1 Introduction

In this lecture we study the inference problem for simple types in the presence of subtyping and recursive types. For simplicity, we restrict our attention to a simple equirecursive type system consist only of the function space constructor $\to$ and a universal type $\top$. This system is known in the literature as *partial types*.

Finite partial types were originally introduced by Thatte in 1988 as a means of typing certain objects that are not typable with simple types, such as heterogeneous lists and persistent data.

In this lecture we will outline an approach that gives an $O(n^3)$ algorithm for type inference. As in the previous lecture, the algorithm is automata-theoretic. It constructs a certain finite automaton with $O(n^2)$ states representing a canonical solution to a given set of type constraints. The canonical solution to the system is just the regular language accepted by the automaton, where we represent types as binary trees and binary trees as prefix-closed sets of strings over a two-letter alphabet. The construction works equally well for finite or recursive types.

The canonical solution always exists, but it may not be finite; however, since it is contained in all other solutions, we can check for the existence of a finite solution by checking whether the canonical solution is finite. Thus the typability question reduces essentially to the finiteness problem for regular sets.

It is known that the type inference problem for partial types is PTIME-hard; thus it is PTIME-complete. It can also be shown that every λ-term with a partial type is strongly normalizing and that every λ-term in normal form has a partial type.

This result is quite long, so for brevity we will omit proofs.

1.1 Partial Types

Formally, partial types comprise a partially ordered set $(T, \leq)$, where T is the set of well-formed terms over the constant symbol $\top$ and the binary type constructor $\to$, and $\leq$ is the partial order defined inductively by the following rules:

$$\tau \leq \top \qquad\qquad \frac{\sigma' \leq \sigma \qquad \tau \leq \tau'}{\sigma \to \tau \leq \sigma' \to \tau'}$$

Intuitively, the type constructor $\to$ represents the usual function space constructor, and $\top$ is a *universal type* that includes every other type. The partial order $\leq$ can be thought of as *type inclusion* or *coercion*; that is, $\sigma \leq \tau$ if it is possible to cast type σ to type τ.

Clause (ii) in the definition of $\leq$ models the fact that a function with domain σ and range τ can be coerced to a function with domain σ' and range τ' provided σ' can be coerced to σ and τ can be coerced to τ'; thus the subtype order on functions is covariant in the range and contravariant in the domain. That $\leq$ is contravariant in the domain is the main source of difficulty in type inference in the presence of subtyping.

1.2 Typing Rules

If e is a λ-term, τ is a partial type, and Γ is a type environment, that is, a partial function assigning types to variables, then the judgment $\Gamma \vdash e : \tau$ means that e has the partial type τ in the environment Γ. This

holds when the judgment is derivable using the rules

$$\Gamma \vdash x : \Gamma(x) \qquad \frac{\Gamma[\sigma/x] \vdash e : \tau}{\Gamma \vdash \lambda x.\, e : \sigma \to \tau} \qquad \frac{\Gamma \vdash e : \sigma \to \tau \qquad \Gamma \vdash e' : \sigma}{\Gamma \vdash e\ e' : \tau} \qquad \frac{\Gamma \vdash e : \sigma \qquad \sigma \leq \tau}{\Gamma \vdash e : \tau}$$

The first three rules are the usual rules for simple types and the last rule is the rule of *subsumption*.

More λ-terms are typable with partial types than with simple types. For example, the term $\lambda f.\, (fK(fI))$, where $K = \lambda xy.\, x$ and $I = \lambda z.\, z$, has partial type $(\top \to \top \to \top) \to \top$, but no simple type.

2 From Rules to Constraints

Given a λ-term e, the type inference question can be rephrased in terms of solving a system of type constraints. Assume that e has been α-converted so that all bound variables are distinct. Let X be the set of λ-variables x occurring in e, and let Y be a set of variables disjoint from X consisting of one variable $[\![d]\!]$ for each occurrence of a subterm d of e. (The notation $[\![d]\!]$ is ambiguous because there may be more than one occurrence of d in e. However, it will always be clear from context which occurrence is meant.) We generate the following system of inequalities over $X \cup Y$:

- for every occurrence in e of a subterm of the form $\lambda x.\, d$, the inequality

$$x \to [\![d]\!] \ \leq\ [\![\lambda x.\, d]\!];$$

- for every occurrence in e of a subterm of the form $(d\ c)$, the inequality

$$[\![d]\!] \ \leq\ [\![c]\!] \to [\![d\ c]\!];$$

- for every occurrence in e of a λ-variable x, the inequality

$$x \ \leq\ [\![x]\!].$$

Denote by $C(e)$ the system of constraints generated from e in this fashion.

Let Γ be a type environment assigning a type to each λ-variable occurring freely in e. If Δ is a function assigning a type to each variable in $X \cup Y$, we say that Δ *extends* Γ if Γ and Δ agree on the domain of Γ.

The following theorem asserts that the solutions of $C(e)$ over T correspond exactly to the possible type annotations of e.

Theorem 1. *The judgment $\Gamma \vdash e : \tau$ is derivable if and only if there exists a solution Δ of $C(e)$ extending Γ such that $\Delta([\![e]\!]) = \tau$. In particular, if e is closed, then e is typable with type τ if and only if there exists a solution Δ of $C(e)$ such that $\Delta([\![e]\!]) = \tau$.*

3 From Types to Trees

In the previous lecture, we represented types as labeled trees, but here the picture is even simpler: a node is either an internal node, in which case its label must be $\to$, or it is a leaf, in which case its label must be $\top$. Thus we can dispense with the labeling altogether and represent types as binary trees, or subsets $\sigma \subseteq \{0,1\}^*$ that are

- nonempty,

- closed under prefix, and
- binary, in the sense that for all x, $x0 \in \sigma$ iff $x1 \in \sigma$.

As before, the *parity* of a string $x \in \{0,1\}^*$ is the number mod 2 of 0's in x. A string x is *even* (respectively, *odd*) if its parity is 0 (respectively, 1). A tree is *finite* if it is finite as a set of strings. A *path* in a tree σ is a maximal subset of σ linearly ordered by the prefix relation. By König's Lemma, a tree is finite iff it has no infinite paths. An element $x \in \sigma$ is a *leaf* of σ if it is not a proper prefix of any other element of σ.

For trees σ, τ and $x \in \{0,1\}^*$, define

$$\sigma \cdot \tau \stackrel{\triangle}{=} \{\varepsilon\} \cup \{0x \mid x \in \sigma\} \cup \{1y \mid y \in \tau\} \qquad\qquad \sigma_x \stackrel{\triangle}{=} \{y \mid xy \in \sigma\}.$$

Thus $\sigma \cdot \tau$ is the tree with left subtree σ and right subtree τ, and σ_x is the subtree of σ at position x if $x \in \sigma$.

The partial order on types defined in Section 1.1 can now be reformulated. For $\sigma, \tau \in T$, define $\sigma \leq \tau$ if both of the following conditions hold for any x:

(i) if x is an even leaf of σ, then $x1 \notin \tau$;

(ii) if x is an odd leaf of τ, then $x1 \notin \sigma$.

Lemma 2. *The relation $\leq$ is a partial order on trees, and agrees with the order $\leq$ on finite types defined in Section 1.1.*

4 From Constraints to Graphs

Instead of systems of type constraints involving type variables, we consider a more general notion of a *constraint graph*.

A *constraint graph* is a directed graph $G = (S, 0, 1, \leq)$ consisting of a set of nodes S and three sets of directed edges $0, 1, \leq$. We write $s \stackrel{0}{\to} t$ to indicate that the pair (s,t) is in the edge set 0, and similarly $s \stackrel{1}{\to} t$, $s \stackrel{\leq}{\to} t$. A constraint graph must satisfy the properties:

- any node has at most one outgoing 0 edge and at most one outgoing 1 edge;
- a node has an outgoing 0 edge if and only if it has an outgoing 1 edge.

A *solution* for G is any map $h : S \to T$ such that

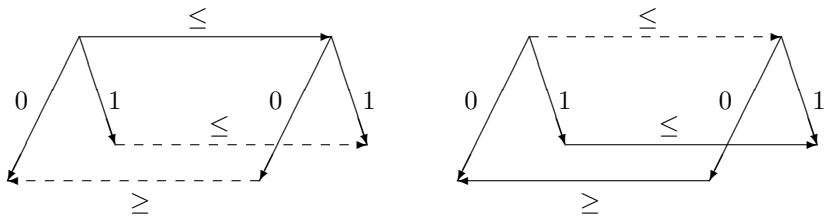
(i) if $u \stackrel{0}{\to} v$ and $u \stackrel{1}{\to} w$, then $h(u) = h(v) \cdot h(w)$;

(ii) if $u \stackrel{\leq}{\to} v$, then $h(u) \leq h(v)$.

The solution h is *finite* if $h(s)$ is a finite set for all s.

A system of type constraints as described in §2 gives rise to a constraint graph by associating a unique node with every subexpression occurring in the system of constraints, defining 0 and 1 edges from an occurrence of an expression to its left and right subexpressions, and defining $\leq$ edges for the inequalities.

A constraint graph is *closed* if the edge relation $\leq$ is reflexive, transitive, and closed under the following two rules which say that the dashed edges exist whenever the solid ones do:



The *closure* of a constraint graph G is the smallest closed graph containing G as a subgraph.

Lemma 3. *A constraint graph and its closure have the same set of solutions.*

5 From Graphs to Automata

In this section we define an automaton M that will be used to characterize the canonical solution of a given constraint graph G.

Let a closed constraint graph $G = (S, 0, 1, \leq)$ be given. The automaton M is defined as follows. The input alphabet of M is $\{0, 1\}$. The states of M are $S^2 \cup S^1 \cup S^0$. States in S^2 are written (s, t), those in S^1 are written (s), and the unique state in S^0 is written $(\ )$. The transitions are defined as follows.

$$
\begin{array}{ll}
(u,v) \xrightarrow{\varepsilon} (u,v') & \text{if } v \overset{\leq}{\to} v' \text{ in } G \\
(u,v) \xrightarrow{\varepsilon} (u',v) & \text{if } u' \overset{\leq}{\to} u \text{ in } G \\
(u,v) \xrightarrow{1} (u',v') & \text{if } u \xrightarrow{1} u' \text{ and } v \xrightarrow{1} v' \text{ in } G \\
(u,v) \xrightarrow{0} (v',u') & \text{if } u \xrightarrow{0} u' \text{ and } v \xrightarrow{0} v' \text{ in } G \\
(u,v) \xrightarrow{\varepsilon} (v) & \text{always} \\
(v) \xrightarrow{\varepsilon} (v') & \text{if } v \overset{\leq}{\to} v' \text{ in } G \\
(v) \xrightarrow{1} (v') & \text{if } v \xrightarrow{1} v' \text{ in } G \\
(v) \xrightarrow{0} (\ ) & \text{if } v \xrightarrow{0} v' \text{ in } G
\end{array}
$$

If p and q are states of M and $\alpha \in \{0,1\}^*$, we write $p \xrightarrow{\alpha} q$ if the automaton can move from state p to state q under input α, including possible ε-transitions.

The automaton M_s is the automaton M with start state (s, s). All states are accept states; thus the language accepted by M_s is the set of strings α for which there exists a state p such that $(s, s) \xrightarrow{\alpha} p$. We denote this language by $\mathcal{L}(s)$.

Informally, we can think of the automaton M_s as follows. We start with two pebbles, one green and one red, on the node s of the constraint graph G. We can move the green pebble forward along a $\leq$ edge at any time, and we can move the red pebble backward along a $\leq$ edge at any time. We can move both pebbles simultaneously along 1 edges leading out of the nodes they occupy. We can also move them simultaneously along outgoing 0 edges, but in the latter case we switch their colors. At any time, we can elect to remove the red pebble; thereafter, we can move the green pebble forward along $\leq$ or 1 edges as often as we like, and forward along a 0 edge once, at which point the pebble must be removed. The sequence of 0's and 1's that were seen gives a string in $\mathcal{L}(s)$, and all strings in $\mathcal{L}(s)$ are obtained in this way.

The intuition motivating the definition of M is that we want to identify the conditions that require a path to exist in any solution. Thus $\mathcal{L}(s)$ is the set of α that *must* be there; this intuition formalized in Lemma 4. Once we identify this set, we can show that it is a solution itself.

Lemma 4. *If $h : S \to T$ is any solution and $(s, s) \xrightarrow{\alpha} p$, then $\alpha \in h(s)$. Moreover,*

(i) *if $p = (u, v)$ then $h(u) \leq h(s)_\alpha \leq h(v)$;*

(ii) *if $p = (v)$ then $h(s)_\alpha \leq h(v)$.*

6 Main Result

The following theorem says that $\mathcal{L}(s)$ gives the canonical solution of G.

Theorem 5. *The sets $\mathcal{L}(s)$ are trees, and the function $\mathcal{L} : S \to T$ is a solution of G. Moreover, if $h : S \to T$ is any other solution, then $\mathcal{L}(s) \subseteq h(s)$ for any s.*

As observed, recursive types are just regular trees. The canonical solution we have constructed, although possibly infinite, is a regular tree. Thus we can also treat the type inference problem for recursive types. Specifically, given a λ-term, we construct the corresponding constraint graph and automaton M. Every subterm corresponds to a node s in the constraint graph, and its setwise-minimal type annotation is represented by the language $\mathcal{L}(s)$.

7 An Algorithm

By Theorem 5, there exists a finite solution if and only if the canonical solution is finite. To determine this, we need only check whether any $\mathcal{L}(s)$ contains an infinite path. We first form the constraint graph, then close it; this gives a graph with n vertices and $O(n^2)$ edges. This can be done in time $O(n^3)$. We then form the automaton M, which has $n^2 + n + 1$ states but only $O(n^3)$ transitions, at most $O(n)$ from each state. We then check for a cycle with at least one non-ε transition reachable from some (s, s). This can be done in linear time in the size of the graph using depth-first search. The entire algorithm requires time $O(n^3)$.

Thus the type inference problem for partial types with or without recursive types is solvable in time $O(n^3)$.