22 *Type Reconstruction*

The typechecking algorithms for the calculi we have seen so far all depend on explicit type annotations—in particular, they require that lambda-abstractions be annotated with their argument types. In this chapter, we develop a more powerful *type reconstruction* algorithm, capable of calculating a *principal type* for a term in which some or all of these annotations are left unspecified. Related algorithms lie at the heart of languages like ML and Haskell.

Combining type reconstruction with other language features is often a somewhat delicate matter. In particular, both records and subtyping pose significant challenges. To keep things simple, we consider type reconstruction here only for simple types; §22.8 gives some starting points for further reading on other combinations.

22.1 Type Variables and Substitutions

In some of the calculi in previous chapters, we have assumed that the set of types includes an infinite collection of *uninterpreted* base types (§11.1). Unlike interpreted base types such as `Bool` and `Nat`, these types come with no operations for introducing or eliminating terms; intuitively, they are just placeholders for some particular types whose exact identities we do not care about. In this chapter, we will be asking questions like "if we instantiate the placeholder `X` in the term `t` with the concrete type `Bool`, do we obtain a typable term?" In other words, we will treat our uninterpreted base types as *type variables,* which can be *substituted* or *instantiated* with other types.

For the technical development in this chapter, it is convenient to separate the operation of substituting types for type variables into two parts: *describing* a mapping σ from type variables to types, called a *type substitution,* and

---

The system studied in this chapter is the simply typed lambda-calculus (Figure 9-1) with booleans (8-1), numbers (8-2), and an infinite collection of base types (11-1). The corresponding OCaml implementations are `recon` and `fullrecon`.

applying this mapping to a particular type T to obtain an instance $\sigma\mathsf{T}$. For example, we might define $\sigma = [\mathsf{X} \mapsto \mathtt{Bool}]$ and then apply σ to the type $\mathsf{X}\to\mathsf{X}$ to obtain $\sigma(\mathsf{X}\to\mathsf{X}) = \mathtt{Bool}\to\mathtt{Bool}$.

22.1.1 DEFINITION: Formally, a *type substitution* (or just *substitution,* when it's clear that we're talking about types) is a finite mapping from type variables to types. For example, we write $[\mathsf{X} \mapsto \mathsf{T}, \mathsf{Y} \mapsto \mathsf{U}]$ for the substitution that associates X with T and Y with U. We write $dom(\sigma)$ for the set of type variables appearing on the left-hand sides of pairs in σ, and $range(\sigma)$ for the set of types appearing on the right-hand sides. Note that the same variable may occur in both the domain and the range of a substitution. Like term substitutions, the intention in such cases is that all the clauses of the substitution are applied simultaneously; for example, $[\mathsf{X} \mapsto \mathtt{Bool}, \mathsf{Y} \mapsto \mathsf{X}\to\mathsf{X}]$ maps X to $\mathtt{Bool}$ and Y to $\mathsf{X}\to\mathsf{X}$, not $\mathtt{Bool}\to\mathtt{Bool}$.

Application of a substitution to a type is defined in the obvious way:

$$\begin{array}{lcl}
\sigma(\mathsf{X}) & = & \begin{cases} \mathsf{T} & \text{if } (\mathsf{X} \mapsto \mathsf{T}) \in \sigma \\ \mathsf{X} & \text{if } \mathsf{X} \text{ is not in the domain of } \sigma \end{cases} \\
\sigma(\mathtt{Nat}) & = & \mathtt{Nat} \\
\sigma(\mathtt{Bool}) & = & \mathtt{Bool} \\
\sigma(\mathsf{T}_1\to\mathsf{T}_2) & = & \sigma\mathsf{T}_1 \to \sigma\mathsf{T}_2
\end{array}$$

Note that we do not need to make any special provisions to avoid variable capture during type substitution, because there are no constructs in the language of type expressions that *bind* type variables. (We'll get to these in Chapter 23.)

Type substitution is extended pointwise to contexts by defining

$$\sigma(\mathsf{x}_1:\mathsf{T}_1,\ldots,\mathsf{x}_n:\mathsf{T}_n) = (\mathsf{x}_1:\sigma\mathsf{T}_1,\ldots,\mathsf{x}_n:\sigma\mathsf{T}_n).$$

Similarly, a substitution is applied to a term $\mathtt{t}$ by applying it to all types appearing in annotations in $\mathtt{t}$.

If σ and γ are substitutions, we write $\sigma \circ \gamma$ for the substitution formed by composing them as follows:

$$\sigma \circ \gamma = \left[\begin{array}{ll} \mathsf{X} \mapsto \sigma(\mathsf{T}) & \text{for each } (\mathsf{X} \mapsto \mathsf{T}) \in \gamma \\ \mathsf{X} \mapsto \mathsf{T} & \text{for each } (\mathsf{X} \mapsto \mathsf{T}) \in \sigma \text{ with } \mathsf{X} \notin dom(\gamma) \end{array} \right]$$

Note that $(\sigma \circ \gamma)\mathsf{S} = \sigma(\gamma\mathsf{S})$. □

A crucial property of type substitutions is that they preserve the validity of typing statements: if a term involving variables is well typed, then so are all of its substitution instances.

22.1.2 THEOREM [PRESERVATION OF TYPING UNDER TYPE SUBSTITUTION]: If σ is any type substitution and $\Gamma \vdash \mathtt{t} : \mathsf{T}$, then $\sigma\Gamma \vdash \sigma\mathtt{t} : \sigma\mathsf{T}$. □

Proof: Straightforward induction on typing derivations. □

22.2 Two Views of Type Variables

Suppose that `t` is a term containing type variables and Γ is an associated context (possibly also containing type variables). There are two quite different questions that we can ask about `t`:

1. "Are *all* substitution instances of `t` well typed?" That is, for every σ, do we have $\sigma\Gamma \vdash \sigma\,\texttt{t} : \texttt{T}$ for some `T`?

2. "Is *some* substitution instance of `t` well typed?" That is, can we find a σ such that $\sigma\Gamma \vdash \sigma\,\texttt{t} : \texttt{T}$ for some `T`?

According to the first view, type variables should be *held abstract* during typechecking, thus ensuring that a well-typed term will behave properly no matter what concrete types are later substituted for its type variables. For example, the term

```
λf:X→X. λa:X. f (f a);
```

has type `(X→X)→X→X`, and, whenever we replace `X` by a concrete type `T`, the instance

```
λf:T→T. λa:T. f (f a);
```

is well typed. Holding type variables abstract in this way leads us to *parametric polymorphism,* where type variables are used to encode the fact that a term can be used in many concrete contexts with different concrete types. We will return to parametric polymorphism later in this chapter (in §22.7) and, in more depth, in Chapter 23.

On the second view, the original term `t` may not even be well typed; what we want to know is whether it can be *instantiated* to a well typed term by choosing appropriate values for some of its type variables. For example, the term

```
λf:Y. λa:X. f (f a);
```

is not typable as it stands, but if we replace `Y` by `Nat→Nat` and `X` by `Nat`, we obtain

```
λf:Nat→Nat. λa:Nat. f (f a);
```

of type `(Nat→Nat)→Nat→Nat`. Or, if we simply replace `Y` by `X→X`, we obtain the term

```
λf:X→X. λa:X. f (f a);
```

which is well typed even though it contains variables. Indeed, this term is a *most general* instance of `λf:Y. λa:X. f (f a)`, in the sense that it makes the smallest commitment about the values of type variables that yields a well-typed term.

Looking for valid instantiations of type variables leads to the idea of *type reconstruction* (sometimes called *type inference*), in which the compiler helps fill in type information that has been left out by the programmer. In the limit, we may, as in ML, allow the programmer to leave out *all* type annotations and write in the syntax of the bare, untyped lambda-calculus. During parsing, we annotate each bare lambda-abstraction `λx.t` with a type variable, `λx:X.t`, choosing `X` to be different from the type variables on all the other abstractions in the program. We then perform type reconstruction to find the most general values for all these variables that make the term typecheck. (This story becomes a little more complicated in the presence of ML's let-polymorphism; we return to this in §22.6 and §22.7.)

To formalize type reconstruction, we will need a concise way of talking about the possible ways that type variables can be substituted by types, in a term and its associated context, to obtain a valid typing statement.[1]

22.2.1 DEFINITION: Let Γ be a context and `t` a term. A *solution* for $(\Gamma, \mathtt{t})$ is a pair $(\sigma, \mathtt{T})$ such that $\sigma\Gamma \vdash \sigma\mathtt{t} : \mathtt{T}$. □

22.2.2 EXAMPLE: Let Γ = `f:X, a:Y` and `t` = `f a`. Then

```
([X ↦ Y→Nat], Nat)          ([X ↦ Y→Z], Z)
([X ↦ Y→Z, Z ↦ Nat], Z)     ([X ↦ Y→Nat→Nat], Nat→Nat)
([X ↦ Nat→Nat, Y ↦ Nat], Nat→Nat)
```

are all solutions for $(\Gamma, \mathtt{t})$. □

22.2.3 EXERCISE [⋆ ↛]: Find three different solutions for the term

```
λx:X. λy:Y. λz:Z. (x z) (y z).
```

in the empty context. □

1. There are other ways of setting up these basic definitions. One is to use a general mechanism called *existential unificands,* due to Kirchner and Jouannaud (1990), instead of all the individual freshness conditions in the constraint generation rules in Figure 22-1. Another possible improvement, employed by Rémy (1992a, 1992b, long version, 1998, Chapter 5), is to treat typing statements themselves as unificands; we begin with a triple $(\Gamma, \mathtt{t}, \mathtt{T})$, where all three components may contain type variables, and look for substitutions σ such that $\sigma\Gamma \vdash \sigma(\mathtt{t}) : \sigma(\mathtt{T})$, i.e., substitutions that *unify* the schematic typing statement $\Gamma \vdash \mathtt{t} : \mathtt{T}$.

22.3 Constraint-Based Typing

We now present an algorithm that, given a term $\mathtt{t}$ and a context Γ, calculates a set of constraints—equations between type expressions (possibly involving type variables)—that must be satisfied by any solution for $(\Gamma, \mathtt{t})$. The intuition behind this algorithm is essentially the same as the ordinary typechecking algorithm; the only difference is that, instead of *checking* constraints, it simply *records* them for later consideration. For example, when presented with an application $\mathtt{t}_1\ \mathtt{t}_2$ with $\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1$ and $\Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2$, rather than checking that $\mathtt{t}_1$ has the form $\mathtt{T}_2 \to \mathtt{T}_{12}$ and returning $\mathtt{T}_{12}$ as the type of the application, it instead chooses a fresh type variable $\mathtt{X}$, records the constraint $\mathtt{T}_1 = \mathtt{T}_2 \to \mathtt{X}$, and returns $\mathtt{X}$ as the type of the application.

22.3.1 DEFINITION: A *constraint set* C is a set of equations $\{\mathtt{S}_i = \mathtt{T}_i \,^{i \in 1..n}\}$. A substitution σ is said to *unify* an equation $\mathtt{S} = \mathtt{T}$ if the substitution instances $\sigma\mathtt{S}$ and $\sigma\mathtt{T}$ are identical. We say that σ *unifies* (or *satisfies*) C if it unifies every equation in C. □

22.3.2 DEFINITION: The *constraint typing relation* $\Gamma \vdash \mathtt{t} : \mathtt{T} \mid_X C$ is defined by the rules in Figure 22-1. Informally, $\Gamma \vdash \mathtt{t} : \mathtt{T} \mid_X C$ can be read "term $\mathtt{t}$ has type $\mathtt{T}$ under assumptions Γ whenever constraints C are satisfied." In rule T-APP, we write $FV(\mathtt{T})$ for the set of all type variables mentioned in $\mathtt{T}$.

The X subscripts are used to track the type variables introduced in each subderivation and make sure that the fresh variables created in different subderivations are actually distinct. On a first reading of the rules, it may be helpful to ignore these subscripts and all the premises involving them. On the next reading, observe that these annotations and premises ensure two things. First, whenever a type variable is chosen by the final rule in some derivation, it must be different from any variables chosen in subderivations. Second, whenever a rule involves two or more subderivations, the sets of variables chosen by these subderivations must be disjoint. Also, note that these conditions never prevent us from building *some* derivation for a given term; they merely prevent us from building a derivation in which the same variable is used "fresh" in two different places. Since there is an infinite supply of type variable names, we can always find a way of satisfying the freshness requirements.

When read from bottom to top, the constraint typing rules determine a straightforward procedure that, given Γ and $\mathtt{t}$, calculates $\mathtt{T}$ and C (and X) such that $\Gamma \vdash \mathtt{t} : \mathtt{T} \mid_X C$. However, unlike the ordinary typing algorithm for the simply typed lambda-calculus, this one never fails, in the sense that for every Γ and $\mathtt{t}$ there are always some $\mathtt{T}$ and C such that $\Gamma \vdash \mathtt{t} : \mathtt{T} \mid_X C$, and moreover that $\mathtt{T}$ and C are uniquely determined by Γ and $\mathtt{t}$. (Strictly

$$\frac{\mathtt{x}:\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T} \mid_{\varnothing} \{\}} \quad \text{(CT-VAR)}$$

$$\frac{\Gamma, \mathtt{x}:\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2 \mid_{\mathcal{X}} C}{\Gamma \vdash \lambda\mathtt{x}:\mathtt{T}_1.\mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2 \mid_{\mathcal{X}} C} \quad \text{(CT-ABS)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \mid_{\mathcal{X}_1} C_1 \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2 \mid_{\mathcal{X}_2} C_2 \\ \mathcal{X}_1 \cap \mathcal{X}_2 = \mathcal{X}_1 \cap FV(\mathtt{T}_2) = \mathcal{X}_2 \cap FV(\mathtt{T}_1) = \varnothing \\ \mathtt{X} \notin \mathcal{X}_1, \mathcal{X}_2, \mathtt{T}_1, \mathtt{T}_2, C_1, C_2, \Gamma, \mathtt{t}_1, \text{ or } \mathtt{t}_2 \\ C' = C_1 \cup C_2 \cup \{\mathtt{T}_1 = \mathtt{T}_2 \to \mathtt{X}\}\end{array}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{X} \mid_{\mathcal{X}_1 \cup \mathcal{X}_2 \cup \{\mathtt{X}\}} C'} \quad \text{(CT-APP)}$$

$$\Gamma \vdash \mathtt{0} : \mathtt{Nat} \mid_{\varnothing} \{\} \quad \text{(CT-ZERO)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T} \mid_{\mathcal{X}} C \qquad C' = C \cup \{\mathtt{T} = \mathtt{Nat}\}}{\Gamma \vdash \mathtt{succ}\ \mathtt{t}_1 : \mathtt{Nat} \mid_{\mathcal{X}} C'} \quad \text{(CT-SUCC)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T} \mid_{\mathcal{X}} C \qquad C' = C \cup \{\mathtt{T} = \mathtt{Nat}\}}{\Gamma \vdash \mathtt{pred}\ \mathtt{t}_1 : \mathtt{Nat} \mid_{\mathcal{X}} C'} \quad \text{(CT-PRED)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T} \mid_{\mathcal{X}} C \qquad C' = C \cup \{\mathtt{T} = \mathtt{Nat}\}}{\Gamma \vdash \mathtt{iszero}\ \mathtt{t}_1 : \mathtt{Bool} \mid_{\mathcal{X}} C'} \quad \text{(CT-ISZERO)}$$

$$\Gamma \vdash \mathtt{true} : \mathtt{Bool} \mid_{\varnothing} \{\} \quad \text{(CT-TRUE)}$$

$$\Gamma \vdash \mathtt{false} : \mathtt{Bool} \mid_{\varnothing} \{\} \quad \text{(CT-FALSE)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \mid_{\mathcal{X}_1} C_1 \\ \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2 \mid_{\mathcal{X}_2} C_2 \qquad \Gamma \vdash \mathtt{t}_3 : \mathtt{T}_3 \mid_{\mathcal{X}_3} C_3 \\ \mathcal{X}_1, \mathcal{X}_2, \mathcal{X}_3 \text{ nonoverlapping} \\ C' = C_1 \cup C_2 \cup C_3 \cup \{\mathtt{T}_1 = \mathtt{Bool}, \mathtt{T}_2 = \mathtt{T}_3\}\end{array}}{\Gamma \vdash \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{T}_2 \mid_{\mathcal{X}_1 \cup \mathcal{X}_2 \cup \mathcal{X}_3} C'} \quad \text{(CT-IF)}$$

Figure 22-1: Constraint typing rules

speaking, the algorithm is deterministic only if we consider it "modulo the choice of fresh names." We return to this point in Exercise 22.3.9.)

To lighten the notation in the following discussion, we sometimes elide the $\mathcal{X}$ and write just $\Gamma \vdash \mathtt{t} : \mathtt{T} \mid C$. □

22.3.3 EXERCISE [⋆ ↛]: Construct a constraint typing derivation whose conclusion is

$$\vdash \lambda\mathtt{x}:\mathtt{X}.\ \lambda\mathtt{y}:\mathtt{Y}.\ \lambda\mathtt{z}:\mathtt{Z}.\ (\mathtt{x}\ \mathtt{z})\ (\mathtt{y}\ \mathtt{z}) : \mathtt{S} \mid_{\mathcal{X}} C$$

for some $\mathtt{S}$, $\mathcal{X}$, and C. □

The idea of the constraint typing relation is that, given a term $\mathtt{t}$ and a context Γ, we can check whether $\mathtt{t}$ is typable under Γ by first collecting the constraints C that must be satisfied in order for $\mathtt{t}$ to have a type, together with a result type $\mathtt{S}$, sharing variables with C, that characterizes the possible types of $\mathtt{t}$ in terms of these variables. Then, to find solutions for $\mathtt{t}$, we just look for substitutions σ that satisfy C (i.e., that make all the equations in C into identities); for each such σ, the type $\sigma\mathtt{S}$ is a possible type of $\mathtt{t}$. If we find that there are *no* substitutions that satisfy C, then we know that $\mathtt{t}$ cannot be instantiated in such a way as to make it typable.

For example, the constraint set generated by the algorithm for the term $\mathtt{t} = \lambda \mathtt{x{:}X{\to}Y.\ x\ 0}$ is $\{\mathtt{Nat{\to}Z = X{\to}Y}\}$, and the associated result type is $\mathtt{(X{\to}Y){\to}Z}$. The substitution $\sigma = [\mathtt{X} \mapsto \mathtt{Nat},\ \mathtt{Z} \mapsto \mathtt{Bool},\ \mathtt{Y} \mapsto \mathtt{Bool}]$ makes the equation $\mathtt{Nat{\to}Z = X{\to}Y}$ into an identity, so we know that $\sigma(\mathtt{(X{\to}Y){\to}Z})$, i.e., $\mathtt{(Nat{\to}Bool){\to}Bool}$, is a possible type for $\mathtt{t}$.

This idea is captured formally by the following definition.

22.3.4 DEFINITION: Suppose that $\Gamma \vdash \mathtt{t : S} \mid C$. A *solution* for $(\Gamma, \mathtt{t}, \mathtt{S}, C)$ is a pair $(\sigma, \mathtt{T})$ such that σ satisfies C and $\sigma\mathtt{S} = \mathtt{T}$. □

The algorithmic problem of finding substitutions unifying a given constraint set C will be taken up in the next section. First, though, we should check that our constraint typing algorithm corresponds in a suitable sense to the original, declarative typing relation.

Given a context Γ and a term $\mathtt{t}$, we have two different ways of characterizing the possible ways of instantiating type variables in Γ and $\mathtt{t}$ to produce a valid typing:

1. [DECLARATIVE] as the set of all solutions for $(\Gamma, \mathtt{t})$ in the sense of Definition 22.2.1; or

2. [ALGORITHMIC] via the constraint typing relation, by finding $\mathtt{S}$ and C such that $\Gamma \vdash \mathtt{t : S} \mid C$ and then taking the set of solutions for $(\Gamma, \mathtt{t}, \mathtt{S}, C)$.

We show the equivalence of these two characterizations in two steps. First we show that every solution for $(\Gamma, \mathtt{t}, \mathtt{S}, C)$ is also a solution for $(\Gamma, \mathtt{t})$ (Theorem 22.3.5). Then we show that every solution for $(\Gamma, \mathtt{t})$ can be *extended* to a solution for $(\Gamma, \mathtt{t}, \mathtt{S}, C)$ (Theorem 22.3.7) by giving values for the type variables introduced by constraint generation.

22.3.5 THEOREM [SOUNDNESS OF CONSTRAINT TYPING]: Suppose that $\Gamma \vdash \mathtt{t : S} \mid C$. If $(\sigma, \mathtt{T})$ is a solution for $(\Gamma, \mathtt{t}, \mathtt{S}, C)$, then it is also a solution for $(\Gamma, \mathtt{t})$. □

For this direction of the argument, the fresh variable sets $\mathcal{X}$ are secondary and can be elided.

Proof: By induction on the given constraint typing derivation for $\Gamma \vdash \mathtt{t : S} \mid C$, reasoning by cases on the last rule used.

Case CT-VAR*:* $\mathtt{t = x} \quad \mathtt{x{:}S} \in \Gamma \quad C = \{\}$

We are given that $(\sigma, \mathtt{T})$ is a solution for $(\Gamma, \mathtt{t}, \mathtt{S}, C)$; since C is empty, this means just that $\sigma\mathtt{S} = \mathtt{T}$. But then by T-VAR we immediately obtain $\sigma\Gamma \vdash \mathtt{x : T}$, as required.

Case CT-ABS: $\mathtt{t} = \lambda \mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_2 \quad \mathtt{S} = \mathtt{T}_1 \to \mathtt{S}_2 \quad \Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{S}_2 \mid C$

We are given that $(\sigma, \mathtt{T})$ is a solution for $(\Gamma, \mathtt{t}, \mathtt{S}, C)$, that is, σ unifies C and $\mathtt{T} = \sigma\mathtt{S} = \sigma\mathtt{T}_1 \to \sigma\mathtt{S}_2$. So $(\sigma, \sigma\mathtt{S}_2)$ is a solution for $(\Gamma, \mathtt{t}_2, \mathtt{S}_2, C)$. By the induction hypothesis, $(\sigma, \sigma\mathtt{S}_2)$ is a solution for $((\Gamma, \mathtt{x}{:}\mathtt{T}_1), \mathtt{t}_2)$, i.e., $\sigma\Gamma, \mathtt{x}{:}\sigma\mathtt{T}_1 \vdash \sigma\mathtt{t}_2 : \sigma\mathtt{S}_2$. By T-ABS, $\sigma\Gamma \vdash \lambda\mathtt{x}{:}\sigma\mathtt{T}_1.\sigma\mathtt{t}_2 : \sigma\mathtt{T}_1 \to \sigma\mathtt{S}_2 = \sigma(\mathtt{T}_1 \to \mathtt{S}_2) = \mathtt{T}$, as required.

Case CT-APP: $\mathtt{t} = \mathtt{t}_1\ \mathtt{t}_2 \quad \mathtt{S} = \mathtt{X}$
$\Gamma \vdash \mathtt{t}_1 : \mathtt{S}_1 \mid C_1 \quad \Gamma \vdash \mathtt{t}_2 : \mathtt{S}_2 \mid C_2$
$C = C_1 \cup C_2 \cup \{\mathtt{S}_1 = \mathtt{S}_2 \to \mathtt{X}\}$

By definition, σ unifies C_1 and C_2 and $\sigma\mathtt{S}_1 = \sigma(\mathtt{S}_2 \to \mathtt{X})$. So $(\sigma, \sigma\mathtt{S}_1)$ and $(\sigma, \sigma\mathtt{S}_2)$ are solutions for $(\Gamma, \mathtt{t}_1, \mathtt{S}_1, C_1)$ and $(\Gamma, \mathtt{t}_2, \mathtt{S}_2, C_2)$, from which the induction hypothesis gives us $\sigma\Gamma \vdash \sigma\mathtt{t}_1 : \sigma\mathtt{S}_1$ and $\sigma\Gamma \vdash \sigma\mathtt{t}_2 : \sigma\mathtt{S}_2$. But since $\sigma\mathtt{S}_1 = \sigma\mathtt{S}_2 \to \sigma\mathtt{X}$, we have $\sigma\Gamma \vdash \sigma\mathtt{t}_1 : \sigma\mathtt{S}_2 \to \sigma\mathtt{X}$, and, by T-APP, $\sigma\Gamma \vdash \sigma(\mathtt{t}_1\ \mathtt{t}_2) : \sigma\mathtt{X} = \mathtt{T}$.

Other cases:

Similar. □

The argument for the *completeness* of constraint typing with respect to the ordinary typing relation is a bit more delicate, because we must deal carefully with fresh names.

22.3.6 DEFINITION: Write $\sigma\backslash\mathcal{X}$ for the substitution that is undefined for all the variables in $\mathcal{X}$ and otherwise behaves like σ. □

22.3.7 THEOREM [COMPLETENESS OF CONSTRAINT TYPING]: Suppose $\Gamma \vdash \mathtt{t} : \mathtt{S} \mid_{\mathcal{X}} C$. If $(\sigma, \mathtt{T})$ is a solution for $(\Gamma, \mathtt{t})$ and $dom(\sigma) \cap \mathcal{X} = \emptyset$, then there is some solution $(\sigma', \mathtt{T})$ for $(\Gamma, \mathtt{t}, \mathtt{S}, C)$ such that $\sigma'\backslash\mathcal{X} = \sigma$. □

Proof: By induction on the given constraint typing derivation.

Case CT-VAR: $\mathtt{t} = \mathtt{x} \quad \mathtt{x}{:}\mathtt{S} \in \Gamma$

From the assumption that $(\sigma, \mathtt{T})$ is a solution for $(\Gamma, \mathtt{x})$, the inversion lemma for the typing relation (9.3.1) tells us that $\mathtt{T} = \sigma\mathtt{S}$. But then $(\sigma, \mathtt{T})$ is also a $(\Gamma, \mathtt{x}, \mathtt{S}, \{\})$-solution.

Case CT-ABS: $\mathtt{t} = \lambda\mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_2 \quad \Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{S}_2 \mid_{\mathcal{X}} C \quad \mathtt{S} = \mathtt{T}_1 \to \mathtt{S}_2$

From the assumption that $(\sigma, \mathtt{T})$ is a solution for $(\Gamma, \lambda\mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_2)$, the inversion lemma for the typing relation yields $\sigma\Gamma, \mathtt{x}{:}\sigma\mathtt{T}_1 \vdash \sigma\mathtt{t}_2 : \mathtt{T}_2$ and $\mathtt{T} = \sigma\mathtt{T}_1 \to \mathtt{T}_2$ for some $\mathtt{T}_2$. By the induction hypothesis, there is a solution $(\sigma', \mathtt{T}_2)$ for $((\Gamma, \mathtt{x}{:}\mathtt{T}_1), \mathtt{t}_2, \mathtt{S}_2, C)$ such that $\sigma'\backslash\mathcal{X}$ agrees with σ. Now, $\mathcal{X}$ cannot include any of the type variables in $\mathtt{T}_1$. So $\sigma'\mathtt{T}_1 = \sigma\mathtt{T}_1$, and $\sigma'(\mathtt{S}) = \sigma'(\mathtt{T}_1 \to \mathtt{S}_2) = \sigma\mathtt{T}_1 \to \sigma'\mathtt{S}_2 = \sigma\mathtt{T}_1 \to \mathtt{T}_2 = \mathtt{T}$. Thus, we see that $(\sigma', \mathtt{T})$ is a solution for $(\Gamma, (\lambda\mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_2), \mathtt{T}_1 \to \mathtt{S}_2, C)$.

Case CT-APP: $t = t_1\ t_2 \quad \Gamma \vdash t_1 : S_1 \mid_{X_1} C_1 \quad \Gamma \vdash t_2 : S_2 \mid_{X_2} C_2$
$X_1 \cap X_2 = \emptyset$
$X_1 \cap FV(T_2) = \emptyset$
$X_2 \cap FV(T_1) = \emptyset$
X not mentioned in $X_1, X_2, S_1, S_2, C_1, C_2$
$S = X \quad X = X_1 \cup X_2 \cup \{X\} \quad C = C_1 \cup C_2 \cup \{S_1 = S_2 \to X\}$

From the assumption that (σ, T) is a solution for $(\Gamma,\ t_1\ t_2)$, the inversion lemma for the typing relation yields $\sigma\Gamma \vdash \sigma t_1 : T_1 \to T$ and $\sigma\Gamma \vdash \sigma t_2 : T_1$. By the induction hypothesis, there are solutions $(\sigma_1, T_1 \to T)$ for (Γ, t_1, S_1, C_1) and (σ_2, T_1) for (Γ, t_2, S_2, C_2), and $\sigma_1 \backslash X_1 = \sigma = \sigma_2 \backslash X_2$. We must exhibit a substitution σ' such that: (1) $\sigma' \backslash X$ agrees with σ; (2) $\sigma' X = T$; (3) σ' unifies C_1 and C_2; and (4) σ' unifies $\{S_1 = S_2 \to X\}$, i.e., $\sigma' S_1 = \sigma' S_2 \to \sigma' X$. Define σ' as follows:

$$\sigma' = \left[\begin{array}{ll} Y \mapsto U & \text{if } Y \notin X \text{ and } (Y \mapsto U) \in \sigma, \\ Y_1 \mapsto U_1 & \text{if } Y_1 \in X_1 \text{ and } (Y_1 \mapsto U_1) \in \sigma_1, \\ Y_2 \mapsto U_2 & \text{if } Y_2 \in X_2 \text{ and } (Y_2 \mapsto U_2) \in \sigma_2, \\ X \mapsto T & \end{array}\right]$$

Conditions (1) and (2) are obviously satisfied. (3) is satisfied because X_1 and X_2 do not overlap. To check (4), first note that the side-conditions about freshness guarantee that $FV(S_1) \cap (X_2 \cup \{X\}) = \emptyset$, so that $\sigma' S_1 = \sigma_1 S_1$. Now calculate as follows: $\sigma' S_1 = \sigma_1 S_1 = T_1 \to T = \sigma_2 S_2 \to T = \sigma' S_2 \to \sigma' X = \sigma'(S_2 \to X)$.

Other cases:

Similar. □

22.3.8 COROLLARY: Suppose $\Gamma \vdash t : S \mid C$. There is some solution for (Γ, t) iff there is some solution for (Γ, t, S, C). □

Proof: By Theorems 22.3.5 and 22.3.7. □

22.3.9 EXERCISE [RECOMMENDED, ⋆⋆⋆]: In a production compiler, the nondeterministic choice of a fresh type variable name in the rule CT-APP would typically be replaced by a call to a *function* that generates a new type variable—different from all others that it ever generates, and from all type variables mentioned explicitly in the context or term being checked—each time it is called. Because such global "`gensym`" operations work by side effects on a hidden global variable, they are difficult to reason about formally. However, we can mimic their behavior in a fairly accurate and mathematically more tractable way by "threading" a sequence of unused variable names through the constraint generation rules.

Let F denote a sequence of distinct type variable names. Then, instead of writing $\Gamma \vdash \texttt{t} : \texttt{T} \;\; |_X \;\; C$ for the constraint generation relation, we write $\Gamma \vdash_F \texttt{t} : \texttt{T} \;\; |_{F'} \;\; C$, where Γ, F, and $\texttt{t}$ are inputs to the algorithm and $\texttt{T}$, F', and C are outputs. Whenever it needs a fresh type variable, the algorithm takes the front element of F and returns the rest of F as F'.

Write out the rules for this algorithm. Prove that they are equivalent, in an appropriate sense, to the original constraint generation rules. □

22.3.10 EXERCISE [RECOMMENDED, ⋆⋆]: Implement the algorithm from Exercise 22.3.9 in ML. Use the datatype

```
type ty =
    TyBool
  | TyArr of ty * ty
  | TyId of string
  | TyNat
```

for types, and

```
type constr = (ty * ty) list
```

for constraint sets. You will also need a representation for infinite sequences of fresh variable names. There are lots of ways of doing this; here is a fairly direct one using a recursive datatype:

```
type nextuvar = NextUVar of string * uvargenerator
and uvargenerator = unit → nextuvar

let uvargen =
  let rec f n () = NextUVar("?X_" ^ string_of_int n, f (n+1))
  in f 0
```

That is, `uvargen` is a function that, when called with argument `()`, returns a value of the form `NextUVar(x,f)`, where `x` is a fresh type variable name and `f` is another function of the same form. □

22.3.11 EXERCISE [⋆⋆]: Show how to extend the constraint generation algorithm to deal with general recursive function definitions (§11.11). □

22.4 Unification

To calculate solutions to constraint sets, we use the idea, due to Hindley (1969) and Milner (1978), of using *unification* (Robinson, 1971) to check that the set of solutions is nonempty and, if so, to find a "best" element, in the sense that all solutions can be generated straightforwardly from this one.

```
unify(C)  =  if C = ∅, then [ ]
             else let {S = T} ∪ C′ = C in
               if S = T
                 then unify(C′)
               else if S = X and X ∉ FV(T)
                 then unify([X ↦ T]C′) ∘ [X ↦ T]
               else if T = X and X ∉ FV(S)
                 then unify([X ↦ S]C′) ∘ [X ↦ S]
               else if S = S1→S2 and T = T1→T2
                 then unify(C′ ∪ {S1 = T1, S2 = T2})
               else
                 fail
```

Figure 22-2: Unification algorithm

22.4.1 DEFINITION: A substitution σ is *less specific* (or *more general*) *than* a substitution σ', written $\sigma \sqsubseteq \sigma'$, if $\sigma' = \gamma \circ \sigma$ for some substitution γ. □

22.4.2 DEFINITION: A *principal unifier* (or sometimes *most general unifier*) for a constraint set C is a substitution σ that satisfies C and such that $\sigma \sqsubseteq \sigma'$ for every substitution σ' satisfying C. □

22.4.3 EXERCISE [⋆]: Write down principal unifiers (when they exist) for the following sets of constraints:

{X = Nat, Y = X→X}	{Nat→Nat = X→Y}
{X→Y = Y→Z, Z = U→W}	{Nat = Nat→Y}
{Y = Nat→Y}	{} (the empty set of constraints)

□

22.4.4 DEFINITION: The *unification algorithm* for types is defined in Figure 22-2.[2] The phrase "let $\{S = T\} \cup C' = C$" in the second line should be read as "choose a constraint S=T from the set C and let C' denote the remaining constraints from C." □

The side conditions $X \notin FV(T)$ in the fifth line and $X \notin FV(S)$ in the seventh are known as the *occur check*. Their effect is to prevent the algorithm from generating a solution involving a cyclic substitution like $X \mapsto X \to X$, which

2. Note that nothing in this algorithm depends on the fact that we are unifying type expressions as opposed to some other sort of expressions; the same algorithm can be used to solve equality constraints between any kind of (first-order) expressions.

makes no sense if we are talking about finite type expressions. (If we expand our language to include *infinite* type expressions—i.e. recursive types in the sense of Chapters 20 and 21—then the occur check can be omitted.)

22.4.5 THEOREM: The algorithm *unify* always terminates, failing when given a non-unifiable constraint set as input and otherwise returning a principal unifier. More formally:

1. $unify(C)$ halts, either by failing or by returning a substitution, for all C;
2. if $unify(C) = \sigma$, then σ is a unifier for C;
3. if δ is a unifier for C, then $unify(C) = \sigma$ with $\sigma \sqsubseteq \delta$. □

Proof: For part (1), define the *degree* of a constraint set C to be the pair (m, n), where m is the number of distinct type variables in C and n is the total size of the types in C. It is easy to check that each clause of the *unify* algorithm either terminates immediately (with success in the first case or failure in the last) or else makes a recursive call to *unify* with a constraint set of lexicographically smaller degree.

Part (2) is a straightforward induction on the number of recursive calls in the computation of $unify(C)$. All the cases are trivial except for the two involving variables, which depend on the observation that, if σ unifies $[\mathsf{X} \mapsto \mathsf{T}]D$, then $\sigma \circ [\mathsf{X} \mapsto \mathsf{T}]$ unifies $\{\mathsf{X} = \mathsf{T}\} \cup D$ for any constraint set D.

Part (3) again proceeds by induction on the number of recursive calls in the computation of $unify(C)$. If C is empty, then $unify(C)$ immediately returns the trivial substitution $[\,]$; since $\delta = \delta \circ [\,]$, we have $[\,] \sqsubseteq \delta$ as required. If C is non-empty, then $unify(C)$ chooses some pair (S, T) from C and continues by cases on the shapes of S and T.

Case: $\mathsf{S} = \mathsf{T}$

Since δ is a unifier for C, it also unifies C'. By the induction hypothesis, $unify(C) = \sigma$ with $\sigma \sqsubseteq \delta$, as required.

Case: $\mathsf{S} = \mathsf{X}$ and $\mathsf{X} \notin FV(\mathsf{T})$

Since δ unifies S and T, we have $\delta(\mathsf{X}) = \delta(\mathsf{T})$. So, for any type U, we have $\delta(\mathsf{U}) = \delta([\mathsf{X} \mapsto \mathsf{T}]\mathsf{U})$; in particular, since δ unifies C', it must also unify $[\mathsf{X} \mapsto \mathsf{T}]C'$. The induction hypothesis then tells us that $unify([\mathsf{X} \mapsto \mathsf{T}]C') = \sigma'$, with $\delta = \gamma \circ \sigma'$ for some γ. Since $unify(C) = \sigma' \circ [\mathsf{X} \mapsto \mathsf{T}]$, showing that $\delta = \gamma \circ (\sigma' \circ [\mathsf{X} \mapsto \mathsf{T}])$ will complete the argument. So consider any type variable Y. If $\mathsf{Y} \neq \mathsf{X}$, then clearly $(\gamma \circ (\sigma' \circ [\mathsf{X} \mapsto \mathsf{T}]))\mathsf{Y} = (\gamma \circ \sigma')\mathsf{Y} = \delta\mathsf{Y}$. On the other hand, $(\gamma \circ (\sigma' \circ [\mathsf{X} \mapsto \mathsf{T}]))\mathsf{X} = (\gamma \circ \sigma')\mathsf{T} = \delta\mathsf{X}$, as we saw above. Combining these observations, we see that $\delta\mathsf{Y} = (\gamma \circ (\sigma' \circ [\mathsf{X} \mapsto \mathsf{T}]))\mathsf{Y}$ for all variables Y, that is, $\delta = (\gamma \circ (\sigma' \circ [\mathsf{X} \mapsto \mathsf{T}]))$.

Case: $\mathsf{T} = \mathsf{X}$ and $\mathsf{X} \notin FV(\mathsf{S})$

Similar.

Case: $\mathsf{S} = \mathsf{S}_1 \to \mathsf{S}_2$ and $\mathsf{T} = \mathsf{T}_1 \to \mathsf{T}_2$

Straightforward. Just note that δ is a unifier of $\{\mathsf{S}_1 \to \mathsf{S}_2 = \mathsf{T}_1 \to \mathsf{T}_2\} \cup C'$ iff it is a unifier of $C' \cup \{\mathsf{S}_1 = \mathsf{T}_1, \mathsf{S}_2 = \mathsf{T}_2\}$.

If none of the above cases apply to S and T, then $\mathit{unify}(C)$ fails. But this can happen in only two ways: either S is $\mathtt{Nat}$ and T is an arrow type (or vice versa), or else $\mathsf{S} = \mathsf{X}$ and $\mathsf{X} \in \mathsf{T}$ (or vice versa). The first case obviously contradicts the assumption that C is unifiable. To see that the second does too, recall that, by assumption, $\delta\mathsf{S} = \delta\mathsf{T}$; if X occurred in T, then $\delta\mathsf{T}$ would always be strictly larger than $\delta\mathsf{S}$. Thus, if $\mathit{unify}(C)$ fails, then C is not unifiable, contradicting our assumption that δ is a unifier for C; so this case cannot occur. □

22.4.6 EXERCISE [RECOMMENDED, ⋆⋆⋆]: Implement the unification algorithm. □

22.5 Principal Types

We remarked above that if there is *some* way to instantiate the type variables in a term so that it becomes typable, then there is a *most general* or *principal* way of doing so. We now formalize this observation.

22.5.1 DEFINITION: A *principal solution* for $(\Gamma, \mathtt{t}, \mathsf{S}, C)$ is a solution (σ, T) such that, whenever (σ', T') is also a solution for $(\Gamma, \mathtt{t}, \mathsf{S}, C)$, we have $\sigma \sqsubseteq \sigma'$. When (σ, T) is a principal solution, we call T a *principal type* of $\mathtt{t}$ under Γ.[3] □

22.5.2 EXERCISE [⋆ ↛]: Find a principal type for $\lambda\mathtt{x{:}X.}\ \lambda\mathtt{y{:}Y.}\ \lambda\mathtt{z{:}Z.}\ (\mathtt{x}\ \mathtt{z})\ (\mathtt{y}\ \mathtt{z})$. □

22.5.3 THEOREM [PRINCIPAL TYPES]: If $(\Gamma, \mathtt{t}, \mathsf{S}, C)$ has any solution, then it has a principal one. The unification algorithm in Figure 22-2 can be used to determine whether $(\Gamma, \mathtt{t}, \mathsf{S}, C)$ has a solution and, if so, to calculate a principal one. □

Proof: By the definition of a solution for $(\Gamma, \mathtt{t}, \mathsf{S}, C)$ and the properties of unification. □

22.5.4 COROLLARY: It is decidable whether $(\Gamma, \mathtt{t})$ has a solution. □

Proof: By Corollary 22.3.8 and Theorem 22.5.3. □

3. Principal types should not be confused with *principal typings*. See page 337.

22.5.5 EXERCISE [RECOMMENDED, ⋆⋆⋆ ↛]: Combine the constraint generation and unification algorithms from Exercises 22.3.10 and 22.4.6 to build a typechecker that calculates principal types, taking the `reconbase` checker as a starting point. A typical interaction with your typechecker might look like:

```
λx:X. x;
▸ <fun> : X → X

λz:ZZ. λy:YY. z (y true);
▸ <fun> : (?X0→?X1) → (Bool→?X0) → ?X1

λw:W. if true then false else w false;
▸ <fun> : (Bool→Bool) → Bool
```

Type variables with names like $?X_0$ are automatically generated. □

22.5.6 EXERCISE [⋆⋆⋆]: What difficulties arise in extending the definitions above (22.3.2, etc.) to deal with records? How might they be addressed? □

The idea of principal types can be used to build a type reconstruction algorithm that works more incrementally than the one we have developed here. Instead of generating all the constraints first and then trying to solve them, we can interleave generation and solving, so that the type reconstruction algorithm actually returns a principal type at each step. The fact that the types are always principal ensures that the algorithm never needs to re-analyze a subterm: it makes only the minimum commitments needed to achieve typability at each step. One major advantage of such an algorithm is that it can pinpoint errors in the user's program much more precisely.

22.5.7 EXERCISE [⋆⋆⋆ ↛]: Modify your solution to Exercise 22.5.5 so that it performs unification incrementally and returns principal types. □

22.6 Implicit Type Annotations

Languages supporting type reconstruction typically allow programmers to completely omit type annotations on lambda-abstractions. One way to achieve this (as we remarked in §22.2) is simply to make the parser fill in omitted annotations with freshly generated type variables. A better alternative is to add un-annotated abstractions to the syntax of terms and a corresponding rule to the constraint typing relation.

$$\frac{X \notin \mathcal{X} \qquad \Gamma, x{:}X \vdash t_1 : T \mid_{\mathcal{X}} C}{\Gamma \vdash \lambda x.t_1 : X \to T \mid_{\mathcal{X} \cup \{X\}} C} \quad \text{(CT-AbsInf)}$$

This account of un-annotated abstractions is a bit more direct than regarding them as syntactic sugar. It is also more expressive, in a small but useful way: if we make several *copies* of an un-annotated abstraction, the CT-AbsInf rule will allow us to choose a *different* variable as the argument type of each copy. By contrast, if we regard a bare abstraction as being annotated with an invisible type variable, then making copies will yield several expressions sharing the *same* argument type. This difference is important for the discussion of let-polymorphism in the following section.

22.7 Let-Polymorphism

The term *polymorphism* refers to a range of language mechanisms that allow a single part of a program to be used with different types in different contexts (§23.2 discusses several varieties of polymorphism in more detail). The type reconstruction algorithm shown above can be generalized to provide a simple form of polymorphism known as *let-polymorphism* (also *ML-style* or *Damas-Milner* polymorphism). This feature was introduced in the original dialect of ML (Milner, 1978) and has been incorporated in a number of successful language designs, where it forms the basis of powerful *generic libraries* of commonly used structures (lists, arrays, trees, hash tables, streams, user-interface widgets, etc.).

The motivation for let-polymorphism arises from examples like the following. Suppose we define and use a simple function `double`, which applies its first argument twice in succession to its second:

```
let double = λf:Nat→Nat. λa:Nat. f(f(a)) in
double (λx:Nat. succ (succ x)) 2;
```

Because we want to apply `double` to a function of type `Nat→Nat`, we choose type annotations that give it type `(Nat→Nat)→(Nat→Nat)`. We can alternatively define `double` so that it can be used to double a boolean function:

```
let double = λf:Bool→Bool. λa:Bool. f(f(a)) in
double (λx:Bool. x) false;
```

What we *cannot* do is use the same `double` function with both booleans and numbers: if we need both in the same program, we must define two versions that are identical except for type annotations.

```
let doubleNat = λf:Nat→Nat. λa:Nat. f(f(a)) in
let doubleBool = λf:Bool→Bool. λa:Bool. f(f(a)) in
```

```
let a = doubleNat (λx:Nat. succ (succ x)) 1 in
let b = doubleBool (λx:Bool. x) false in ...
```

Even annotating the abstractions in `double` with a *type variable*

```
let double = λf:X→X. λa:X. f(f(a)) in ...
```

does not help. For example, if we write

```
let double = λf:X→X. λa:X. f(f(a)) in
let a = double (λx:Nat. succ (succ x)) 1 in
let b = double (λx:Bool. x) false in ...
```

then the use of `double` in the definition of `a` generates the constraint `X→X` = `Nat→Nat`, while the use of `double` in the definition of `b` generates the constraint `X→X` = `Bool→Bool`. These constraints place unsatisfiable demands on `X`, making the whole program untypable.

What went wrong here? The variable `X` plays two distinct roles in the example. First, it captures the constraint that the first argument to `double` in the calculation of `a` must be a function whose domain and range types are the same as the type (`Nat`) of the other argument to `double`. Second, it captures the constraint that the arguments to `double` in the calculation of `b` must be similarly related. Unfortunately, because the same variable `X` is used in both cases, we also end up with the spurious constraint that the second arguments to the two uses of `double` must have the same type.

What we'd like is to break this last connection—i.e., to associate a *different* variable `X` with each use of `double`. Fortunately, this is easily accomplished. The first step is to change the ordinary typing rule for `let` so that, instead of calculating a type for the right-hand side $\mathtt{t}_1$ and then using this as the type of the bound variable `x` while calculating a type for the body $\mathtt{t}_2$,

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \qquad \Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{let}\ \mathtt{x}{=}\mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}_2} \tag{T-Let}$$

it instead *substitutes* $\mathtt{t}_1$ for `x` in the body, and then typechecks this expanded expression:

$$\frac{\Gamma \vdash [\mathtt{x} \mapsto \mathtt{t}_1]\mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{let}\ \mathtt{x}{=}\mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}_2} \tag{T-LetPoly}$$

We write a constraint-typing rule for `let` in a similar way:

$$\frac{\Gamma \vdash [\mathtt{x} \mapsto \mathtt{t}_1]\mathtt{t}_2 : \mathtt{T}_2 \ \mid_X C}{\Gamma \vdash \mathtt{let}\ \mathtt{x}{=}\mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}_2 \ \mid_X C} \tag{CT-LetPoly}$$

In essence, what we've done is to change the typing rules for `let` so that they perform a step of evaluation

$$\mathtt{let}\ \mathtt{x}{=}\mathtt{v}_1\ \mathtt{in}\ \mathtt{t}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_1]\mathtt{t}_2 \tag{E-LetV}$$

before calculating types.

The second step is to rewrite the definition of `double` using the *implicitly annotated* lambda-abstractions from §22.6.

```
let double = λf. λa. f(f(a)) in
let a = double (λx:Nat. succ (succ x)) 1 in
let b = double (λx:Bool. x) false in ...
```

The combination of the constraint typing rules for `let` (CT-LETPOLY) and the implicitly annotated lambda-abstraction (CT-ABSINF) gives us exactly what we need: CT-LETPOLY makes two copies of the definition of `double`, and CT-ABSINF assigns each of the abstractions a different type variable. The ordinary process of constraint solving does the rest.

However, this scheme has some flaws that need to be addressed before we can use it in practice. One obvious one is that, if we don't happen to actually use the `let`-bound variable in the body of the `let`, then the definition will never actually be typechecked. For example, a program like

```
let x = <utter garbage> in 5
```

will pass the typechecker. This can be repaired by adding a premise to the typing rule

$$\frac{\Gamma \vdash [\mathtt{x} \mapsto \mathtt{t}_1]\mathtt{t}_2 : \mathtt{T}_2 \qquad \Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1}{\Gamma \vdash \mathtt{let}\ \mathtt{x}{=}\mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathtt{T}_2} \qquad \text{(T-LETPOLY)}$$

and a corresponding premise to CT-LETPOLY, ensuring that $\mathtt{t}_1$ is well typed.

A related problem is that, if the body of the `let` contains *many* occurrences of the `let`-bound variable, then the whole right-hand side of the `let`-definition will be checked once per occurrence, whether or not it contains any implicitly annotated lambda-abstractions. Since the right-hand side itself can contain `let`-bindings, this typing rule can cause the typechecker to perform an amount of work that is exponential in the size of the original term!

To avoid this re-typechecking, practical implementations of languages with let-polymorphism actually use a more clever (though formally equivalent) re-formulation of the typing rules. In outline, the typechecking of a term $\mathtt{let}\ \mathtt{x}{=}\mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2$ in a context Γ proceeds as follows:

1. We use the constraint typing rules to calculate a type $\mathtt{S}_1$ and a set C_1 of associated constraints for the right-hand side $\mathtt{t}_1$.

2. We use unification to find a most general solution σ to the constraints C_1 and apply σ to $\mathtt{S}_1$ (and Γ) to obtain $\mathtt{t}_1$'s *principal type* $\mathtt{T}_1$.

3. We *generalize* any variables remaining in $\mathtt{T}_1$. If $\mathtt{X}_1 \ldots \mathtt{X}_n$ are the remaining variables, we write $\forall \mathtt{X}_1 \ldots \mathtt{X}_n . \mathtt{T}_1$ for the principal *type scheme* of $\mathtt{t}_1$.

One caveat is here that we need to be careful *not* to generalize variables T_1 that are also mentioned in Γ, since these correspond to real constraints between $\mathtt{t}_1$ and its environment. For example, in

```
λf:X→X. λx:X. let g=f in g(x);
```

we should not generalize the variable `X` in the type `X→X` of `g`, since doing so would allow us to type wrong programs like this one:

```
(λf:X→X. λx:X. let g=f in g(0))
  (λx:Bool. if x then true else false)
  true;
```

4. We extend the context to record the type scheme $\forall \mathsf{X}_1 \ldots \mathsf{X}_n . \mathsf{T}_1$ for the bound variable `x`, and start typechecking the body $\mathtt{t}_2$. In general, the context now associates each free variable with a type scheme, not just a type.

5. Each time we encounter an occurrence of `x` in $\mathtt{t}_2$, we look up its type scheme $\forall \mathsf{X}_1 \ldots \mathsf{X}_n . \mathsf{T}_1$. We now generate fresh type variables $\mathsf{Y}_1 \ldots \mathsf{Y}_n$ and use them to *instantiate* the type scheme, yielding $[\mathsf{X}_1 \mapsto \mathsf{Y}_1, \ldots, \mathsf{X}_n \mapsto \mathsf{Y}_n]\mathsf{T}_1$, which we use as the type of `x`.[4]

This algorithm is much more efficient than the simplistic approach of substituting away `let` expressions before typechecking. Indeed, decades of experience have shown that in practice it appears "essentially linear" in the size of the input program. It therefore came as a significant surprise when Kfoury, Tiuryn, and Urzyczyn (1990) and independently Mairson (1990) showed that its worst-case complexity is still exponential! The example they constructed involves using deeply nested sequences of `let`s in the right-hand sides of other `let`s—rather than in their bodies, where nesting of `let`s is common—to build expressions whose types grow exponentially larger than the expressions themselves. For example, the following OCaml program, due to Mairson (1990), is well typed but takes a very long time to typecheck.

```
let f0 = fun x → (x,x) in
  let f1 = fun y → f0(f0 y) in
    let f2 = fun y → f1(f1 y) in
      let f3 = fun y → f2(f2 y) in
        let f4 = fun y → f3(f3 y) in
          let f5 = fun y → f4(f4 y) in
            f5 (fun z → z)
```

4. The difference between a lambda-abstraction that is explicitly annotated with a type variable and an un-annotated abstraction for which the constraint generation algorithm creates a variable becomes moot once we introduce generalization and instantiation. Either way, the right-hand side of a `let` is assigned a type involving a variable, which is generalized before being added to the context and replaced by a fresh variable every time it is instantiated.

To see why, try entering $\texttt{f}_0$, $\texttt{f}_1$, etc., one at a time, into the OCaml top-level. See Kfoury, Tiuryn, and Urzyczyn (1994) for further discussion.

A final point worth mentioning is that, in designing full-blown programming languages with let-polymorphism, we need to be a bit careful of the interaction of polymorphism and side-effecting features such as mutable storage cells. A simple example illustrates the danger:

```
let r = ref (λx. x) in
(r:=(λx:Nat. succ x); (!r)true);
```

Using the algorithm sketched above, we calculate `Ref(X→X)` as the principal type of the right-hand side of the `let`; since `X` appears nowhere else, this type can be generalized to `∀X.Ref(X→X)`, and we assign this type scheme to `r` when we add it to the context. When typechecking the assignment in the second line, we instantiate this type to `Ref(Nat→Nat)`. When typechecking the third line, we instantiate it to `Ref(Bool→Bool)`. But this is unsound, since when the term is evaluated it will end up applying `succ` to `true`.

The problem here is that the typing rules have gotten out of sync with the evaluation rules. The typing rules introduced in this section tell us that, when we see a `let` expression, we should *immediately* substitute the right-hand side into the body. But the evaluation rules tell us that we may perform this substitution only *after* the right-hand side has been reduced to a value. The typing rules see two uses of the `ref` constructor, and analyze them under different assumptions, but at run time only one `ref` is actually allocated.

We can correct this mismatch in two ways—by adjusting evaluation or typing. In the former case, the evaluation rule for `let` would become[5]

$$\texttt{let x=}t_1\ \texttt{in}\ t_2 \longrightarrow [\texttt{x} \mapsto t_1]t_2 \qquad \text{(E-Let)}$$

Under this strategy, the first step in evaluating our dangerous example from above would replace `r` by its definition, yielding

```
(ref (λx. x)) := (λx:Nat. succ x) in
(!(ref (λx. x))) true;
```

which is perfectly safe! The first line creates a reference cell initially containing the identity function, and stores `(λx:Nat. succ x)` into it. The second creates *another* reference containing the identity, extracts its contents, and applies it to `true`. However, this calculation also demonstrates that changing the evaluation rule to fit the typing rule gives us a language with a rather

5. Strictly speaking, we should annotate this rule with a *store,* as we did in Chapter 13, since we are talking about a language with references:

$$\texttt{let x=}t_1\ \texttt{in}\ t_2 \mid \mu \longrightarrow [\texttt{x} \mapsto t_1]t_2 \mid \mu \qquad \text{(E-Let)}$$

strange semantics that no longer matches standard intuitions about call-by-value evaluation order. (Imperative languages with non-CBV evaluation strategies are not unheard-of [Augustsson, 1984], but they have never become popular because of the difficulty of understanding and controlling the ordering of side effects at run time.)

It is better to change the typing rule to match the evaluation rule. Fortunately, this is easy: we just impose the restriction (often called the *value restriction*) that a `let`-binding can be treated polymorphically—i.e., its free type variables can be generalized—only if its right-hand side is a syntactic value. This means that, in the dangerous example, the type assigned to `r` when we add it to the context will be `X→X`, not `∀X.X→X`. The constraints imposed by the second line will force `X` to be `Nat`, and this will cause the typechecking of the third line to fail, since `Nat` cannot be unified with `Bool`.

The value restriction solves our problem with type safety, at some cost in expressiveness: we can no longer write programs in which the right-hand sides of `let` expressions can both perform some interesting computation and be assigned a polymorphic type scheme. What is surprising is that this restriction makes hardly any difference in practice. Wright (1995) settled this point by analyzing a huge corpus of code written in an ML dialect—the 1990 definition of Standard ML (Milner, Tofte, and Harper, 1990)—that provided a more flexible `let`-typing rule based on *weak type variables* and observing that all but a tiny handful of right-hand sides were syntactic values anyway. This observation more or less closed the argument, and all major languages with ML-style let-polymorphism now adopt the value restriction.

22.7.1 EXERCISE [⋆⋆⋆ ↛]: Implement the algorithm sketched in this section. □

22.8 Notes

Notions of principal types for the lambda-calculus go back at least to the work of Curry in the 1950s (Curry and Feys, 1958). An algorithm for calculating principal types based on Curry's ideas was given by Hindley (1969); similar algorithms were discovered independently by Morris (1968) and Milner (1978). In the world of propositional logic, the ideas go back still further, perhaps to Tarski in the 1920s and certainly to the Meredith cousins in the 1950s (Lemmon, Meredith, Meredith, Prior, and Thomas, 1957); their first implementation on a computer was by David Meredith in 1957. Additional historical remarks on principal types can be found in Hindley (1997).

Unification (Robinson, 1971) is fundamental to many areas of computer science. Thorough introductions can be found, for example, in Baader and Nipkow (1998), Baader and Siekmann (1994), and Lassez and Plotkin (1991).

ML-style let-polymorphism was first described by Milner (1978). A number of type reconstruction algorithms have been proposed, notably the classic *Algorithm W (Damas and Milner)* of Damas and Milner (1982; also see Lee and Yi, 1998). The main difference between Algorithm W and the presentation in this chapter is that the former is specialized for "pure type reconstruction"—assigning principal types to completely *untyped* lambda-terms—while we have mixed type checking and type reconstruction, permitting terms to include explicit type annotations that may, but need not, contain variables. This makes our technical presentation a bit more involved (especially the proof of completeness, Theorem 22.3.7, where we must be careful to keep the programmer's type variables separate from the ones introduced by the constraint generation rules), but it meshes better with the style of the other chapters.

A classic paper by Cardelli (1987) lays out a number of implementation issues. Other expositions of type reconstruction algorithms can be found in Appel (1998), Aho et al. (1986), and Reade (1989). A particularly elegant presentation of the core system called *mini-ML* (Clement, Despeyroux, Despeyroux, and Kahn, 1986) often forms the basis for theoretical discussions. Tiuryn (1990) surveys a range of type reconstruction problems.

Principal types should not be confused with the similar notion of *principal typings*. The difference is that, when we calculate principal types, the context Γ and term `t` are considered as inputs to the algorithm, while the principal type `T` is the output. An algorithm for calculating principal typings takes just `t` as input and yields both Γ and `T` as outputs—i.e., it calculates the *minimal assumptions* about the types of the free variables in `t`. Principal typings are useful in supporting separate compilation and "smartest recompilation," performing incremental type inference, and pinpointing type errors. Unfortunately, many languages, in particular ML, have principal types but not principal typings. See Jim (1996).

ML-style polymorphism, with its striking combination of power and simplicity, hits a "sweet spot" in the language design space; mixing it with other sophisticated typing features has often proved quite delicate. The biggest success story in this arena is the elegant account of type reconstruction for record types proposed by Wand (1987) and further developed by Wand (1988, 1989b), Remy (1989, 1990; 1992a, 1992b, 1998), and many others. The idea is to introduce a new kind of variable, called a *row variable*, that ranges not over types but over entire "rows" of field labels and associated types. A simple form of *equational unification* is used solve constraint sets involving row variables. See Exercise 22.5.6. Garrigue (1994) and others have developed related methods for variant types. These techniques have been extended to general notions of *type classes* (Kaes, 1988; Wadler and Blott, 1989), *constraint types*

(Odersky, Sulzmann, and Wehr, 1999), and *qualified types* (Jones, 1994b,a), which form the basis of Haskell's system of *type classes* (Hall et al., 1996; Hudak et al., 1992; Thompson, 1999); similar ideas appear in Mercury (Somogyi, Henderson, and Conway, 1996) and Clean (Plasmeijer, 1998).

Type reconstruction for the more powerful form of *impredicative polymorphism* discussed in Chapter 23 was shown to be undecidable by Wells (1994). Indeed, several forms of *partial type reconstruction* for this system also turn out to be undecidable. §23.6 and §23.8 give more information on these results and on methods for combining ML-style type reconstruction with stronger forms of polymorphism such as *rank-2 polymorphism*.

For the combination of subtyping with ML-style type reconstruction, some promising initial results have been reported (Aiken and Wimmers, 1993; Eifrig, Smith, and Trifonov, 1995; Jagannathan and Wright, 1995; Trifonov and Smith, 1996; Odersky, Sulzmann, and Wehr, 1999; Flanagan and Felleisen, 1997; Pottier, 1997), but practical checkers have yet to see widespread use.

Extending ML-style type reconstruction to handle recursive types (Chapter 20) has been shown *not* to pose significant difficulties (Huet, 1975, 1976). The only significant difference from the algorithms presented in this chapter appears in the definition of unification, where we omit the *occur check* (which ordinarily ensures that the substitution returned by the unification algorithm is acyclic). Having done this, to ensure termination we also need to modify the representation used by the unification algorithm so that it maintains sharing, e.g., using by destructive operations on (potentially cyclic) pointer structures. Such representations are common in high-performance implementations.

The mixture of type reconstruction with recursively defined *terms,* on the other hand, raises one tricky problem, known as *polymorphic recursion*. A simple (and unproblematic) typing rule for recursive function definitions in ML specifies that a recursive function can be used within the body of its definition only monomorphically (i.e., all recursive calls must have identically typed arguments and results), while occurrences in the rest of the program may be used polymorphically (with arguments and results of different types). Mycroft (1984) and Meertens (1983) proposed a polymorphic typing rule for recursive definitions that allows recursive calls to a recursive function from its own body to be instantiated with different types. This extension, often called the *Milner-Mycroft Calculus,* was shown to have an undecidable reconstruction problem by Henglein (1993) and independently by Kfoury, Tiuryn, and Urzyczyn (1993a); both of these proofs depend on the undecidability of the (unrestricted) semi-unification problem, shown by Kfoury, Tiuryn, and Urzyczyn (1993b).