Type Inference:
Unification, Parameterized Types,
Underconstrained Systems, and Principal Types

sk and dbtucker

2002-11-13

1 Solving Type Constraints

To solve type constraints, we turn to a classic algorithm: *unification*. Unification consumes a set of constraints and either

- identifies inconsistencies amongst the constraints, or
- generates a *substitution* that represents the solution of the constraints.

A substitution is a mapping from names to constants. In our universe, inconsistencies indicate type errors, the constants are the base types (such as number and boolean), and the names are identifiers representing the values of individual boxes (thus $\boxed{4}$ is a funny notation for an identifier that represents the type of the expression labeled 4).

The unification algorithm is extremely simple. Begin with an empty substitution. Push all the constraints onto a stack. If the stack is empty, return the substitution; otherwise, pop the constraint $X = Y$ off the stack:

1. If X and Y are identical constants (base types), do nothing.
2. If X and Y are identical variables, do nothing.
3. If X is a variable, replace all occurrences of X by Y both on the stack and in the substitution, and add $X \mapsto Y$ to the substitution.
4. If Y is a variable, replace all occurrences of Y by X both on the stack and in the substitution, and add $Y \mapsto X$ to the substitution.
5. If X is of the form $C(X_1, \ldots, X_n)$ for some constructor C (in the type language, only $\rightarrow$ so far), and Y is of the form $C(Y_1, \ldots, Y_n)$ (i.e., it has the same constructor), then push $X_i = Y_i$ for all $1 \leq i \leq n$ onto the stack.
6. Otherwise, X and Y do not unify. Report an error.

Let us argue that this algorithm terminates. On every iteration of the main loop, we pop a constraint off the stack. In some cases, however, we push new constraints on. The *size* of each of these constraints is, however, smaller than the constraint just popped. Therefore, the total number of iterations cannot be greater than the sum of the sizes of subterms of every term in the initial constraint set. Since each iteration pops, the stack must eventually become empty.

Problem What is this algorithm's complexity?

2 Example of Unification at Work

First we will consider an extremely simple example:

$\boxed{1}$($\boxed{2}$(**lambda** (*x*) *x*)
$\boxed{3}$7)

This generates the following constraints:

$$[\![\boxed{2}]\!] = [\![\boxed{3}]\!] \rightarrow [\![\boxed{1}]\!]$$
$$[\![\boxed{2}]\!] = [\![x]\!] \rightarrow [\![x]\!]$$
$$[\![\boxed{3}]\!] = \mathsf{number}$$

Our algorithm works as follows.

Action	Stack	Substitution
Initialize	$[\![\boxed{2}]\!] = [\![\boxed{3}]\!] \rightarrow [\![\boxed{1}]\!]$ $[\![\boxed{2}]\!] = [\![x]\!] \rightarrow [\![x]\!]$ $[\![\boxed{3}]\!] = \mathsf{number}$	empty
Step 3	$[\![\boxed{3}]\!] \rightarrow [\![\boxed{1}]\!] = [\![x]\!] \rightarrow [\![x]\!]$ $[\![\boxed{3}]\!] = \mathsf{number}$	$[\![\boxed{2}]\!] \mapsto [\![\boxed{3}]\!] \rightarrow [\![\boxed{1}]\!]$
Step 5	$[\![\boxed{3}]\!] = [\![x]\!]$ $[\![\boxed{1}]\!] = [\![x]\!]$ $[\![\boxed{3}]\!] = \mathsf{number}$	$[\![\boxed{2}]\!] \mapsto [\![\boxed{3}]\!] \rightarrow [\![\boxed{1}]\!]$
Step 3	$[\![\boxed{1}]\!] = [\![x]\!]$ $[\![x]\!] = \mathsf{number}$	$[\![\boxed{2}]\!] \mapsto [\![x]\!] \rightarrow [\![\boxed{1}]\!]$ $[\![\boxed{3}]\!] \mapsto [\![x]\!]$
Step 3	$[\![x]\!] = \mathsf{number}$	$[\![\boxed{2}]\!] \mapsto [\![x]\!] \rightarrow [\![x]\!]$ $[\![\boxed{3}]\!] \mapsto [\![x]\!]$ $[\![\boxed{1}]\!] \mapsto [\![x]\!]$
Step 3	empty	$[\![\boxed{2}]\!] \mapsto \mathsf{number} \rightarrow \mathsf{number}$ $[\![\boxed{3}]\!] \mapsto \mathsf{number}$ $[\![\boxed{1}]\!] \mapsto \mathsf{number}$ $[\![x]\!] \mapsto \mathsf{number}$

At this point, we have solutions for all the sub-expressions *and* we know that the constraint set is consistent.

Writing these in detail in notes format is painstaking, but it's usually easy to do them on paper by just crossing out old values when performing a substitution. Be sure to work through our examples for more practice with unification!

3 Parameterized Types

In the presentation of unification above, we saw only one kind of type constructor, for functions: $\rightarrow$. A regular programming language will typically have many more constuctors. One common built-in constructor is for list types. That is, it is common to think of lists as parameterized over their content, thus yielding list(number), list(symbol), list(list(number)) and even list(number $\rightarrow$ symbol) and so on. A small change in the type rules, and identifying list as one of the type constructors for the unification algorithm, suffice for typing *unityped* lists.

What about other constructors, especially for so-called *container* types (such as trees, queues, etc.)? It is common in languages like ML and Haskell to permit the datatype definition to introduce *parameterized* types, such as tree(number). Again, the datatype mechanism simply needs to generate type-sensitive information, and notify the unification algorithm of the presence of new constuctors.

4 The "Occurs" Check

Suppose we generate the following type constraint:

$$\mathsf{list}(\mathsf{list}(\alpha)) = \mathsf{list}(\alpha)$$

We must then unify α with $\mathsf{list}(\alpha)$, which is impossible. A traditional unification algorithm therefore checks in Steps 3 and 4 whether the identifier about to be (re)bound in the substitution *occurs* in the term that will take its place. If the identifier does occur, the unifier halts with an error.[1] Otherwise, the algorithm proceeds as before.

Puzzle Write a program that will generate the above constraint!

5 Underconstrained Systems

We have seen in our previous lecture that if the system has too many competing constraints—for instance, forcing an identifier to have both type number and $\mathsf{boolean}$—there can be no satisfying type assignment, so the system should halt with an error. We saw the indication of this error informally in the previous lecture. Working through that example with the unification algorithm should confirm that the algorithm does indeed detect the error (in Step 6).

But what if the system is *under*-constrained? This is interesting, because some of the program identifiers never get assigned a type! In a procedure such as *map*, for instance:

```
(define (map f l)
  (cond
    [(empty? l) empty]
    [(cons? l) (cons (f (first l))
                     (map f (rest l)))]))
```

Working through the constraint generation and unification algorithm for examples such as this is educational. Briefly, however, because the list does not have a fixed type (with parameterized types, we no longer need to program solely on nlists), it does not place any constraint on the function. In turn, because the function has no type, it places no constraint on the kind of constructed list. Well, it's not true that they place *no* constraint: they simply impose a consistency requirement. Working through the steps, we get a type for *map* of this form:

$$\forall \alpha, \beta . (\alpha \rightarrow \beta) \times \mathsf{list}(\alpha) \rightarrow \mathsf{list}(\beta)$$

This is the same type we obtained through explicit parametric polymorphism ... except that the unification algorithm has found it for us *automatically*!

6 Principal Types

The type generated by the Hindley-Milner[2] system has a particularly pleasing property: it is a *principal* type. A principal type has the following property. For a term t, consider a type τ. τ is a principal type of t if, for any other type τ' that types t, there exists a substitution (perhaps empty) that, when applied to τ, yields τ'.

There are a few ways of re-phrasing the above:

- The Hindley-Milner type system infers the "most general" type for a term.
- The type generated by the Hindler-Milner type system imposes fewest constraints on the program's behavior. In particular, it imposes all constraints necessary for type soundness, and no more.

[1] This is not the only reasonable behavior! It is possible to define *fixed-point types*, which are solutions to the circular constraint equations above. This topic is, however, beyond the scope of this course.

[2] Named for Roger Hindley and Robin Milner, who independently discovered such a type system in the late 1960s and early 1970s.