CS 565
Jens Palsberg

October 30, 1996
Readings: Nothing else.

Lecture Note 10:

Type Rules and Constraints

In Lecture Note 2, we studied the separation of interface and implementation. 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.

In the light of the above discussion of interface and implementation, let us reconsider the notion of type from Lecture Note 8 and *OOTS*. That notion of type is "a set of class names," so each type is of the form

$$\{C_1, \ldots, C_n\}$$

where $C_1, \ldots, C_n$ are class names. A class yields the implementation for its objects so the above types are "sets of implementations". If a variable has the above type, then the type says

> The content of the variable is an object (or nil), and the object is implemented by one of the classes $C_1, \ldots, C_n$.

We conclude that this notion of type is much closer to implementation than to specification. For that reason, some people call them "concrete types."

If a piece of data is an object, then all we can do with it is to pass it around, store it in variables, and send messages to it. We can pass around and store all data, but an object will only respond to certain messages. What is the specification of an object? The specification should contain:

- a description of what methods are available,
- for each method,
  - for each parameter of the method, a specification that actual parameters should satisfy, and
  - a specification that will be satisfied by the result of the method.

For example, we could write the specification

> **interface**
>   **method** m(**S**) **returns T**
>   **method** p(**Int**) **returns Int**
> **end**

If a variable has this type, the the type says

> The content of the variable is an object (or nil), and the object contains methods m and p. Moreover, if one sends a message m with an argument that has interface **S**, then the result will have interface **T**. Finally, if one sends a message p with an argument that has interface **Int**, then the result will have interface **Int**.

Such interfaces contain only specification, no implementation. We could call them "abstract types" to distinguish them from the concrete types.

Abstract types are the classical idea of type. Henceforth we will simply refer to them as type. We will now embark on a journey into the land of types. One of the purposes is to be able to understand strengths and weaknesses of Java. As a warm-up to study object-oriented ideas, we will first study types for functional languages.

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:

$$E \ ::= x \mid \lambda x.E \mid E_1E_2 \mid \mathsf{0} \mid \mathsf{succ}\ E$$

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:

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

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

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

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

$$A \vdash E : t$$

means that E has the type t in the environment A. Formally, this holds when the judgement is derivable using the following five rules:

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

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

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

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

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

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

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

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

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

The apply function:

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

The K combinator:

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

The following λ-term does not have a 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.

Constraints

How can we in general decide if a λ-term has a simple type or not? The answer is: use constraints! The constraints are different from the ones we considered in Lecture 8 and *OOTS*.

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

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

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

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

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

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

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

$$[\![G]\!] = [\![H]\!] \to [\![GH]\!] \; ;$$

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

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

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

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

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. This is exactly the same recipe that we used for object-oriented programs (but with completely different types).

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:

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

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.

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 the previous section, let us think of it as $\lambda x.x_1x_2$. The constraints are:

$$\begin{aligned}
\langle x\rangle \to [\![x_1x_2]\!] &= [\![\lambda x.x_1x_2]\!] \\
[\![x_1]\!] &= [\![x_2]\!] \to [\![x_1x_2]\!] \\
\langle x\rangle &= [\![x_1]\!] \\
\langle x\rangle &= [\![x_2]\!]
\end{aligned}$$

Some unification gives:

$$\begin{aligned}
[\![x_1]\!] &= [\![x_2]\!] \to [\![x_1x_2]\!] \\
&= \langle x\rangle \to [\![x_1x_2]\!] \\
&= [\![x_1]\!] \to [\![x_1x_2]\!]
\end{aligned}$$

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 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:

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

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

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

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:

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

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 wierd 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.