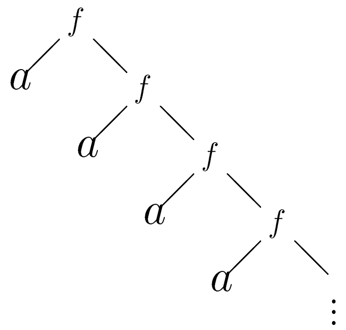
Subterms

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.

Example terms:





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 described by a finite set of equations involving the μ operator.

8 The Type Ordering

Types are terms over $\Sigma = \{\bot, \to, \top\}$.
Over this signature, every $\mathcal{D}(t) \subseteq \{0,1\}^*$.
The order $\leq_{\mathrm{FIN}}$ is the smallest binary relation on F_Σ 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
$s \to t \leq_{\mathrm{FIN}} s' \to t'$.

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

$$\bot \ \leq_0 \ \to \ \leq_0 \ \top$$

on Σ, and let $\leq_1$ be its reverse

$$\top \ \leq_1 \ \to \ \leq_1 \ \bot \ .$$

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

9 Basic Result

The relation $\leq$ is a partial order on T_Σ, and agrees with $\leq_{\mathrm{FIN}}$ on F_Σ. In particular, for any s, t, s', t',

(i) $\bot \leq t \leq \top$

(ii) $t \leq \bot$ if and only if $t = \bot$

(iii) $\top \leq t$ if and only if $t = \top$

(iv) $s \to t \leq s' \to t'$ if and only if $s' \leq s$ and $t \leq t'$.

Recall that $s \not\leq t$ iff
$\{\alpha \in \mathcal{D}(s) \cap \mathcal{D}(t) \mid s(\alpha) \not\leq_{\pi\alpha} t(\alpha)\}$ is nonempty.

Theorem
A certain product automaton accepts

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

10 Algorithm

Input: Two types s and t

Output: Is $s \leq t$?

1. Construct the term automata for s and t (linear time).
2. Construct the product automaton as sketched before (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: $O(n^2)$.

11 Tree Truncations

The *level-k truncations* $t|_k$, $t\lceil_k$, and $t\lfloor_k$ of the term t are defined by

$$\mathcal{D}(t|_k) = \mathcal{D}(t\lceil_k) = \mathcal{D}(t\lfloor_k) = \{\alpha \in \mathcal{D}(t) \mid |\alpha| \leq k\}$$

and

$$
\begin{aligned}
t|_k(\alpha) &= \begin{cases} t(\alpha), & |\alpha| < k \\ \top, & |\alpha| = k \end{cases} \\
t\lceil_k(\alpha) &= \begin{cases} t(\alpha), & |\alpha| < k \\ \lceil\alpha\rceil, & |\alpha| = k \end{cases} \\
t\lfloor_k(\alpha) &= \begin{cases} t(\alpha), & |\alpha| < k \\ \lfloor\alpha\rfloor, & |\alpha| = k \end{cases}
\end{aligned}
$$

where

$$\lceil\alpha\rceil = \begin{cases} \bot, & \alpha \text{ even} \\ \top, & \alpha \text{ odd}, \end{cases} \qquad \lfloor\alpha\rfloor = \begin{cases} \top, & \alpha \text{ even} \\ \bot, & \alpha \text{ odd}. \end{cases}$$

Notice that $t\lceil_k \leq_{\mathrm{FIN}} t\lceil_{k+1}$

12 Equivalence Result

Lemma:
For any $k \geq 0$ and terms s and t, the following are equivalent:

- $s|_k \ \leq_{\mathrm{FIN}} \ t|_k$
- $s\lceil_k \leq_{\mathrm{FIN}} \ t\lceil_k$
- $s\lfloor_k \leq_{\mathrm{FIN}} \ t\lfloor_k$.

Definition:
Amadio-Cardelli's order $\leq_{\mathrm{AC}}$ is defined as

$$s \leq_{\mathrm{AC}} t \iff s|_k \leq_{\mathrm{FIN}} t|_k \text{ for all } k \geq 0 \ .$$

Theorem: The relation $\leq$ agrees with $\leq_{\mathrm{AC}}$.