CS 565 October 6, 1997
Jens Palsberg Readings: Nothing else.

Lecture Note 7:

Simple Types for the Core of Scheme

Contents

1 Introduction

Specification is about *What*, and implementation is about *How*. When dealing with a piece of data, it is sufficient to know the specification. The specification is a contract between the user and the implementer. The implementer promises to implement at least what the specification says, and the user promises to do at most what is allowed by the specification. For example, suppose that a specification of a piece of data says:

> The data f is a function which maps integers to integers.

Notice that the specification only promises that if we apply f to an integer, then we will get an integer back. Thus, the contract is clear: the implementer is done with the work when f can map integers to integers, and the user can then use f to do exactly that. A type is a simple form of specification. This note presents a type system for the λ-calculus, and a way of doing type inference, based on solving type constraints by first-order unification.

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:

$$E ::= x \mid \lambda x.E \mid E_1 E_2 \mid \mathsf{0} \mid \mathsf{succ}\ E$$

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

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

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

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

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

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, i.e., a partial function assigning types to variables, 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 using the following five rules:

$$A \vdash \mathsf{0} : \mathsf{Int} \tag{1}$$

$$\frac{A \vdash E : \mathsf{Int}}{A \vdash \mathsf{succ}\ E : \mathsf{Int}} \tag{2}$$

$$A \vdash x : t \quad (\text{provided } A(x) = t) \tag{3}$$

$$\frac{A[x \leftarrow s] \vdash E : t}{A \vdash \lambda x.E : s \rightarrow t} \tag{4}$$

$$\frac{A \vdash E : s \rightarrow t \quad A \vdash F : s}{A \vdash EF : t} \tag{5}$$

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:

$$\frac{\dfrac{\{x : \mathsf{Int}\} \vdash x : \mathsf{Int}}{\{x : \mathsf{Int}\} \vdash \mathsf{succ}\ x : \mathsf{Int}}}{\emptyset \vdash \lambda x.\mathsf{succ}\ x : \mathsf{Int} \rightarrow \mathsf{Int}}$$

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

$$\frac{\{x : \mathsf{Int}\} \vdash x : \mathsf{Int}}{\emptyset \vdash \lambda x.x : \mathsf{Int} \rightarrow \mathsf{Int}}$$

The apply function:

$$\frac{\dfrac{\dfrac{\{f : s \rightarrow t,\ x : s\} \vdash f : s \rightarrow t \qquad \{f : s \rightarrow t,\ x : s\} \vdash x : s}{\{f : s \rightarrow t,\ x : s\} \vdash fx : t}}{\{f : s \rightarrow t\} \vdash \lambda x.fx : s \rightarrow t}}{\emptyset \vdash \lambda f.\lambda x.fx : (s \rightarrow t) \rightarrow (s \rightarrow t)}$$

The K combinator:

$$\frac{\dfrac{\{x : s,\ y : t\} \vdash x : s}{\{x : s\} \vdash \lambda y.x : t \rightarrow s}}{\emptyset \vdash \lambda x.\lambda y.x : s \rightarrow (t \rightarrow s)}$$

The following λ-term does not have a type:

$$\lambda f.(f(\lambda y.y))(f\mathsf{0})$$

The problem is that f is applied both to a function and to $\mathsf{0}$. Let us try to build a type derivation. First, $\lambda f.(f(\lambda y.y))(f\mathsf{0})$ is an abstraction, and the corresponding rule says that it must be given a function type. Let us give it the type $s \to t$, where we have not yet committed to particular choices of s and t. Next we have to find a type for the application $(f(\lambda y.y))(f\mathsf{0})$. The corresponding rule says that $(f(\lambda y.y))$ must be given a function type. The range of that function type must be t, so let us choose $r \to t$, where we have not yet committed to r. Let us summarize what we have so far:

$$\dfrac{\dfrac{\{f : s\} \vdash f(\lambda y.y) : r \to t \qquad \{f : s\} \vdash f\mathsf{0} : r}{\{f : s\} \vdash (f(\lambda y.y))(f\mathsf{0}) : t}}{\emptyset \vdash \lambda f.(f(\lambda y.y))(f\mathsf{0}) : s \to t}$$

Now we have to type each of the applications $f(\lambda y.y)$ and $f\mathsf{0}$. Surely, $\lambda y.y$ has a type of the form $u \to u$, so we are trying to build a derivation of the form:

$$\dfrac{\dfrac{\dfrac{\{f : s\} \vdash f : s \quad \{f : s\} \vdash \lambda y.y : u \to u}{\{f : s\} \vdash f(\lambda y.y) : r \to t} \qquad \dfrac{\{f : s\} \vdash f : s \quad \{f : s\} \vdash \mathsf{0} : \mathsf{Int}}{\{f : s\} \vdash f\mathsf{0} : r}}{\{f : s\} \vdash (f(\lambda y.y))(f\mathsf{0}) : t}}{\emptyset \vdash \lambda f.(f(\lambda y.y))(f\mathsf{0}) : s \to t}$$

If we first look at the left part of this derivation, then we see that it only works if

$$s = (u \to u) \to (r \to t) \ .$$

If we next look at the right part of the derivation, then we see that it only works if

$$s = \mathsf{Int} \ \to r \ .$$

In both cases, we get this from the rule for application. But now we have a problem, because we need

$$(u \to u) \to (r \to t) \ = \ s \ = \ \mathsf{Int} \ \to r$$

and this is only possible if $(u \to u) = \mathsf{Int}$ which is not the case. We conclude that $\lambda f.(f(\lambda y.y))(f\mathsf{0})$ has no simple type.

3 Constraints

How can we in general decide if a λ-term has a simple type or not? The answer is: use constraints! Our strategy is to

1. generate type constraints from the program text, and
2. solve the constraints, or decide that no solution exists.

The constraints will be designed such that the program has a simple type if and only if the constraint system is satisfiable.

Given a λ-term E, we now describe how to generate a certain constraint system. Assume that E has been α-converted so that all bound variables are distinct. Let X_E be a set of type variables consisting of one type variable $\langle x \rangle$ for each λ-variable x occurring in E, and let Y_E be a set of variables disjoint from X_E consisting of one variable $[\![F]\!]$ for each occurrence of a subterm F of E. (The notation $[\![F]\!]$ is ambiguous because there may be more than one occurrence of F in E. However, it will always be clear from context which occurrence is meant.) The following constraint system uses $X_E \cup Y_E$ as type variables:

- for every occurrence in E of a subterm of the form $\mathsf{0}$, the equality

$$\mathsf{Int} \quad = \quad [\![\mathsf{0}]\!] \ ;$$

- for every occurrence in E of a subterm of the form $\mathsf{succ}\ F$, the two equalities

$$\begin{aligned} \mathsf{Int} &\quad = \quad [\![\mathsf{succ}\ F]\!] \\ [\![F]\!] &\quad = \quad \mathsf{Int} \ ; \end{aligned}$$

- for every occurrence in E of a subterm of the form $\lambda x.F$, the equality

$$\langle x \rangle \rightarrow [\![F]\!] \quad = \quad [\![\lambda x.F]\!] \ ;$$

- for every occurrence in E of a subterm of the form GH, the equality

$$[\![G]\!] \quad = \quad [\![H]\!] \rightarrow [\![GH]\!] \ ;$$

- for every occurrence in E of a λ-variable x, the equality

$$\langle x \rangle \quad = \quad [\![x]\!] \ .$$

Denote by $T(E)$ the system of constraints generated from E in this fashion. All constraints are of one of the three forms:

$$\begin{aligned} v &= \mathsf{Int} \\ v &= v' \\ v &= v' \to v'' \end{aligned}$$

A set of such constraints is *satisfiable* if there is a mapping from type variables to types such that all constraints are satisfied. To check satisfiability, we apply first-order unification. This doable in $O(n)$ time, where n is the size of the constraint system.

Theorem 1 For every λ-term E we have that E has a simple type if and only if $T(E)$ is satisfiable.

We will not study the proof here, but informally it is easy to see that the theorem makes sense: for each construct in the language, the constraints directly reflect relationships imposed by the corresponding type rule.

The good news is that we now can check typability by first generating constraints, and then try to solve them.

Let us take a look again at the untypable λ-term $\lambda f.(f(\lambda y.y))(f\mathsf{0})$. Let us label the two references to f as f_1 and f_2. Thus we think of the term as $\lambda f.(f_1(\lambda y.y))(f_2\mathsf{0})$, where the labels are just our way of distinguishing the two places in the program. The constraints for this λ-term are:

$$\begin{aligned} \langle f \rangle \to [\![(f_1(\lambda y.y))(f_2\mathsf{0})]\!] &= [\![\lambda f.(f_1(\lambda y.y))(f_2\mathsf{0})]\!] \\ [\![f_1(\lambda y.y)]\!] &= [\![f\mathsf{0}]\!] \to [\![(f_1(\lambda y.y))(f_2\mathsf{0})]\!] \\ [\![f_1]\!] &= [\![\lambda y.y]\!] \to [\![f_1(\lambda y.y)]\!] \\ [\![f_2]\!] &= [\![\mathsf{0}]\!] \to [\![f\mathsf{0}]\!] \\ \langle y \rangle \to [\![y]\!] &= [\![\lambda y.y]\!] \\ [\![\mathsf{0}]\!] &= \mathsf{Int} \\ \langle f \rangle &= [\![f_1]\!] \\ \langle f \rangle &= [\![f_2]\!] \\ \langle y \rangle &= [\![y]\!] \end{aligned}$$

Let us then try to solve these constraints by doing some unification:

$$\begin{aligned}
\mathsf{Int} \to [\![f\,\mathsf{0}]\!] &= [\![\mathsf{0}]\!] \to [\![f\,\mathsf{0}]\!] \\
&= [\![f_2]\!] \\
&= \langle f \rangle \\
&= [\![f_1]\!] \\
&= [\![\lambda y.y]\!] \to [\![f_1(\lambda y.y)]\!] \\
&= (\langle y \rangle \to [\![y]\!]) \to [\![f_1(\lambda y.y)]\!]
\end{aligned}$$

Unification gives that

$$\mathsf{Int} \;=\; \langle y \rangle \to [\![y]\!]$$

follows from the above constraints. But this is not satisfiable! By applying Theorem 1 we then get that $\lambda f.(f(\lambda y.y))(f\mathsf{0})$ is not typable. Further inspection reveals that the reason for the unsatisfiability is exactly the same as why we could not build a type derivation for $\lambda f.(f(\lambda y.y))(f\mathsf{0})$ in the previous section. The advantage of using constraints is that now we know for sure that it is impossible to build a type derivation, thanks to Theorem 1.

4 First-Order Unification

In general, the problem of first-order unification is as follows. Given a constraint

$$s = t$$

where s and t are finite trees, possibly with variables at the leaves, find a substitution θ which maps variables to finite trees such that

$$\theta(s) = \theta(t)\ .$$

We say that the substitution θ *unifies* s and t, and that θ *solves* $s = t$, and that θ *satisfies* $s = t$. The problem can be generalized to a family of constraints. In that case, the problem is to produce one substitution which solves all of the constraints. One can solve this unification problem in $O(n)$

time, where n is the size of the constraint 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.

In our setting, a tree is either a type variable, Int, or of the form $s \to t$ where s and t are trees. If we are given the constraint system

$$\begin{aligned} v &= w \to \mathsf{Int} \\ w &= \mathsf{Int} \to \mathsf{Int} \ , \end{aligned}$$

then the substitution θ where

$$\begin{aligned} \theta(v) &= (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int} \\ \theta(w) &= \mathsf{Int} \to \mathsf{Int} \end{aligned}$$

is a solution. If we are given the constraint system

$$v = v \to \mathsf{Int} \ ,$$

then there is no solution because of the cycle.

The forms of constraints we consider can be unsolvable for two reasons:

- there is a cycle in the constraint system such that we can obtain:

$$\begin{aligned} v &= (\ldots\ v\ \ldots) \to \ldots \quad \text{or} \\ v &= \ldots \to (\ldots\ v\ \ldots) \ , \text{ or} \end{aligned}$$

- we can deduce $\mathsf{Int} = s \to t$.

Here is a simple algorithm for unifying two trees s and t. Initialize θ to be the identity function. Repeat the following step until $\theta(s) = \theta(t)$ or failure has occurred:

- Find the leftmost symbol in s such that the corresponding symbol in t is different. Let A_s and A_t be the trees in $\theta(s)$ and $\theta(t)$ which are rooted at the symbols just found.

  1. If neither A_s not A_t are variables, or one is a variable occurring in the other, then halt with failure.
  2. Otherwise, suppose it A_s which is a variable. Update θ such that $\theta(A_s) = A_t$.