Theorem 6.6 (Preservation)**.** *Let $e, e' \in$ Expr such that $e \rightarrow e'$. If e is closed and well-typed, then so is e'.*

Proof (Exercise). Since e is closed and well-typed, we must have $\emptyset \vdash e : \tau$ for some type τ. We prove a slightly stronger property than required, namely that if $e \rightarrow e'$, then e' is closed and $\emptyset \vdash e' : \tau$ (i.e., the exact type τ of e is preserved under evaluation). The proof goes by induction on the derivation of $\emptyset \vdash e : \tau$ using case splitting on the last rule of the derivation. At each step of the induction, we assume that the desired property holds for all subderivations (i.e., whenever $\emptyset \vdash e_1 : \tau_1$ is proved by a subderivation and $e_1 \rightarrow e_1'$, then e_1' is closed and $\emptyset \vdash e_1' : \tau_1$). We leave the details of the proof as an exercise. **Hint:** In each case, the final typing rule in the derivation of $\emptyset \vdash e : \tau$ determines the top-level syntactic structure of e. In turn, this restricts the possible final rules that may have been used in the derivation of $e \rightarrow e'$. By further case splitting on these relevant final rules of the small-step SOS, you can apply the induction hypothesis where needed. The interesting cases are the rules TypeConst and the typing rules for calls in combination with the relevant do rules of the small-step SOS for these cases. In all these interesting cases, you need to apply the substitution lemma (Lemma 6.5) to complete the proof. □

6.3 Parametric Polymorphism (optional)

In the design of our simple type system for JakartaScript we have made several decisions that simplify the type checking and type inference problem. Unfortunately, these decisions also affect the usability and expressiveness of our type system. In particular, we required that the programmer annotates each function with the types of its parameters, and each recursive function with the type of its return value. Annotating programs with types is often tedious. As programmers we prefer to be released from the burden of having to write such annotations explicitly. Moreover, we have observed that the type system rejects certain programs that can be safely executed according to our operational semantics, e.g., programs which make use of polymorphic functions.

In this section, we will study an interesting point in the design space of static type systems that is referred to as the *Hindley-Milner type system.* This type system solves both of the limitations of our simple type system: (1) it enables type inference without requiring any programmer-provided type annotations and (2) it can deal with programs that make use of polymorphic functions. The Hindley-Milner type system is implemented in a number of programming languages, specifically the ML family of languages, which includes SML and OCaml, as well as related languages such as Haskell.

6.3.1 Type Inference without Type Annotations

Before we introduce the Hindley-Milner type system formally, we explain it through a series of examples.

As a first example, consider the following implementation of the factorial function in JakartaScript:

$$\texttt{function}\ \mathit{fac}(x)\ (x === 0\ ?\ 1 : x * \mathit{fac}(x - 1))$$

For this function to be well-typed in our simple type system, we would have to explicitly annotate the parameter x of function *fac* as well as the function's return type. However, if we take a closer look at the function's body:

$$x === 0\ ?\ 1 : x * \mathit{fac}(x - 1)$$

we observe that there is really only one possible type annotation that can work. Specifically, if we look at the test $x === 0$ in the conditional, then we can infer that x must be of type **Num** since it is compared with the value 0. Similarly, from the fact that the result of the recursive call $\mathit{fac}(x - 1)$ is used in a multiplication operation, we can infer that the return value of *fac* must also be of type **Num**. The Hindley-Milner type system exploits this idea to infer all types without explicit type annotations.

Types with Type Variables

In certain cases, the operations from which an expression is built may not be specific enough to infer a monomorphic type for the expression (i.e., a type that is either one of the base types **Num** and **Bool**, or a function type built from base types). For example, consider the following curried function

$$\mathit{apply} = x \texttt{=>} f \texttt{=>} f(x)$$

From the subexpression $f(x)$ of *apply* we can infer that the type of f must be a function type $\tau_1 \Rightarrow \tau_2$ (since f is called). Moreover the type of x must be the same as the parameter type τ_1 of that function type (since x is the argument of the call to f). However, we don't have enough information to determine the specific type τ_1 and what the specific return type τ_2 of f is. In fact, for the evaluation of a call to the function *apply* the specific types of the parameter and return value of f don't matter. We say that *apply* is parameterized in the types τ_1 and τ_2.

To deal with type parameterization, we extend our language of types with type variables that serve as placeholders for other types. We use Greek letters to denote such type variables. The new type language is as follows:

$\alpha \in \mathit{TVar}$	type variables
$\tau \in \mathit{Typ} ::= \textbf{Bool} \mid \textbf{Num} \mid \alpha \mid \tau_1 \Rightarrow \tau_2$	types

If a type τ does not contain any type variables, it is called *monomorphic*, otherwise it is called *polymorphic*. We can think of polymorphic types as types that parameterize over monomorphic types. For example, the type $\alpha \Rightarrow \textbf{Bool}$ stands for the monomorphic types $\textbf{Bool} \Rightarrow \textbf{Bool}$, $\textbf{Num} \Rightarrow \textbf{Bool}$, $(\textbf{Bool} \Rightarrow \textbf{Bool}) \Rightarrow \textbf{Bool}$, etc.

A Complete Example

To infer the type of a given expression e we now proceed in three steps:

1. Associate a fresh type variable with each subexpression occurring in e.
2. Generate a set of equality constraints over types from the syntactic structure of e. These typing constraints relate the introduced type variables with each other and impose restrictions on the types that they stand for.
3. Solve the generated typing constraints. If a solution of the constraints exists, the expression is well-typed and we can read off the types of all subexpressions (including e itself) from the computed solution. Otherwise, if no solution exists, e has a type error.

We explain these three steps using our initial example:

$$x \texttt{ === } 0 \texttt{ ? } 1 \texttt{ : } x \texttt{*} \mathit{fac}\texttt{(}x \texttt{-} 1\texttt{)}$$

Let us call this expression e. We generate the type variables and typing constraints for the subexpressions of e in one go:

x	α_x	-
0	α_0	$\alpha_0 \doteq \texttt{Num}$
$x \texttt{ === } 0$	α_{eq}	$\alpha_x \doteq \alpha_0$, $\alpha_{eq} \doteq \texttt{Bool}$,
1	α_1	$\alpha_1 \doteq \texttt{Num}$
fac	α_{fac}	-
$x \texttt{-} 1$	$\alpha_{\texttt{-}}$	$\alpha_x \doteq \texttt{Num}$, $\alpha_1 \doteq \texttt{Num}$,
		$\alpha_{\texttt{-}} \doteq \texttt{Num}$
$\mathit{fac}\texttt{(}x \texttt{-} 1\texttt{)}$	α_{call}	$\alpha_{fac} \doteq (\alpha_{\texttt{-}} \Rightarrow \alpha_{call})$
$x \texttt{*} \mathit{fac}\texttt{(}x \texttt{-} 1\texttt{)}$	$\alpha_{\texttt{*}}$	$\alpha_x \doteq \texttt{Num}$, $\alpha_{call} \doteq \texttt{Num}$,
		$\alpha_{\texttt{*}} \doteq \texttt{Num}$
$x \texttt{ === } 0 \texttt{ ? } 1 \texttt{ : } x \texttt{*} \mathit{fac}\texttt{(}x \texttt{-} 1\texttt{)}$	α_{ite}	$\alpha_{eq} \doteq \texttt{Bool}$, $\alpha_{ite} \doteq \alpha_1$, $\alpha_{ite} \doteq \alpha_{\texttt{*}}$

Each row lists one of e's subexpressions, the type variable that stands for the type of that subexpression, and a list of equality constraints that constrain the type variable with the type variables of other subexpressions. To avoid notational confusion with actual equality on types in our mathematical meta language, we use the symbol $\doteq$ to equate types in the typing constraints. Note that the subexpressions x and 1, which occur multiple times in e, are only listed once.

We explain two of the rows in the above table in more detail. The subexpression $x \texttt{-} 1$ of e has the associated type variable $\alpha_{\texttt{-}}$. Since we know that the arguments of the binary operator `-` must be of type **`Num`**, we obtain two typing constraints $\alpha_x \doteq \texttt{Num}$ and $\alpha_1 \doteq \texttt{Num}$, where α_x and α_1 are the type variables associated with x and 1, respectively. Similarly, we know that the result of operator `-` is again a value of type **`Num`**. Hence, we obtain the additional constraint $\alpha_{\texttt{-}} \doteq \texttt{Num}$. Another particularly interesting case is the call expression $\mathit{fac}\texttt{(}x \texttt{-} 1\texttt{)}$.

From this expression we can infer that *fac* must be a function whose parameter type matches the type of the subexpression $x - 1$ (which is the argument of the call) and whose result type matches the type of the entire call expression (which is denoted by α_{call}). This information is captured by the typing constraint

$$\alpha_{fac} \doteq (\alpha_{-} \Rightarrow \alpha_{call}).$$

If we collect the typing constraints that have been generated for all of e's subexpressions from the table above, we obtain the following set of constraints:

1. $\alpha_0 \doteq \texttt{Num}$
2. $\alpha_x \doteq \alpha_0$
3. $\alpha_{eq} \doteq \texttt{Bool}$
4. $\alpha_1 \doteq \texttt{Num}$
5. $\alpha_{-} \doteq \texttt{Num}$
6. $\alpha_{fac} \doteq (\alpha_{-} \Rightarrow \alpha_{call})$
7. $\alpha_{call} \doteq \texttt{Num}$
8. $\alpha_{\star} \doteq \texttt{Num}$
9. $\alpha_{eq} \doteq \texttt{Bool}$
10. $\alpha_{ite} \doteq \alpha_1$
11. $\alpha_{ite} \doteq \alpha_{-}$

We denote this set of constraints by C. To see whether e is well-typed, we have to solve C. That is, we have to find a mapping $\sigma : \mathit{TVar} \to \mathit{Typ}$ from type variables to types, such that if we substitute the type variables occuring in each constraint according to σ, then the two sides of each constraint become equal. That is, σ must satisfy that for all constraints $\tau_1 \doteq \tau_2 \in C$, $\tau_1\sigma = \tau_2\sigma$. Here, $\tau\sigma$ denotes substitution of the type variables occurring in a type τ according to σ. We refer to the problem of finding such a solution σ for a given set of typing constraints C as the *unification problem*. A solution σ of a unification problem instance C is called a *unifier* of C.

We can compute a unifier σ from the set of constraints C using a simple iterative algorithm. The algorithm starts with a trivial candidate solution $\sigma_0 = \emptyset$ and then processes the equality constraints one at a time, extending σ_0 to an actual unifier of C. We describe this algorithm below (see Figure 6.4). In the following, we show how it works for our concrete example.

As noted, we start with the trivial candidate solution $\sigma_0 = \emptyset$ and process the constraints one at a time, extending σ_0 as we go along. The order in which the constraints are processed does not matter. We choose to process them in the order given above. That is, the first constraint that we consider is

1. $\alpha_0 \doteq \texttt{Num}$

Observe that the left-hand side of this constraint is a type variable α_0. Thus, to solve this specific constraint, all we need to do is map α_0 to the type on the right-hand side, which is $\texttt{Num}$. We therefore define $\sigma_1 = \sigma_0[\alpha_0 \mapsto \texttt{Num}]$. Observe, that σ_1 is indeed a unifier of constraint 1, since

$$\alpha_0\sigma_1 = \texttt{Num} = \texttt{Num}\,\sigma_1$$

More generally, the unification algorithm will maintain the invariant that after processing the ith constraint, the current candidate solution σ_i will unify all previously processed constraints 1 to i.

We maintain another invariant in our algorithm, namely that any type variable that is assigned by our current candidate solution no longer occurs in any of the constraints that still need to be processed. To ensure this invariant, we have to apply the candidate unifier σ_i to the unprocessed constraints after each extension. That is, in our example, we substitute α_0 by **Num** in constraints 2 to 11, which gives us the new set of constraints:

2. $\alpha_x \doteq \texttt{Num}$
3. $\alpha_{eq} \doteq \texttt{Bool}$
4. $\alpha_1 \doteq \texttt{Num}$
5. $\alpha_{-} \doteq \texttt{Num}$
6. $\alpha_{fac} \doteq (\alpha_{-} \Rightarrow \alpha_{call})$
7. $\alpha_{call} \doteq \texttt{Num}$
8. $\alpha_{*} \doteq \texttt{Num}$
9. $\alpha_{eq} \doteq \texttt{Bool}$
10. $\alpha_{ite} \doteq \alpha_1$
11. $\alpha_{ite} \doteq \alpha_{-}$

We continue processing the constraints in the given order. The constraints 2 to 6 are similar to constraint 1. We extend the candidate unifier σ_0 for each of these cases as described above to obtain the following remaining constraints and current candidate unifier σ_6:

7. $\alpha_{call} \doteq \texttt{Num}$
8. $\alpha_{*} \doteq \texttt{Num}$
9. $\texttt{Bool} \doteq \texttt{Bool}$
10. $\alpha_{ite} \doteq \texttt{Num}$
11. $\alpha_{ite} \doteq \texttt{Num}$

$$\begin{aligned}\sigma_6 = \{\, & \alpha_0 \mapsto \texttt{Num},\\ & \alpha_x \mapsto \texttt{Num},\\ & \alpha_{eq} \mapsto \texttt{Bool},\\ & \alpha_1 \mapsto \texttt{Num},\\ & \alpha_{-} \mapsto \texttt{Num},\\ & \alpha_{fac} \mapsto (\texttt{Num} \Rightarrow \alpha_{call})\}\end{aligned}$$

The case for constraint 7 is similar to the previous cases, except that the type variable α_{call} now also appears in the type to which α_{fac} is mapped by the current candidate unifier σ_6. In order to maintain our invariant that the candidate unifier unifies all constraints processed so far, we also have to apply the mapping $\alpha_{call} \mapsto \texttt{Num}$ to σ_6 before we extend σ_6 with this new mapping.

After we do this, we obtain the new mapping:

$$\begin{aligned}\sigma_7 = \{\ &\alpha_0 \mapsto \texttt{Num},\\ &\alpha_x \mapsto \texttt{Num},\\ &\alpha_{eq} \mapsto \texttt{Bool},\\ &\alpha_1 \mapsto \texttt{Num},\\ &\alpha_- \mapsto \texttt{Num},\\ &\alpha_{fac} \mapsto (\texttt{Num} \Rightarrow \texttt{Num}),\\ &\alpha_{call} \mapsto \texttt{Num}\}\end{aligned}$$

The unprocessed constraints 8 to 11 remain unchanged since α_{call} does not appear in any of these. The case for constraint 8 is again similar to the first cases, so we process it to obtain the new candidate unifier $\sigma_8 = \sigma_7[\alpha_\star \mapsto \texttt{Num}]$.

Constraint 9 is again interesting. It is now of the form $\texttt{Bool} \doteq \texttt{Bool}$. The two sides of this constraint are already unified, so there is no need to extend σ_8. We thus simply define $\sigma_9 = \sigma_8$ and proceed with constraint 10. Processing constraint 10 again extends the candidate unifier, after which constraint 11 becomes trivially unified. After processing all constraints we obtain the following mapping

$$\begin{aligned}\sigma_{11} = \{\ &\alpha_0 \mapsto \texttt{Num},\ \alpha_x \mapsto \texttt{Num},\ \alpha_{eq} \mapsto \texttt{Bool},\ \alpha_1 \mapsto \texttt{Num},\\ &\alpha_- \mapsto \texttt{Num},\ \alpha_{fac} \mapsto (\texttt{Num} \Rightarrow \texttt{Num}),\\ &\alpha_{call} \mapsto \texttt{Num},\ \alpha_\star \mapsto \texttt{Num},\ \alpha_{ite} \mapsto \texttt{Num}\}\end{aligned}$$

Observe that this mapping is indeed a unifier for the original set of constraints. Thus, we have shown that the expression e is well-typed.

Inferring Polymorphic Types

Next, we apply the type inference algorithm to our polymorphic example:

$$apply = x \texttt{ => } f \texttt{ => } f\texttt{(}x\texttt{)}$$

From the expression *apply* we collect the following subexpressions with associated type variables and typing constraints:

f	α_f	-
x	α_x	-
$f\texttt{(}x\texttt{)}$	α_{call}	$\alpha_f \doteq (\alpha_x \Rightarrow \alpha_{call})$
$f \texttt{ => } f\texttt{(}x\texttt{)}$	α_{fun}	$\alpha_{fun} \doteq (\alpha_f \Rightarrow \alpha_{call})$
$x \texttt{ => } f \texttt{ => } f\texttt{(}x\texttt{)}$	α_{apply}	$\alpha_{apply} \doteq (\alpha_x \Rightarrow \alpha_{fun})$

That is, we need to solve the following unification problem to show that *apply* is well-typed:

1. $\alpha_f \doteq (\alpha_x \Rightarrow \alpha_{call})$
2. $\alpha_{fun} \doteq (\alpha_f \Rightarrow \alpha_{call})$
3. $\alpha_{apply} \doteq (\alpha_x \Rightarrow \alpha_{fun})$

Starting with the trivial candidate unifier $\sigma_0 = \emptyset$ we process the first two constraints as described before to obtain the following updated candidate unifier:

$$\begin{aligned}\sigma_2 = \{\ & \alpha_f \mapsto \alpha_x \Rightarrow \alpha_{call} \\ & \alpha_{fun} \mapsto \alpha_x \Rightarrow \alpha_{call} \Rightarrow \alpha_{call}\}\end{aligned}$$

The remaining constraint 3 now looks as follows:

3. $\alpha_{apply} \doteq (\alpha_x \Rightarrow (\alpha_x \Rightarrow \alpha_{call}) \Rightarrow \alpha_{call})$

Processing this remaining constraint yields the actual unifier of the original set of constraints:

$$\begin{aligned}\sigma_3 = \{\ & \alpha_f \mapsto \alpha_x \Rightarrow \alpha_{call} \\ & \alpha_{fun} \mapsto (\alpha_x \Rightarrow \alpha_{call}) \Rightarrow \alpha_{call} \\ & \alpha_{apply} \mapsto (\alpha_x \Rightarrow (\alpha_x \Rightarrow \alpha_{call}) \Rightarrow \alpha_{call})\}\end{aligned}$$

We obtain the inferred type of *apply* by looking up the type to which the associated type variable α_{apply} is mapped by the unifier σ_3. Note that this type is polymorphic as it still contains the type variables α_x and α_{call}. This tells us that we can safely call *apply* with any arguments x and f, as long as f is a function whose parameter type matches the type of x. In particular, both of the following specific usages of *apply* are safe:

$$\mathit{apply}\texttt{(3)(}x\ \texttt{=>}\ x + 2\texttt{)}$$
$$\mathit{apply}\texttt{(true)(}x\ \texttt{=>}\ x\ \texttt{||}\ \texttt{false)}$$

Detecting Type Errors

So far we have only considered cases in which the type inferences succeeded. The question remains what happens if an expression is not well-typed and how to detect this during unification. To this end, we consider another example:

$$x \ \texttt{?}\ x + 1 \ \texttt{:}\ 3$$

From this expression, we generate the following typing constraints:

1. $\alpha_{+} \doteq \texttt{Num}$
2. $\alpha_x \doteq \texttt{Num}$
3. $\alpha_1 \doteq \texttt{Num}$
4. $\alpha_3 \doteq \texttt{Num}$

5. $\alpha_x \doteq \texttt{Bool}$
6. $\alpha_{ite} \doteq \alpha_{+}$
7. $\alpha_{ite} \doteq \alpha_3$

After processing the first 4 constraints we obtain the following candidate unifier

$$\sigma_4 = \{\alpha_{+} \mapsto \texttt{Num}, \alpha_x \mapsto \texttt{Num}, \alpha_1 \mapsto \texttt{Num}, \alpha_3 \mapsto \texttt{Num}\}$$

and the remaining set of constraints now looks as follows:

5. $\texttt{Num} \doteq \texttt{Bool}$
6. $\alpha_{ite} \doteq \texttt{Num}$
7. $\alpha_{ite} \doteq \texttt{Num}$

Continuing with constraint 5, we detect a problem. The constraint is now of the form $\texttt{Num} \doteq \texttt{Bool}$. Since `Num` and `Bool` are two distinct monomorphic types, there exists no unifier that can make these distinct types equal. This means that the generated unification problem has no solution. We therefore abort and report a type error in the original expression. If we look at the original expression from which we generated the typing constraints, we observe that the problem comes from the two usages of x. First, we use x in the test of the conditional expression, which means that x must have type `Bool`. Then we use x again in the "then" branch as an argument to `+`, which means that x must also have type `Num` – a contradiction.

Self Application and Occurrence Check

There is one specific kind of type error that has to do with the restrictions on the degree of polymorphism that we allow in our type language. To explain this issue, consider the following expression where we call a variable x on itself:

$$x(x)$$

We refer to this kind of expression as self application. From this expression, we generate a single typing constraint:

$$\alpha_x \doteq \alpha_x \Rightarrow \alpha_{call}$$

On first sight, it appears that this constraint has a simple solution given by the following mapping:

$$\sigma = \{\alpha_x \mapsto \alpha_x \Rightarrow \alpha_{call}\}$$

However, observe that σ is not actually a unifier of the constraint since applying σ to the two sides of the constraint does not make the two sides equal:

$$\begin{aligned}\alpha_x\sigma &= \alpha_x \Rightarrow \alpha_{call}\\ &\neq (\alpha_x \Rightarrow \alpha_{call}) \Rightarrow \alpha_{call}\\ &= (\alpha_x \Rightarrow \alpha_{call})\sigma\end{aligned}$$

In fact, there is no mapping of the type variables α_x and α_{call} to any type in our language such that the two sides of the constraint would be equal. The expression $x(x)$ is therefore not well-typed. The reason for this restriction is that type variables in polymorphic types stand for monomorphic types only rather than arbitrary types. That is, in our type language we do not consider polymorphic types that are parameterized by other polymorphic types. Such more general polymorphic types are referred to as *higher-ranked polymorphic types*. While we could make our type system more general and allow higher-ranked polymorphic types, we would no longer be able to solve the type inference problem and the programmer would again have to provide type annotations.

We can detect situations such as self application by introducing an additional check in our unification algorithm. When we process a constraint that is of the form $\alpha \doteq \tau$, we first check whether α occurs in τ before we extend the current candidate unifier with the mapping $\alpha \mapsto \tau$. If α occurs in τ, we abort the algorithm and report a type error. We refer to this additional check as the *occurrence check*.

6.3.2 The Hindley-Milner Type System

We formalize the Hindley-Milner type system using a simplified version of the language from Chapter 5. Specifically, we drop equality operators and **`const`** declarations from the syntax. We will discuss later how these constructs can be added back to the language. The simplified grammar of our new language is as follows

$$\begin{array}{rll}
n \in \mathit{Num} & & \text{Nums (double)} \\
x \in \mathit{Var} & & \text{variables} \\
b \in \mathit{Bool} ::= & \mathbf{true} \mid \mathbf{false} & \text{Booleans} \\
v \in \mathit{Val} ::= & n \mid b \mid \mathbf{function}\, p(x)\, e & \text{values} \\
e \in \mathit{Expr} ::= & x \mid v \mid e_1\, \mathit{bop}\, e_2 \mid e_1 \;?\; e_2 \;:\; e_3 \mid e_1(e_2) & \text{expressions} \\
\mathit{bop} \in \mathit{Bop} ::= & \texttt{+} \mid \texttt{*} \mid \texttt{\&\&} \mid \texttt{||} & \text{binary operators} \\
p ::= & x \mid \epsilon & \text{function names}
\end{array}$$

Compared to the language from Section 6.1, the new language does not support type annotations for function abstractions. We could allow such annotations as a form of optional code documentation. However, they are not needed for the type inference. Hence, we omit them for the sake of simplicity.

Typing Constraint Generation

We describe the actual constraint generation for the type inference using a modified version of the typing relation of our simple type system. The new typing relation is denoted by judgments of the form

$$\Gamma \vdash e : \alpha \mid C$$

The typing environment Γ and the expression e are the input of the relation. The output computed by the relation is the type variable α and the set of typing constraints C. Informally, this judgment states that under typing environment Γ, the expression e is well-typed, provided that the typing constraints C have a solution σ. In particular, given a solution σ of C, the type of e for this solution is $\sigma(\alpha)$.

The typing environment Γ is simply a mapping from (expression) variables to type variables, i.e., $\Gamma : \mathit{Var} \rightharpoonup \mathit{TVar}$. We need the typing environment to make sure that we associate the same type variable $\Gamma(x)$ with each free occurrence of a variable x in e.

The inference rules that define the typing constraint generation relation are given in Figure 6.3. It is instructive to compare these rules with the corresponding rules for the typing relation of our simple type system discussed in Section 6.1. Also note that the constraints that we generated for the examples in the previous section exactly correspond to those obtained by the rules in Figure 6.3 (modulo renaming of type variables).

Unification Algorithm

Figure 6.4 formalizes the unification algorithm that we used to solve the generated typing constraints in the examples. The algorithm is split into two functions: *unify* and *unifyOne*. The function *unify* takes a set of typing constraints C and a candidate unifier σ and returns an extension of σ to an actual unifier of C, or $\bot$ if no such unifier exists. The actual unification is done by the function *unifyOne*, which extends the given candidate unifier to a unifier for a single constraint $\tau_1 \doteq \tau_2$. The function *unify* applies the function *unifyOne* to the individual constraints in C, one at a time. The defining cases for *unify* and *unifyOne* should be read top-down. The first matching case applies, similar to a match expression in a Scala program.

In the case for $\alpha \doteq \tau$ of *unifyOne*, the actual extension of the candidate unifier σ happens. The test $\alpha \in \mathit{tv}(\tau)$ implements the occurrence check. Here, we denote by tv the function that takes a type expression τ and returns the set of type variables occurring in τ, e.g., $\mathit{tv}(\alpha \Rightarrow \beta) = \{\alpha, \beta\}$. If the occurrence check fails, *unifyOne* returns $\bot$ to indicate that the constraint cannot be unified. If the check succeeds, *unifyOne* returns the extension of the candidate unifier σ, which is given by the expression:

$$(\lambda\beta.\, \sigma(\beta)[\tau/\alpha])[\alpha \mapsto \tau]$$

The notation $\lambda x.\, e$ stands for an anonymous function. That is, the candidate unifier is a function σ'' that is obtained from σ in two steps. First, σ is updated to σ' by applying the mapping $\alpha \mapsto \tau$ to all current mappings $\beta \mapsto \sigma(\beta)$ in σ using type variable substitution:

$$\begin{aligned} \sigma' &: \mathit{TVar} \rightharpoonup \mathit{Typ} \\ \sigma'(\beta) &= \sigma(\beta)[\tau/\alpha] \end{aligned}$$

$$\textsc{HMBool}\ \frac{\alpha \text{ fresh}}{\Gamma \vdash b : \alpha \mid \{\alpha \doteq \texttt{Bool}\}} \qquad \textsc{HMNum}\ \frac{\alpha \text{ fresh}}{\Gamma \vdash n : \alpha \mid \{\alpha \doteq \texttt{Num}\}} \qquad \textsc{HMVar}\ \Gamma \vdash x : \Gamma(x) \mid \emptyset$$

$$\textsc{HMAndOr}\ \frac{\Gamma \vdash e_1 : \alpha_1 \mid C_1 \qquad \Gamma \vdash e_2 : \alpha_2 \mid C_2 \qquad bop \in \{\texttt{\&\&}, \texttt{||}\} \qquad \alpha \text{ fresh}}{\Gamma \vdash e_1\, bop\, e_2 : \alpha \mid C_1 \cup C_2 \cup \{\alpha \doteq \texttt{Bool}, \alpha_1 \doteq \texttt{Bool}, \alpha_2 \doteq \texttt{Bool}\}}$$

$$\textsc{HMArith}\ \frac{\Gamma \vdash e_1 : \alpha_1 \mid C_1 \qquad \Gamma \vdash e_2 : \alpha_2 \mid C_2 \qquad bop \in \{\texttt{+}, \texttt{*}\} \qquad \alpha \text{ fresh}}{\Gamma \vdash e_1\, bop\, e_2 : \alpha \mid C_1 \cup C_2 \cup \{\alpha \doteq \texttt{Num}, \alpha_1 \doteq \texttt{Num}, \alpha_2 \doteq \texttt{Num}\}}$$

$$\textsc{HMIf}\ \frac{\Gamma \vdash e_1 : \alpha_1 \mid C_1 \qquad \Gamma \vdash e_2 : \alpha_2 \mid C_2 \qquad \Gamma \vdash e_3 : \alpha_3 \mid C_3 \qquad \alpha \text{ fresh}}{\Gamma \vdash e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 : \alpha \mid C_1 \cup C_2 \cup C_3 \cup \{\alpha_1 \doteq \texttt{Bool}, \alpha \doteq \alpha_2, \alpha \doteq \alpha_3\}}$$

$$\textsc{HMCall}\ \frac{\Gamma \vdash e_1 : \alpha_1 \mid C_1 \qquad \Gamma \vdash e_2 : \alpha_2 \mid C_2 \qquad \alpha \text{ fresh}}{\Gamma \vdash e_1\texttt{(}e_2\texttt{)} : \alpha \mid C_1 \cup C_2 \cup \{\alpha_1 \doteq (\alpha_2 \Rightarrow \alpha)\}}$$

$$\textsc{HMFun}\ \frac{\Gamma' = \Gamma[x \mapsto \alpha_1] \qquad \Gamma' \vdash e : \alpha_2 \mid C \qquad \alpha, \alpha_1 \text{ fresh}}{\Gamma \vdash \texttt{function(}x\texttt{)}\ e : \alpha \mid C \cup \{\alpha \doteq (\alpha_1 \Rightarrow \alpha_2)\}}$$

$$\textsc{HMFunRec}\ \frac{\Gamma' = \Gamma[x \mapsto \alpha][y \mapsto \alpha_1] \qquad \Gamma' \vdash e : \alpha_2 \mid C \qquad \alpha, \alpha_1 \text{ fresh}}{\Gamma \vdash \texttt{function}\ x\texttt{(}y\texttt{)}\ e : \alpha \mid C \cup \{\alpha \doteq (\alpha_1 \Rightarrow \alpha_2)\}}$$

Figure 6.3: Typing constraint generation rules for the Hindley-Milner type system

```
$\mathit{unify}(C) = \mathit{unify}(C, \emptyset)$
$\mathit{unify}(\emptyset, \sigma) = \sigma$
$\mathit{unify}(C, \bot) = \bot$
$\mathit{unify}(\{\tau_1 \doteq \tau_2\} \cup C, \sigma) =$
    $\mathbf{let}\ \sigma' = \mathit{unifyOne}(\tau_1 \doteq \tau_2, \sigma)\ \mathbf{in}$
    $\mathbf{if}\ \sigma' = \bot\ \mathbf{then}\ \bot\ \mathbf{else}\ \mathit{unify}(C\sigma', \sigma')$
$\mathit{unifyOne}(\alpha \doteq \alpha, \sigma) = \sigma$
$\mathit{unifyOne}(\alpha \doteq \tau, \sigma) =$
    $\mathbf{if}\ \alpha \in \mathit{tv}(\tau)\ \mathbf{then}\ \bot\ \mathbf{else}\ (\lambda\beta.\, \sigma(\beta)[\tau/\alpha])[\alpha \mapsto \tau]$
$\mathit{unifyOne}(\tau \doteq \alpha, \sigma) = \mathit{unifyOne}(\alpha = \tau, \sigma)$
$\mathit{unifyOne}((\tau_1 \Rightarrow \tau_2) \doteq (\tau_1' \Rightarrow \tau_2'), \sigma) =$
    $\mathit{unify}(\{\tau_1 \doteq \tau_1', \tau_2 \doteq \tau_2'\}, \sigma)$
$\mathit{unifyOne}(\texttt{Num} \doteq \texttt{Num}, \sigma) = \sigma$
$\mathit{unifyOne}(\texttt{Bool} \doteq \texttt{Bool}, \sigma) = \sigma$
$\mathit{unifyOne}(\tau_1 \doteq \tau_2, \sigma) = \bot$
```

Figure 6.4: A simple unification algorithm

Note that this definition of σ' is equivalent to the following definition which defines σ' in terms of an anonymous function:

$$\sigma' = \lambda\beta.\, \sigma(\beta)[\tau/\alpha]$$

Then σ'' is obtained from σ' by extending it with the new mapping $\alpha \mapsto \tau$:

$$\begin{aligned} \sigma'' &: \mathit{TVar} \rightharpoonup \mathit{Typ} \\ \sigma'' &= \sigma'[\alpha \mapsto \tau]. \end{aligned}$$

The case $(\tau_1 \Rightarrow \tau_2) \doteq (\tau_1' \Rightarrow \tau_2')$ of *unifyOne* handles situations in which we need to unify a constraint that equates two function types. In this case, we simply need to recursively solve a new unification problem for the set of constraints $C' = \{\tau_1 \doteq \tau_1', \tau_2 \doteq \tau_2'\}$. The recursion is well-defined (i.e., *unify* and *unifyOne* always terminate) since we only decompose function types in typing constraints but never create new function types during unification.

The final "catch-all" case of *unifyOne* handles all the cases $\tau_1 \doteq \tau_2$ where τ_1 and τ_2 cannot be unified, such as $\texttt{Bool} = \texttt{Num}$, etc.

If we analyze the complexity of our simple unification algorithm we observe that it is worst-case quadratic in the number of typing constraints. This is because each time we extend the candidate unifier σ for a constraint $\alpha \doteq \tau$, we have to iterate over both the existing mappings in σ as well as the unprocessed constraints. Practical implementations of the Hindley-Milner type system use more efficient unification algorithms that use a union-find data structure to represent the candidate unifier. Using this data structure the extension of the candidate unifier can be implemented more efficiently. In fact, these practical algorithms run in quasilinear time.

Parametric Polymorphism

We have not yet fully described how we can actually use expressions with polymorphic types in our programs and how such expressions are handled by the type inference algorithm. In the Hindley-Milner type system, the introduction of expressions with polymorphic types is closely tied to **const** declarations. Consider the following example

```
const id = x => x;
id(true) ? id(1) : 0
```

Recall from our discussion in Section 6.1.5 that this expression cannot be typed in our simple type system with annotated monomorphic types. However, it is well-typed in the Hindley-Milner type system.

The constraint generation for a **const** declaration works as follows. First, we generate the constraints for the defining expression of the declared variable as described before. In the example, the defining expression of *id* is *x* => *x*, which generates the following single typing constraint:

$$\alpha_{id} \doteq \alpha_x \Rightarrow \alpha_x$$

Here, α_{id} represents the actual type of *id*. When we generate the typing constraints for the body of the **const** declaration, we do not simply reuse the same type variable α_{id} for each usage of *id* in the body. Instead, we use a fresh copy of the type variable and generate a fresh copy of the constraints obtained from the defining expression for each usage of *id*. Here, "fresh copy" means that we consistently substitute all the type variables in the constraints by fresh type variables. The copying of constraints ensures that the inferred type for each usage of *id* in the body of the declaration is consistent with the constraints imposed by *id*'s definition. However, different usages of *id* in the body do not interfere. In total, we end up with the following set of constraints for the complete expression in our example:

1. $\alpha_{id} \doteq \alpha_x \Rightarrow \alpha_x$
2. $\alpha_{id,1} \doteq \alpha_{\texttt{true}} \Rightarrow \alpha_{call,1}$
3. $\alpha_{id,1} \doteq \alpha_{x,1} \Rightarrow \alpha_{x,1}$
4. $\alpha_{\texttt{true}} \doteq \texttt{Bool}$
5. $\alpha_{call,1} = \texttt{Bool}$

6. $\alpha_{id,2} \doteq \alpha_1 \Rightarrow \alpha_{call,2}$
7. $\alpha_{id,2} \doteq \alpha_{x,2} \Rightarrow \alpha_{x,2}$
8. $\alpha_1 \doteq \texttt{Num}$
9. $\alpha_0 \doteq \texttt{Num}$
10. $\alpha_{call,1} \doteq \texttt{Bool}$
11. $\alpha_{ite} \doteq \alpha_{call,2}$
12. $\alpha_{ite} \doteq \alpha_0$

Note that the constraints 3 and 7 are the fresh copies of constraint 1 for the two usages of *id* in the body of the **`const`** declaration. The resulting unification problem has a solution, which means that the expression is indeed well-typed.

To see the limitations of the kind of parametric polymorphism that is supported by the Hindley-Milner type system, contrast the expression

```
const id = x => x;
id(true) ? id(1) : 0
```

with the expression

```
(id => id(true) ? id(1) : 0)(x => x)
```

From a semantic point of view the two expressions are equivalent. We have simply replaced the **`const`** declaration of *id* by function abstraction over *id* in the body of the declaration, followed by a call that immediately binds *id*'s defining expression to the parameter of the obtained function. As we have seen, the first expression is well-typed. However, the second one is not. The problem is that during the typing constraint generation for the second expression, the two occurrences of *id* in the function body are bound to the same type variable. That is, the generated constraints require that *id* is at the same time of type

$$\texttt{Bool} \Rightarrow \texttt{Bool}$$

and of type

$$\texttt{Num} \Rightarrow \texttt{Num}$$

This is not possible. In order to support such expressions in the type system, we would have to consider the more general form of higher-ranked parametric polymorphism. The price we would have to pay for this generality is that type inference would no longer be possible in all cases.

The constraints that come from the defining expression of a **`const`** variable x are copied for each usage of x in the body of the declaration of x. This means that the size of the generated constraints can grow exponentially with the nesting depth of **`const`** declarations in defining expressions. In fact, the type inference problem for the Hindley-Milner type system with `const` declarations is known to be EXPTIME-complete. So this exponential blow-up cannot be avoided in general. Actual implementations of the type system avoid this blow-up in practice by solving the generated constraints on the fly instead of separating the constraint generation phase from the unification phase.