# Principal Types

sk and dbtucker

October 27, 2000

## Printing equational types

In the previous lecture, we saw how to compute an equational type for an expression. Given some equational type $\tau \setminus E$, how do we turn it into a printable type? Since $\tau$ may contain type variables that are constrained by equations in $E$, we'll substitute for those type variables until we can no longer apply constraints. More formally, the algorithm is:

Repeat until there are no more possible substitutions:

1. Pick a type variable $\alpha$ in $\tau$.
2. If there exists $\alpha = \beta$ or $\beta = \alpha$ in $E$, then replace $\alpha$ with $\beta$ in $\tau$ if:
   (a) $\beta$ is not a type variable, or
   (b) $\beta$ is a type variable and $\alpha < \beta$ (where $<$ is a lexicographic ordering of type variables).

The purpose of the lexicographic ordering is to ensure that we don't continually substitute two variables for each other. For example, consider the equational type $\alpha \setminus \{\alpha = \beta\}$. Without the ordering condition, we would substitute $\beta$ for $\alpha$, then $\alpha$ for $\beta$, and so on; the process would not terminate.

Even with this guard, there are still cases where this algorithm doesn't terminate. For example, take the equational type $\alpha \setminus \{\alpha = \mathsf{num} \to \alpha\}$. In every iteration of the algorithm, we substitute $\mathsf{num} \to \alpha$ for $\alpha$, and the type just keeps getting larger. Do any expressions actually generate types that look like this one? Certainly—try to type check the infinite loop $((\lambda\ (x)\ (x\ x))\ (\lambda\ (x)\ (x\ x)))$, or the Y combinator, and you will see similar constraints.

This problem arises from cyclic dependencies in the constraint equations. We need to detect cycles before running the above algorithm (we will make this part of step 3—see the previous lecture). For an equational type $\tau \setminus E$, we will check for cycles by constructing a graph that represents the constraint equations:

$$\begin{aligned} \text{nodes of graph} &\equiv \text{type variables} \\ \text{edges of graph} &\equiv \{(u, v) \mid u = \ldots v \ldots \in E\} \end{aligned}$$

Now we use DFS to determine whether there are any cycles in the graph; if so, we reject the program, since we have no way of expressing its type.

A fair question is whether cyclic types are bad. We give three reasons for rejecting them:

1. it's harder to prove type soundness with graph-based types
2. it's harder to print such types (perhaps using fixpoints)
3. it's harder to prove the next property...

## Principal types

What property does the type printed by the algorithm have? Say our program is the identity function, $Id \equiv (\lambda\ (x)\ x)$. Does it have type $\alpha \to \alpha$? How about $\mathsf{num} \to \mathsf{num}$? Or $(\mathsf{num} \to \mathsf{bool}) \to (\mathsf{num} \to \mathsf{bool})$?

What if our algorithm assigns $Id$ the type $\mathsf{num} \to \mathsf{num}$? Then this expression still type checks:

$$((\lambda\ (x)\ x)\ \ 42)$$

But now the following expression *doesn't* type check:

$$((\lambda\ (x)\ x)\ \ \text{true})$$

We really want our algorithm to give $Id$ the type $\alpha \to \alpha$, because that type is "more general" than $\mathsf{num} \to \mathsf{num}$ or any other type. We can define this notion more precisely using contexts. Recall that a context is just some computation parameterized over a hole. For example, given the context:

$$C \equiv (\lambda\ (\blacksquare)\ (+\ \ 1\ \ (*\ \ 2\ \ (-\ \ \blacksquare\ \ 4))))$$

we can fill in the hole with 3, and get:

$$C[3] = ((\lambda\ (\blacksquare)\ (+\ \ 1\ \ (*\ \ 2\ \ (-\ \ \blacksquare\ \ 4))))\ \ 3) = (+\ \ 1\ \ (*\ \ 2\ \ (-\ \ 3\ \ 4)))$$

We can use contexts in the type checking phase as well. The notation $C[e : \tau]$ means that the hole in $C$ is filled in with the expression $e$, and the type of $e$ is constrained to be $\tau$.

**Definition:** Say an expression $e$ has type $\tau$. Let $\tau'$ be some other type such that $e$ also has type $\tau'$. We want the following property to hold:

for all contexts $C$, if $C[e : \tau']$ type checks, then $C[e : \tau]$ type checks

If this property holds, we say $\tau$ is the *principal type* of $e$.

Do we care that $C[e : \tau]$ and $C[e : \tau']$ have the same type? No. Consider this simple example:

$$
\begin{array}{ll}
e \equiv Id & \tau \equiv \alpha \to \alpha \\
C \equiv (\lambda\ (\blacksquare)\ \blacksquare) & \tau' \equiv \mathsf{num} \to \mathsf{num}
\end{array}
$$

Then $C[e : \tau'] : \mathsf{num} \to \mathsf{num}$, but $C[e : \tau] : \alpha \to \alpha$. Both types are legal, since the resulting expression will just be an identity function. We didn't require that the two expressions have the same type, since they very well might not (as in this case). As long as $C[e : \tau]$ type checks whenever $C[e : \tau']$ does, $\tau$ will allow at least as many contexts to type check as $\tau'$ will.

There is an important theorem relating our algorithm and principal types:

**Theorem:** Our algorithm prints out the principal type of an expression!

This remarkable theorem was discovered and proved independently by Hindley and Milner. Hence, this process of deriving principal types is known as Hindley-Milner type inference.