30.5 Solving Type Constraints

30.5.1 The Unification Algorithm

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 associates each identifier (for which we are trying to solve) with a constant or another identifier. Our identifiers represent the types of expressions (thus $\boxed{4}$ is a funny notation for an identifier that represents the type of the expression labeled 4). In our universe, inconsistencies indicate type errors, and the constants are terms in the type language (such as number and number $\rightarrow$ boolean).

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 identifiers, do nothing.
2. If X is an identifier, replace all occurrences of X by Y both on the stack and in the substitution, and add $X \mapsto Y$ to the substitution.
3. If Y is an identifier, replace all occurrences of Y by X both on the stack and in the substitution, and add $Y \mapsto X$ to the substitution.
4. If X is of the form $C(X_1, \ldots, X_n)$ for some constructor C,[4] 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.
5. Otherwise, X and Y do not unify. Report an error.

Does this this algorithm terminate? On every iteration of the main loop, it pops 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 the initial constraint set. The stack must therefore eventually become empty.

Exercise 30.5.1 *What are the space and time complexity of this algorithm?*

30.5.2 Example of Unification at Work

Let's consider the following example:

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

[4] In our type language, the type constructors are $\rightarrow$ and the base types (which are constructors of arity zero). More on this in Section 30.5.3.

This generates the following constraints:

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

$$[\![\boxed{2}]\!] = [\![x]\!] \rightarrow [\![x]\!]$$

$$[\![\boxed{3}]\!] = \mathit{number}$$

The unification algorithm works as follows:

Action	Stack	Substitution
Initialize	$[\![\boxed{2}]\!] = [\![\boxed{3}]\!] \rightarrow [\![\boxed{1}]\!]$ $[\![\boxed{2}]\!] = [\![x]\!] \rightarrow [\![x]\!]$ $[\![\boxed{3}]\!] = \mathit{number}$	empty
Step 2	$[\![\boxed{3}]\!] \rightarrow [\![\boxed{1}]\!] = [\![x]\!] \rightarrow [\![x]\!]$ $[\![\boxed{3}]\!] = \mathit{number}$	$[\![\boxed{2}]\!] \mapsto [\![\boxed{3}]\!] \rightarrow [\![\boxed{1}]\!]$
Step 4	$[\![\boxed{3}]\!] = [\![x]\!]$ $[\![\boxed{1}]\!] = [\![x]\!]$ $[\![\boxed{3}]\!] = \mathit{number}$	$[\![\boxed{2}]\!] \mapsto [\![\boxed{3}]\!] \rightarrow [\![\boxed{1}]\!]$
Step 2	$[\![\boxed{1}]\!] = [\![x]\!]$ $[\![x]\!] = \mathit{number}$	$[\![\boxed{2}]\!] \mapsto [\![x]\!] \rightarrow [\![\boxed{1}]\!]$ $[\![\boxed{3}]\!] \mapsto [\![x]\!]$
Step 2	$[\![x]\!] = \mathit{number}$	$[\![\boxed{2}]\!] \mapsto [\![x]\!] \rightarrow [\![x]\!]$ $[\![\boxed{3}]\!] \mapsto [\![x]\!]$ $[\![\boxed{1}]\!] \mapsto [\![x]\!]$
Step 2	empty	$[\![\boxed{2}]\!] \mapsto \mathit{number} \rightarrow \mathit{number}$ $[\![\boxed{3}]\!] \mapsto \mathit{number}$ $[\![\boxed{1}]\!] \mapsto \mathit{number}$ $[\![x]\!] \mapsto \mathit{number}$

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

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

30.5.3 Parameterized Types

In the presentation of unification above, we saw only one type constructor with positive arity: $\rightarrow$. A regular programming language will typically have many more constuctors. A common source of parameterized types is *containers*: lists, trees, queues, stacks, and so forth. For instance, it is common to think of lists as parameterized over their content, thus yielding list(number), list(symbol), list(list(number)), list(number $\rightarrow$ symbol) and so on. Identifying list as one of the type constructors for the unification algorithm suffices for typing *unityped* lists.

30.5.4 The "Occurs" Check

Suppose we generate the following type constraint:

$$list([\![x]\!]) = list(list([\![x]\!]))$$

By Step 4, we should push $[\![x]\!] = \mathsf{list}([\![x]\!])$ onto the stack. Eventually, obeying to Step 2, we will add the mapping $[\![x]\!] \mapsto \mathsf{list}([\![x]\!])$ to the substition but, in the process, attempt to replace all instances of $[\![x]\!]$ in the substitution with the right-hand side, which does not terminate.

This is a familiar problem: we saw it earlier when trying to define substitution in the presence of recursion. Because these are problematic to handle, a traditional unification algorithm checks in Steps 2 and 3 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.[5] Otherwise, the algorithm proceeds as before.

Exercise 30.5.2 *Write a program that will generate the above constraint!*

30.6 Underconstrained Systems

We have seen earlier that if the system has too many competing constraints—for instance, forcing an identifier to have both type number and boolean—there can be no satisfying type assignment, so the system should halt with an error. We saw this informally earlier; Step 5 of the unification algorithm confirms that the implementation matches this informal behavior.

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)))]))
```

This is an instructive example. Solving the constraints reveals that there is no constraint on the type passed as a parameter to the list type constructor. Working through the steps, we get a type for *map* of this form:

$$(\alpha \rightarrow \beta) \times list(\alpha) \rightarrow list(\beta)$$

where α and β are unconstrained type identifiers. This is the same type we obtained through explicit parametric polymorphism... except that the unification algorithm has found it for us *automatically*!

[5] 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 text.

30.7 Principal Types

The type generated by this Hindley-Milner[6] system has a particularly pleasing property: it is a *principal* type. What does that mean? 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 constraints necessary for type soundness, but no more.

From a software engineering perspective, this is very attractive: it means a programmer could not possibly annotate a procedure with a more general type than the type inference algorithm would derive. Thus, using the algorithm instead of performing manual annotation will not restrict the reusability of code, and may even increase it (because the programmer's annotation may mistakenly overconstrain the type). Of course, there are other good reasons for manual annotation, such as documentation and readability, so a good programming style will mix annotation and inference judiciously.

[6] Named for Roger Hindley and Robin Milner, who independently discovered this in the late 1960s and early 1970s.