6

Type Systems, Type Inference, and Polymorphism

Programming involves a wide range of computational constructs, such as data structures, functions, objects, communication channels, and threads of control. Because programming languages are designed to help programmers organize computational constructs and use them correctly, many programming languages organize data and computations into collections called types. In this chapter, we look at the reasons for using types in programming languages, methods for type checking, and some typing issues such as polymorphism and type equality. A large section of this chapter is devoted to type inference, the process of determining the types of expressions based on the known types of some symbols that appear in them. Type inference is a generalization of type checking, with many characteristics in common, and a representative example of the kind of algorithms that are used in compilers and programming environments to determine properties of programs. Type inference also provides an introduction to polymorphism, which allows a single expression to have many types.

6.1 TYPES IN PROGRAMMING

In general, a *type* is a collection of computational entities that share some common property. Some examples of types are the type `Int` of integers, the type `Int→Int` of functions from integers to integers, and the Pascal subrange type `[1 .. 100]` of integers between 1 and 100. In concurrent ML there is the type `Chan Int` of communication channels carrying integer values and, in Java, a hierarchy of types of exceptions.

There are three main uses of types in programming languages:

- naming and organizing concepts,
- making sure that bit sequences in computer memory are interpreted consistently,
- providing information to the compiler about data manipulated by the program.

These ideas are elaborated in the following subsections.

Although some programming language descriptions will say things like, “Lisp is an untyped language,” there is really no such thing as an untyped programming language. In Lisp, for example, lists and atoms are two different types: list operations can be applied to lists but not to atoms. Programming languages do vary a great deal, however, in the ways that types are used in the syntax and semantics (implementation) of the language.

6.1.1 Program Organization and Documentation

A well-designed program uses concepts related to the problem being solved. For example, a banking program will be organized around concepts common to banks, such as accounts, customers, deposits, withdrawals, and transfers. In modern programming languages, customers and accounts, for example, can be represented as separate types. Type checking can then check to make sure that accounts and customers are treated separately, with account operations applied to accounts but not used to manipulate customers. Using types to organize a program makes it easier for someone to read, understand, and maintain the program. Types therefore serve an important purpose in documenting the design and intent of the program.

An important advantage of type information, in comparison with comments written by a programmer, is that types may be checked by the programming language interpreter or compiler. Type checking guarantees that the types written into a program are correct. In contrast, many programs contain incorrect comments, either because the person writing the explanation was careless or because the program was later changed but the comments were not.

6.1.2 Type Errors

A *type error* occurs when a computational entity, such as a function or a data value, is used in a manner that is inconsistent with the concept it represents. For example, if an integer value is used as a function, this is a type error. A common type error is to apply an operation to an operand of the wrong type. For example, it is a type error to use integer addition to add a string to an integer. Although most programmers have a general understanding of type errors, there are some subtleties that are worth exploring.

Hardware Errors. The simplest kind of type error to understand is a machine instruction that results in a hardware error. For example, executing a "function call"

```
x()
```

is a type error if `x` is not a function. If `x` is an integer variable with value 256, for example, then executing `x()` will cause the machine to jump to location 256 and begin executing the instructions stored at that place in memory. If location 256 contains data that do not represent a valid machine instruction, this will cause a hardware interrupt. Another example of a hardware type error occurs in executing an operation

```
float_add(3, 4.5)
```

where the hardware floating-point unit is invoked on an integer argument 3. Because the bit pattern used to represent 3 does not represent a floating-point number in the form expected by the floating-point hardware, this instruction will cause a hardware interrupt.

Unintended Semantics. Some type errors do not cause a hardware fault or interrupt because compiled code does not contain the same information as the program source code does. For example, an operation

```
int_add(3, 4.5)
```

is a type error, as int_add is an integer operation and is applied here to a floating-point number. Most hardware would perform this operation, however, because the bits used to represent 4.5 are a legal integer bit pattern. The integer that this bit pattern represents, however, is not mathematically related to 4.5, so the operation is not meaningful. More specifically, int_add is intended to perform integer addition, but the result of int_add(3, 4.5) is not the arithmetic sum of the two operands.

The error associated with int_add(3, 4.5) may become clearer if we think about how a program might apply integer addition to a floating-point argument. To be concrete, suppose a program defines a function f that adds three to its argument,

```
f x = 3 + x
```

and someplace within the scope of this definition we also declare a floating-point value z:

```
z :: Float = 4.5
```

If the programming language compiler or interpreter allows the call f z and the language does not automatically convert floating-point numbers to integers in this situation, then the function call f z will cause a run-time type error because int_add(3, 4.5) will be executed. This situation is a type error because integer addition is applied to a non-integer argument.

The reason why many people find the concept of type error confusing is that type errors generally depend on the concepts defined in a program or programming language, not the way that programs are executed on the underlying hardware. To be specific, it is just as much of a type error to apply an integer operation to a floating-point argument as it is to apply a floating-point operation to an integer argument. It does not matter which causes a hardware interrupt on any particular computer.

Inside a computer, all values are stored as sequences of bytes of bits. Because integers and floating-point numbers are stored as four bytes on many machines, some integers and floating-point numbers overlap; a single bit pattern may represent an integer when it is used one way and a floating-point number when it is used in another. Nonetheless, a type error occurs when a pattern that is stored in the computer for the purpose of representing one type of value is used as the representation of another type of value.

6.1.3 Types and Optimization

Type information in programs can be used for many kinds of optimizations. One example is finding components of records (as they are called in Haskell and ML) or structs (as they are called in C). The component-finding problem also arises in object-oriented languages. A record consists of a set of entries of different types. For example, a student record may contain a student name of type `String` and a student number of type `Integer`, written here as Haskell type declaration:

```
data Student = Student {name :: String, number :: Int}
```

In a program that manipulates records, there might be an expression of the form `name r`, meaning the name field of the record `r`. A compiler must generate machine code that, given the location of record `r` in memory at run time, finds the location of the field name of this record at run time. If the compiler can compute the type of the record at compile time, then this type information can be used to generate efficient code. More specifically, the type of `r` makes it is possible to compute the location of `name r` relative to the location `r`, at compile time. For example, if the type of `r` is `Student`, then the compiler can build a little table storing the information that `name` occurs before `number` in each `Student` record. Using this table, the compiler can determine that `name` is in the first location allocated to the record `r`. In this case, the expression `name r` is compiled to code that reads the value stored in location `r+1` (if location `r` is used for something else besides the first field). However, for records of a different type, the `name` field might appear second or third. Therefore, if the type of `r` is not known at compile time, the compiler must generate code to compute the location of `name` from the location of `r` at run time. This will make the program run more slowly. To summarize: Some operations can be computed more efficiently if the type of the operand is known at compile time.

In some object-oriented programming languages, the type of an object may be used to find the relative location of parts of the object. In other languages, however, the type system does not give this kind of information and run-time search must be used.

6.2 TYPE SAFETY AND TYPE CHECKING

6.2.1 Type Safety

A programming language is *type safe* if no program is allowed to violate its type distinctions. Sometimes it is not completely clear what the type distinctions are in a specific programming language. However, there are some type distinctions that are meaningful and important in all languages. For example, a function has a different type from an integer. Therefore, any language that allows integers to be used as functions is not type safe. Another action that we always consider a type error is to access memory that is not allocated to the program.

The following table characterizes the type safety of some common programming languages. We will discuss each form of type error listed in the table in turn.

Safety	Example languages	Explanation
Not safe	C and C++	Type casts, pointer arithmetic
Almost safe	Pascal	Explicit deallocation; dangling pointers
Safe	Lisp, Smalltalk, ML, Haskell, Java	Complete type checking

Type Casts. Type casts allow a value of one type to be used as another type. In C in particular, an integer can be cast to a function, allowing a jump to a location that does not contain the correct form of instructions to be a C function.

Pointer Arithmetic. C pointer arithmetic is not type safe. The expression `*(p+i)` has type `A` if `p` is defined to have type `A*`. Because the value stored in location `p+i` might have any type, an assignment like `x = *(p+i)` may store a value of one type into a variable of another type and therefore may cause a type error.

Explicit Deallocation and Dangling Pointers. In Pascal, C, and some other languages, the location reached through a pointer may be deallocated (freed) by the programmer. This creates a *dangling pointer*, a pointer that points to a location that is not allocated to the program. If `p` is a pointer to an integer, for example, then after we deallocate the memory referenced by `p`, the program can allocate new memory to store another type of value. This new memory may be reachable through the old pointer `p`, as the storage allocation algorithm may reuse space that has been freed. The old pointer `p` allows us to treat the new memory as an integer value, as `p` still has type `int`. This violates type safety. Pascal is considered “mostly safe” because this is the only violation of type safety (after the variant record and other original type problems are repaired).

6.2.2 Compile-Time and Run-Time Checking

In many languages, type checking is used to prevent some or all type errors. Some languages use type constraints in the definition of legal program. Implementations of these languages check types at compile time, before a program is started. In these languages, a program that violates a type constraint is not compiled and cannot be run. In other languages, checks for type errors are made while the program is running.

Run-Time Checking. In programming languages with run-time type checking, the compiler generates code so that, when an operation is performed, the code checks to make sure that the operands have the correct type. For example, the Lisp language operation `car` returns the first element of a cons cell. Because it is a type error to apply `car` to something that is not a cons cell, Lisp programs are implemented so that, before `(car x)` is evaluated, a check is made to make sure that `x` is a cons cell. An advantage of run-time type checking is that it catches type errors. A disadvantage is the run-time cost associated with making these checks.

Compile-Time Checking. Many modern programming languages are designed so that it is possible to check expressions for potential type errors. In these languages, it is common to reject programs that do not pass the compile-time type checks. An advantage of compile-time type checking is that it catches errors earlier than run-time checking does: A program developer is warned about the error before the program is given to other users or shipped as a product. In addition, compile-time checking guarantees the absence of type errors regardless of the input to the program. In contrast, dynamic checking will only find type errors in program paths followed during execution. Hence, dynamic checking can make it difficult to find type errors on program paths that are only executed for rarely-occurring inputs. Because compile-time checks may eliminate the need to check for certain errors at run time, compile-time checking can make it possible to produce more efficient code. For a specific example, compiled Haskell code is two to four times faster than Lisp code. The primary reason for this speed increase is that static type checking of Haskell programs greatly reduces the need for run-time tests.

Conservativity of Compile-Time Checking. A property of compile-time type checking is that the compiler must be conservative. This mean that compile-time type checking will find all statements and expressions that produce run-time type errors, but also may flag statements or expressions as errors even if they do not produce run-time errors. More specifically, most checkers are both sound and conservative. A type checker is sound if no programs with errors are considered correct. A type checker is conservative if some programs without errors are still considered to have errors.

There is a reason why most type checkers are conservative: For any Turing-complete programming language, the set of programs that may produce a run-time type error is undecidable. This follows from the undecidability of the halting problem. To see why, consider the following form of program expression:

```
if (complicated-expression-that-could-run-forever)
    then (expression-with-type-error)
    else (expression-with-type-error)
```

It is undecidable whether this expression causes a run-time type error, as the only way for `expression-with-type-error` to be evaluated is for `complicated-expression-that-could-run-forever` to halt. Therefore, deciding whether this expression causes a run-time type error involves deciding whether `complicated-expression-that-could-run-forever` halts.

Because the set of programs that have run-time type errors is undecidable, no compile-time type checker can find type errors exactly. Because the purpose of type checking is to prevent errors, type checkers for type-safe languages are conservative. It is useful that type checkers find type errors, and a consequence of the undecidability of the halting problem is that some programs that could execute without run-time error will fail the compile-time type-checking tests.

The main trade-offs between compile-time and run-time checking are summarized in the following table.

Form of Type Checking	Advantages	Disadvantages
Run-time	Prevents type errors Need not be conservative	Slows program execution
Compile-time	Prevents type errors Eliminates run-time tests Finds type errors *before* execution and run-time tests	May restrict programming because tests are *conservative.*

Combining Compile-Time and Run-Time Checking. Most programming languages actually use some combination of compile-time and run-time type checking. In Java, for example, static type checking is used to distinguish arrays from integers, but array bounds errors (which are a form of type error) are checked at run time.

6.3 TYPE INFERENCE

Type inference is the process of determining the types of expressions based on the known types of some symbols that appear in them. The difference between type inference and compile-time type checking is really a matter of degree. A *type-checking* algorithm goes through the program to check that the types declared by the programmer agree with the language requirements. In *type inference*, the idea is that some information is not specified, and some form of logical inference is required for determining the types of identifiers from the way they are used. For example, identifiers in Haskell are not usually declared to have a specific type. The type system *infers* the types of Haskell identifiers and expressions that contain them from the operations that are used. Type inference was invented by Robin Milner (see the biographical sketch) for the ML programming language. Similar ideas were developed independently by Curry and Hindley in connection with the study of lambda calculus.

Although practical type inference was developed for ML, type inference is applicable to other languages. Haskell, for example, uses the same basic technique. In principle, type inference could also be applied to languages like C. We study type inference in some detail because it illustrates the central issues in type checking and because type inference illustrates some of the central issues in algorithms that find any kind of program errors.

In addition to providing a flexible form of compile-time type checking, type inference supports polymorphism. As we will see when we subsequently look at the type-inference algorithm, the type-inference algorithm uses *type variables* as placeholders for types that are not known. In some cases, the type-inference algorithm resolves all type variables and determines that they must be equal to specific types such as `Int`, `Bool`, or `String`. In other cases, the type of a function may contain type variables that are not constrained by the way the function is defined. In these cases, the function may be applied to any arguments whose types match the form given by a type expression containing type variables.

Although type inference and polymorphism are independent concepts, we discuss polymorphism in the context of type inference because polymorphism arises naturally from the way type variables are used in type inference.

We will use Haskell to illustrate the basic features of the type inference algorithm. Because overloading complicates type inference, we will work with a simplified version of Haskell for the rest of this chapter, called μHaskell. In μHaskell, there is no overloading, so all constants, built-in operators, and other functions have purely monomorphic or polymormphic types. For example, the number `1` in μHaskell has type `Int` rather than the overloaded type `Num a => a` that we saw in Section 5.2.3. Any overloaded type in Haskell has a prefix similar to `Num a =>...`, so it is easy to identify overloaded types by noting the absence of such a prefix.

6.3.1 First Examples of Type Inference

Here are two Haskell type-inference examples to give you some feel for how Haskell type inference works. The behavior of the type-inference algorithm is explained only superficially in these examples, just to give some of the main ideas. We will go through the type inference process in detail in Subsection 6.3.2.

Example 6.1

```
f1 x = x + 2
f1 :: Int → Int
```

The function `f1` adds `2` to its argument. In μHaskell, constant `2` has type Int and the operator `+` has type `Int → Int`. Therefore, the function argument `x` must be an integer. Putting these observations together, we can see that `f1` must have type `Int → Int`.

Example 6.2

```
f2 (g,h) = g(h(0))
f2 :: (a → b, Int → a) → b
```

The type-inference algorithm notices that `h` is applied to an integer argument, and so `h` must be a function from `Int` to something. The algorithm represents “something” by introducing a type variable, which is written as a lower-case letter `a`. The type-inference algorithm then deduces that `g` must be a function that takes whatever `h` returns (something of type `a`) and then returns something else. Because `g` is not constrained to return the same type of value as `h`, the algorithm represents this second something by a new type variable, `b`. Putting the types of `h` and `g` together, we can see that the first argument to `f2` has type `(a → b)` and the second has type `(Int → a)`. Function `f2` takes the pair of these two functions as an argument and returns the same type of value as `g` returns. Therefore, the type of `f2` is `(a → b, Int → a) → b`.

6.3.2 Type-Inference Algorithm

The Haskell type-inference algorithm uses the following three steps to determine the type of an expression:

1. Assign a type to the expression and each subexpression. For any compound expression or variable, use a type variable. For known operations or constants, such as `+` or `3`, use the type that is known for this symbol.
2. Generate a set of constraints on types, using the parse tree of the expression. These constraints reflect the fact that if a function is applied to an argument, for example, then the type of the argument must equal the type of the domain of the function.
3. Solve these constraints by means of unification, which is a substitution-based algorithm for solving systems of equations. (More information on unification appears in the chapter on logic programming.)

The type-inference algorithm is explained by a series of examples. These examples present the following issues:

- explanation of the algorithm
- a polymorphic function definition
- application of a polymorphic function
- a recursive function
- a function with multiple clauses
- type inference indicates a program error

Altogether, these six examples should give you a good understanding of the type-inference algorithm, except for the interaction between type inference and overloading. The interaction between overloading and type inference is not covered in this book.

Example 6.3 Explanation of the Algorithm

We will explain the type-inference algorithm using this example function:

```
add x = 2 + x
add :: Int → Int
```

The easiest way to understand the algorithm is to consider the parse tree of the expression, which is shown in Figure 6.1. The top-level node `Fun` indicates that the parse tree is that of a function declaration. The first two children of the `Fun` node represent variables bound by the `Fun` node: the name of the function, `add`, and its argument, `x`, respectively. The final child is the parse tree of the body of the function. In the body, the operator ‘`+`’ is treated as a curried function rather than an infix operator (Think of it as `(+) 2 x`. In Haskell, putting parentheses around an operator converts the operator to a curried function). The nodes labeled ‘`@`’ denote function applications, in which the left child is applied to the right child. Constant expressions (‘`+`’ and `3`) get their own nodes, as do variables (`x`). Variable nodes are special because they point back to their binding occurrence using a dashed line.

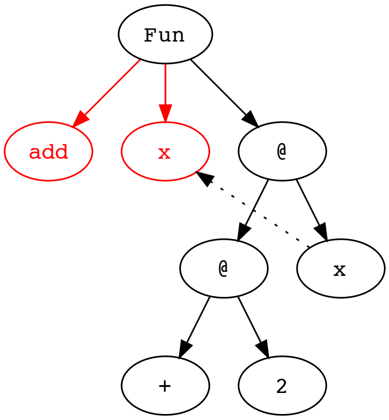

Figure 6.1. Parse tree for add function

Type inference works by applying the following steps to this parse tree.

1. Assign a type variable to the expression and each subexpression.

We illustrate this step by redrawing the graph, writing a type next to each node, as shown in Figure 6.2. Each of these types, written `t_i` for some integer `i`, is a *type variable*, representing the eventual type of the associated expression.

For example, the type `t_0`, the type associated with the `add` identifier, is the type of the function as a whole, type `t_3` is the type of the literal `2`, and type `t_4` is the type of the `(+)` function applied to the literal `2`. Each occurrence of a bound variable must be given the same type because dynamically, all such variables refer to the same value. Consequently, each bound variable is given the same type variable as its binding site. For example, the `x` variable node under the application node is given type `t_1` to match the binding node it references.

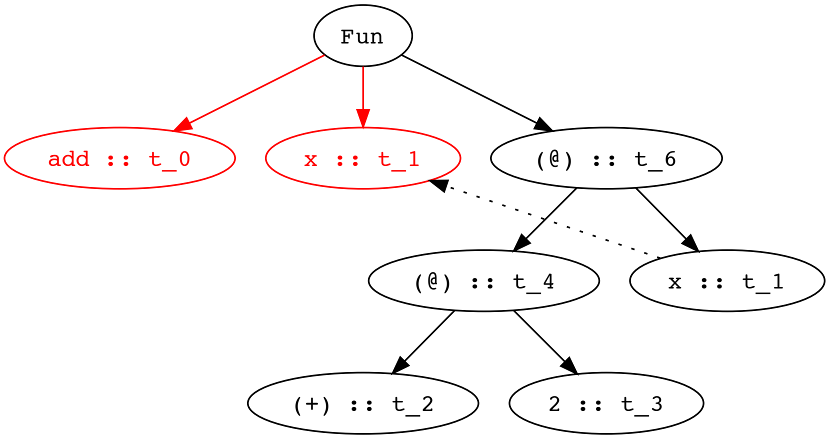

Figure 6.2. Parse tree labeled with type variables

2. *Generate a set of constraints on types, using the parse tree of the expression.*

Constraints are equations between type expressions that must be solved. Figure 6.8 shows the parse tree with the generated constraints. The constraints generated at each node depend upon what kind of node it is.

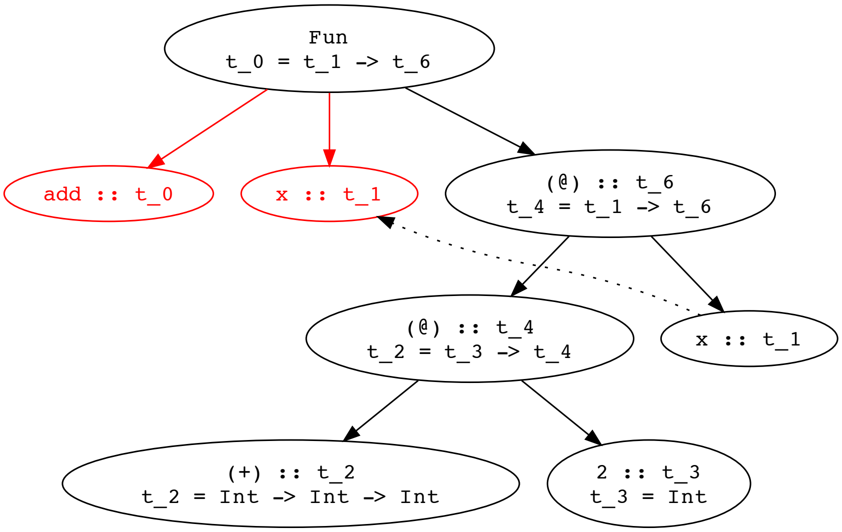

Figure 6.3. Parse tree labeled with type constraints

Constant Expression: Because μHaskell has no overloading, the type of each constant expression is fixed. Hence, we add a constraint equating the type variable of the node with the known type of the constant.

For example, we set t_3 = Int because literal 2 has type Int. Similarly, we set t_2 = Int -> (Int -> Int) because the (+) function has type Int -> (Int -> Int).

Variable: By themselves, variables do not tell us anything about the kind of value they represent, so variable nodes do not introduce any type constraints.

Function Application: (@ nodes). If expression f is applied to expression a, then f must have a function type. In addition, the type of a must be the type of the domain of this function, and the type of f a must be the type of the result of the function. In symbols, if the type of f is t_f, the type of a is t_a, and the type of f a is t_r, then we must have t_f = t_a -> t_r.

In the example, this typing rule is used at the '@' nodes:

Subexpression @ (+) 2: Constraint t_2 = t_3 -> t_4
Subexpression @ (@ (+) 2) x: Constraint t_4 = t_1 -> t_6

In the subexpression @ (+) 2, the type of the left-child (the '(+)' node) is t_2, the type of the right child (the '2' node) is t_3, and the type of the application is t_4. Therefore, we must have that t_2 = t_3 -> t_4. This is the constraint associated with the @ (+) 2 node. The reasoning for subexpression @(@ (+) 2) x is similar: the type of the function @ (+) 2 is t_4, the type of the argument x is t_1, and the type of the application is t_6.

Therefore, we must have `t_4 = t_1 -> t_6`.

Function Definition: The type of a function is a function type from the type of the argument to the type of the body. In symbols, if `f` is a function with argument `x` and body `b`, then if `f` has type `t_f`, `x` has type `t_x`, and `b` has type `t_b`, then these types must satisfy the constraint `t_f = t_x -> t_b`.

For our example expression, there is one function definition, corresponding to the `Fun` node, which gives rise the following constraint:

Subexpression `add x = @ (@ (+) 2) x`: Constraint `t_0 = t_1 -> t_6`

In words, the type of the `add` function is `t_0`, the type of the function argument `x` is `t_1`, and the type of the function body is `t_6`, which gives us the equation `t_0 = t_1 -> t_6`.

3. *Solve the generated constraints using unification.*

Unification is a standard algorithm for solving systems of equations by substitution. The general properties of this algorithm are not discussed here. Instead, the process is shown by example in enough detail that you should be able to figure out the types of simple expressions on your own.

For our example, we have generated the following constraints, which we can read off from the annotated parse tree in Figure 6.8.

(1) `t_0` = `t_1 -> t_6`
(2) `t_4` = `t_1 -> t_6`
(3) `t_2` = `t_3 -> t_4`
(4) `t_2` = `Int -> (Int -> Int)`
(5) `t_3` = `Int`

If there is a way of associating type expression to type variables that makes all of these equations true, then the expression is well typed. If so, the type of the `add` function will be the type expression equal to the type variable `t_0`. If there is no way of associating type expression to type variables that makes all of these equations true, then there is no type for this expression. In this case, the type-inference algorithm will fail, resulting in an error message that says the expression is not well typed.

For Equations (3) and (4) to be true, it must be the case that `t_3 -> t_4` = `Int -> (Int -> Int)`, which implies that

(6) `t_3` = `Int`
(7) `t_4` = `Int -> Int`

We already knew that `t_3 = Int`, but Equation (7) is new. Equations (2) and (7) imply that

(8) `t_1` = `Int`
(9) `t_6` = `Int`

Together, these equations are sufficient to give a satisfying assignment for all the variables in the system of equations.

```
t_0  =  Int -> Int
t_1  =  Int
t_2  =  Int -> Int -> Int
t_3  =  Int
t_4  =  Int -> Int
t_6  =  Int
```

This assignment of type expressions to type variables satisfies the constraints (1) to (4). Hence, the `add` function is well typed, and the type of the function is `Int -> Int`, which is the type expression associated with `t_0`, the type variable assigned to the `add` node.

Example 6.4 A Polymorphic Function Definition

The `apply` function has a type involving type variables, making the function polymorphic.

```
apply (f, x) = f x
apply :: (t → t1, t) → t1
```

In this section, we show how the type inference algorithm infers this polymorphic type. As before, the type-inference algorithm starts with a parse tree for the function, shown in Figure 6.4. The only new kind of node in this parse tree is the node labeled `Pair`, which is the parse tree representation of the pair argument `(f,x)` to the function `apply`. The `f` and `x` children of the `Pair` node are the binding occurrences of those variables. Hence, the other occurrences of `f` and `x` in the body of the function point to these binding occurrences with dashed arrows.

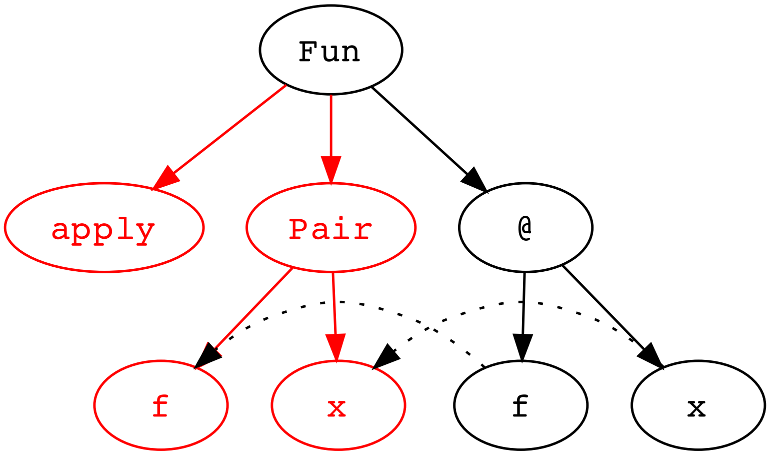

Figure 6.4. Parse tree for apply function

Given a parse tree, the next step of the algorithm is to assign types to each node as shown in Figure 6.5.

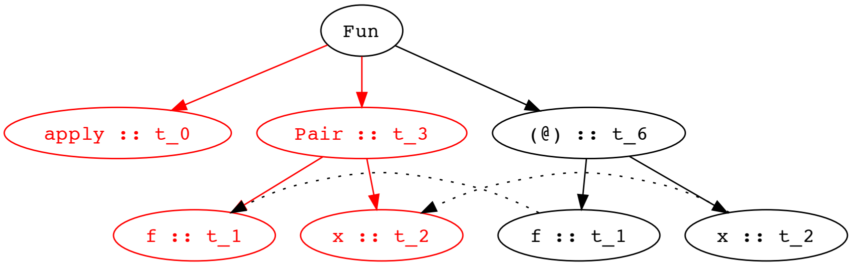

Figure 6.5. Parse tree for apply function labeled with type constraints

The second step of the algorithm is to collect a set of constraints by gathering the constraints generated for each node in the parse tree, following the rules from the previous section.

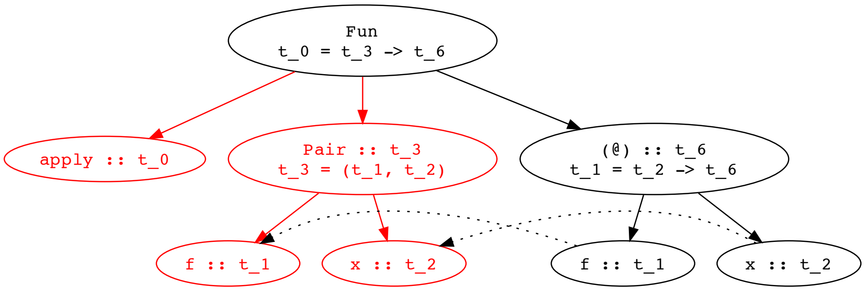

Figure 6.6. Constraints for apply function.

For the application (`@`) node, we get the constraint that `t_1 = t_2 -> t_6` because `t_1`, which is the type of the function, must be a function type from `t_2`, the type of the argument, to `t_6`, the type of the application.

For the abstraction (`Fun`) node, we get the constraint that `t_0 = t_3 -> t_6` because `t_0`, the type of the function, must be a function type from the type of its argument `t_3` to the type of its body `t_6`.

The `Pair` node is a new kind of node and thus we need a rule for how to generate the appropriate constraint.

> *Pair Expression:* The type of a pair expression is a pair of types. The first of these types is the type of the first component of the pair, while the second is the type of the second component. In symbols, if `(a,b)` is a pair and if `a` has type `t_a`, `b` has type `t_b`, and `(a,b)` has type `t_p`, then these types must satisfy the constraint `t_p = (t_a, t_b)`.
>
> For the example, we get the constraint `t_3 = (t_1, t_2)`.

Collecting these constraints, we have three constraints to solve:

(1) `t_1` = `t_2 -> t_6`
(2) `t_0` = `t_3 -> t_6`
(3) `t_3` = `(t_1, t_2)`

The constraints can be solved in order. Using Equation (3), we can substitute for `t_3` in Equation (1), yielding:

(4) `t_0` = `(t_1, t_2) -> t_6`

Next, we can use Equation (1) to substitute for `t_1` in Equation (4):

(5) `t_0` = `(t_2 -> t_6, t_2) -> t_6`

Equation (5) tells us the type of the function. If we rewrite `(t_2 -> t_6, t_2) -> t_6` by replacing `t_2` with `t` and `t_6` with `t1`, we get the compiler output `(t -> t1, t) -> t1` previously shown Because there are type variables in the type of the expression, the function may be applied to arguments with many different types. The following example illustrates this polymorphism by considering an application of the `apply` function.

Example 6.5 Application of a Polymorphic Function

In the last example, we calculated the type of `apply` to be `(t -> t1, t) -> t1`, which is a type that contains type variables. The type variables in this type mean that `apply` is a *polymorphic* function, a function that may be applied to different types of arguments. In the case of `apply`, the type `(t -> t1, t) -> t1` means that `apply` may be applied to a pair of arguments of type `(t -> t1, t)` for any types `t` and `t1`. In particular, recall that function `add x = 2 + x` from Example 6.3 has type `Int -> Int`. Therefore, the pair `(add,3)` has type `(Int -> Int, Int)`, which matches the form `(t -> t1, t)` for function `apply`. In this example, we calculate the type of the application

```
apply(add,3)
```

Following the steps of the type inference algorithm, we begin by assigning types to the nodes in the parse tree for the expression:

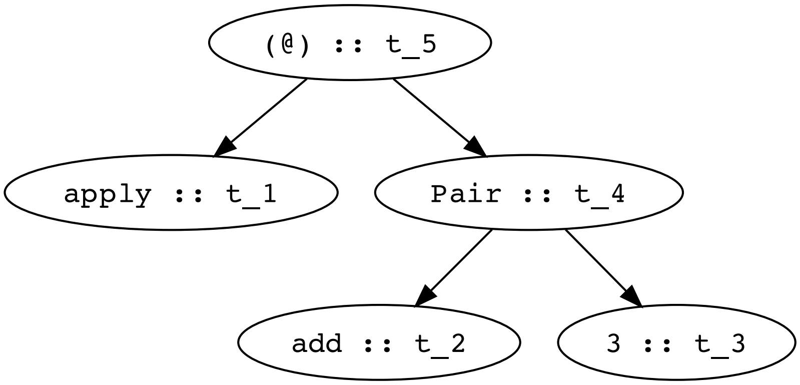

Figure 6.7. Type variable assignment for `apply` function application parse tree.

Next, we annotate the parse tree with constraints over the type variables.

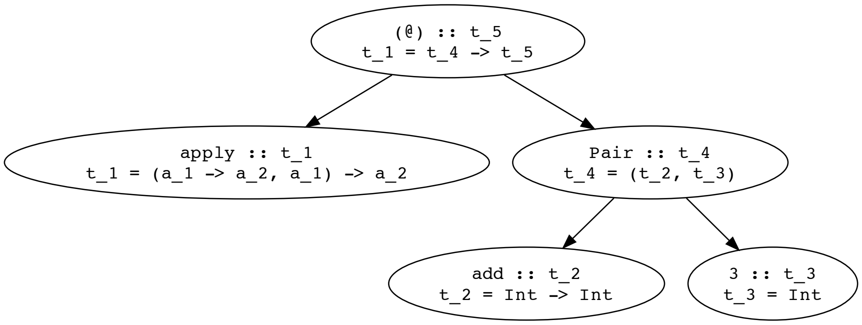

Figure 6.8. Constraints for `apply` function application parse tree.

The constant `3` gives us the constraint that `t_3 = Int`. We know that the type of the `add` function is `Int -> Int`, which produces the constraint `t_2 = Int -> Int`. Similarly, we know that the type of the `apply` function is `(t -> t1, t) -> t1`. Because the type variables `t` and `t1` are variables that can stand for any type, we need to make sure that the variables `t` and `t1` do not already appear elsewhere in the type inference problem we are solving. Therefore, we replace `t` with a *fresh* type variable `a_1` and `t1` with fresh type variable `a_2`, where "fresh" means the variable does not appear already in the set of variables we are working with. After this replacement, we get the constraint `t_1` = `(a_1 -> a_2, a_1) -> a_2`. Finally, following the rules we saw previously for pair and application nodes, we get the additional constraints: `t_4` = `(t_2, t_3)` and `t_1` = `t_4 -> t_5`. Collecting all these constraints together produces the following system of equations:

```
(1)  t_1  =  (a_1 -> a_2, a_1) -> a_2
(2)  t_2  =  Int -> Int
(3)  t_3  =  Int
(4)  t_4  =  (t_2, t_3)
(5)  t_1  =  t_4 -> t_5
```

Now we must solve the constraints. Combining the first and fifth equations:

```
(6)  (a_1 -> a_2, a_1) -> a_2  =  t_4 -> t_5
```

This constraint has an expression on each side of the equal sign. To solve this constraint, corresponding parts of each expression must be equal. In other words, this constraint implies the following two constraints:

```
(7)  (a_1 -> a_2, a_1)  =  t_4
(8)  a_2                =  t_5
```

Equations (4) and (7) yield the following two constraints:

```
(9)   a_1 -> a_2  =  t_2
(10)  a_1         =  t_3
```

and combining Equations (2) and (9) produces:

```
(11)  a_1  =  Int
(12)  a_2  =  Int
```

Thus the substitution

```
t_1  =  (Int -> Int, Int) -> Int
t_2  =  Int -> Int
t_3  =  Int
t_4  =  (Int -> Int, Int)
t_5  =  Int
a_1  =  Int
a_2  =  Int
```

solves all the constraints.

Because all of the constraints are solved, the expression `apply(add,3)` is typeable in the Haskell type system. The type of `apply(add,3)` is the solution for type variable `t_5`, namely `Int`.

We can also apply the function `apply` to other types of arguments. For example, if the function `not` has type `Bool -> Bool`, then

```
apply(not, False)
```

is a well-typed expression with type `Bool`, which can be calculated by exactly the same type-inference process as for `apply(add,3)`. This fact illustrates the polymorphism of `apply`: Because the type `(t -> t1, t) -> t1` of `apply` contains type variables, the function may be applied to any type of arguments that can be obtained if the type variables in `(t -> t1, t) -> t1` are replaced with type names or type expressions.

Example 6.6 A Recursive Function

When a function is defined recursively, we must determine the type of the function body without knowing the type of recursive function calls. To see how this works, consider this simple recursive function that sums the integers up to a given integer. This function does not terminate, but it does type check:

```
sum x = x + sum (x-1)
sum :: Int -> Int
```

Figure 6.9 shows the parse tree of the `sum` function annotated with type variables and associated constraints. The recursive call to the `sum` function appears in the parse tree as a variable node. Just like any other variable node, this node has a dotted arrow pointing to the node where the variable is bound. Again just like any other variable node, the type given to the variable node is the same as the type given to the binding node; in this case, type `t_0`. We follow the same procedure as in the previous examples to generate constraints from the parse tree and to solve those constraints, producing the following solution:

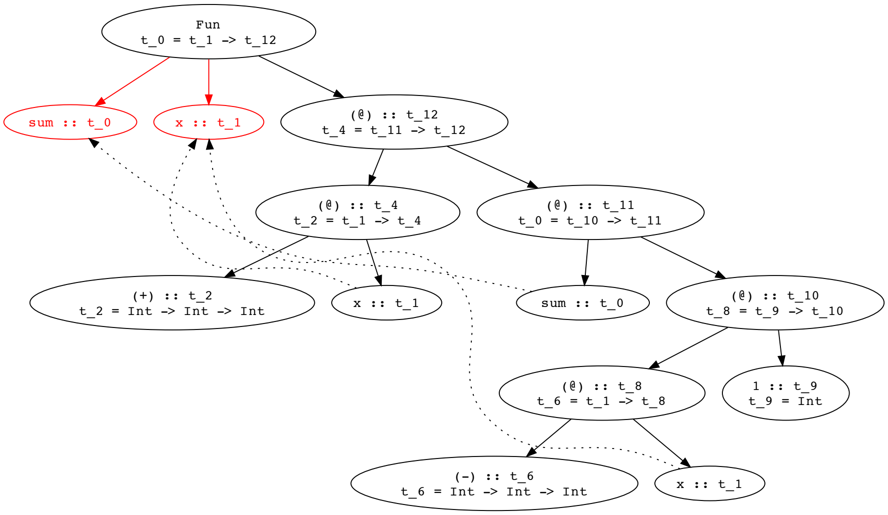

Figure 6.9. Parse tree for sum function annotated with type variables and associated constraints.

```
t_0   =  Int -> Int
t_1   =  Int
t_2   =  Int -> (Int -> Int)
t_4   =  Int -> Int
t_6   =  Int -> (Int -> Int)
t_8   =  Int -> Int
t_9   =  Int
t_10  =  Int
t_11  =  Int
t_12  =  Int
```

Because the constraints can be solved, the function is typeable. In the process of solving the constraints, we have calculated that the type of sum (the type t_0) is Int -> Int.

Example 6.7 A Function with Multiple Clauses

Type inference for functions with several clauses may be done by a type check of each clause separately. Then, because all clauses define the same function, we impose the constraint that the types of all clauses must be equal. For example, consider the append function on lists, defined as follows:

```
append ([], r) = r
append (x:xs, r) = x : append(xs, r)
append :: ([t], [t]) -> [t]
```

As the type ([t], [t]) -> [t] indicates, append can be applied to any pair of lists, as long as both lists contain the same type of list elements. Thus, append is a polymorphic function on lists.

We begin type inference for `append` by following the three-step algorithm for the first clause of the definition, then repeating the steps for the second clause. This gives us two types:

```
append :: ([t], t_1) -> t_1
append :: ([t], t_1) -> [t]
```

Intuitively, the first clause has type `([t], t_1) -> t_1` because the first argument must match the empty list `[]`, but the second argument may be anything. The second clause has type `([t], t_1) -> [t]` because the return result is a list containing one element from the list passed as the first argument.

If we require that the two clauses have the same type by imposing the constraint

```
([t], t_1) -> t_1 = ([t], t_1) -> [t]
```

then we must have `t_1 = [t]`. This equality gives us the final type for append:

```
append :: ([t], [t]) -> [t]
```

Example 6.8 Type Inference Indicates a Program Error

Here is an example that shows how type inference may produce output that indicates a programming error, even though the program may type correctly. Here is a sample (incorrect) declaration of a `reverse` function on lists and its type:

```
reverse [] = []
reverse (x:xs) = reverse xs
reverse :: [t] -> [t_1]
```

As the typing shows, this function is typeable; there is no type error in this declaration. However, look carefully at the type of `reverse`. The type `[t] -> [t_1]` means that we can apply `reverse` to any type of list and obtain any type of list as a result. However, the type of the "reversed" list is not the same as the type of the list we started with!

Because it does not make sense for `reverse` to return a list that is a different type from its argument, there must be something wrong with this code. The problem is that, in the second clause, the first element `x` of the input list is not used as part of the output. Therefore, `reverse` always returns the empty list.

As this example illustrates, the type-inference algorithm may sometimes return a type that is more general than the one we expect. This does not indicate a type error. In this example, the faulty reverse can be used anywhere that a correct reverse function could be used. However, the type of reverse is useful because it tells the programmer that there is an error in the program.

6.4 POLYMORPHISM

Polymorphism, which literally means "having multiple forms," refers to constructs that can take on different types as needed. For example, a function that can compute the length of any type of list is polymorphic because it has type `[t] -> Int` for every type `t`.

There are three forms of polymorphism in contemporary programming languages:

- *parametric polymorphism*, in which a function may be applied to any arguments whose types match a type expression involving type variables;
- *ad hoc polymorphism*, another term for overloading, in which two or more implementations with different types are referred to by the same name;
- *subtype polymorphism*, in which the subtype relation between types allows an expression to have many possible types.

We discuss parametric polymorphism in this section and ad hoc polymorphism (overloading) in the next chapter. We consider subtype polymorphism in later chapters in connection with object-oriented programming.

6.4.1 Parametric Polymorphism

The main characteristic of parametric polymorphism is that the set of types associated with a function or other value is given by a type expression that contains type variables. For example, a Haskell function that sorts lists might have the Haskell type

```
sort :: ((t, t) -> Bool, [t]) -> [t]
```

In words, `sort` can be applied to any pair consisting of a function and a list, as long as the function has a type of the form `(t,t) -> Bool`, in which the type `t` must also be the type of the elements of the list. The function argument is a `less-than` operation used to determine the order of elements in the sorted list.

In parametric polymorphism, a function may have infinitely many types, as there are infinitely many ways of replacing type variables with actual types. The `sort` function, for example, may be used to sort lists of integers, lists of lists of integers, lists of lists of lists of integers, and so on.

Parametric polymorphism may be implicit or explicit. In *explicit parametric polymorphism*, the program text contains type variables that determine the way that a function or other value may be treated polymorphically. In addition, explicit polymorphism often involves explicit instantiation or type application to indicate how type variables are replaced with specific types in the use of a polymorphic value. C++ templates are a well-known example of explicit polymorphism. Haskell polymorphism is called *implicit parametric polymorphism* because programs that declare and use polymorphic functions do not need to contain types – the type-inference algorithm computes when a function is polymorphic and computes the instantiation of type variables as needed.

C++ Function Templates

For many readers, the most familiar type parameterization mechanism is the C++ template mechanism. Although some C++ programmers associate templates with classes and object-oriented programming, function templates are also useful for programs that do not declare any classes.

As an illustrative example, suppose we write a simple function to swap the values of two integer variables:

```
void swap(int& x, int& y){
   int tmp = x; x = y; y = tmp;
}
```

Although this code is useful for exchanging values of integer variables, the sequence of instructions also works for other types of variables. If we wish to swap values of variables of other types, then we can define a function template that uses a type variable `T` in place of the type `int`:

```
template <typename T>
void swap(T& x, T& y){
   T tmp = x; x = y; y = tmp;
}
```

For those who are not familiar with templates, the main idea is to think of the type name `T` as a parameter to a function from types to functions. When applied to, or *instantiated* to, a specific type, the result is a version of `swap` that has `int` replaced with another type. In other words, `swap` is a general function that would work perfectly well for many types of arguments. Templates allow us to treat `swap` as a function with a type argument.

In C++, function templates are instantiated automatically as needed, with the types of the function arguments used to determine which instantiation is needed. This is illustrated in the following example lines of code.

```
int i,j;      ... swap(i,j);  // replace T with int
float a,b;    ... swap(a,b);  // replace T with float
String s,t;   ... swap(s,t);  // replace T with String
```

Comparison with Haskell Polymorphism

In Haskell polymorphism, the type-inference algorithm infers the type of a function and the type of a function application (as explained in Section 6.3). When a function is polymorphic, the actions of the type-inference algorithm can be understood as automatically inserting "template declarations" and "template instantiation" into the program. We can see how this works by considering a Haskell sorting function that is analogous to the C++ sort function previously declared:

```
insert (less, x, [])   = [x]
insert (less, x, y:ys) = if less(x,y) then x:y:ys
                         else y:insert(less,x,ys)
sort(less, [])   = []
sort(less, x:xs) = insert(less, x, sort(less,xs))
```

For sort to be polymorphic, a less-than operation must be passed as a function argument to sort.

The types of insert and sort, as inferred by the type-inference algorithm, are

```
insert :: ((t, t) -> Bool, t, [t]) -> [t]
sort   :: ((t, t) -> Bool, [t]) -> [t]
```

In these types, the type variable t can be instantiated to any type, as needed. In effect, the functions are treated as if they were "templates." If we were able to combine syntax for C++ templates, Haskell functions, and Haskell types, the functions previously defined could also be written as

```
template <type t>
insert(less :: (t, t) -> Bool, x :: t, [] :: [t]) = [x]
insert(less, x, y:ys) = if less(x,y) then x:y:ys
                        else y:insert(less,x,ys)
template <type t>
sort(less :: (t, t) -> Bool, [] :: [t]) = []
sort(less, x:xs) = insert(less, x, sort(less,xs))
```

These declarations are the explicitly typed versions of the implicitly polymorphic Haskell functions. In other words, the Haskell type-inference algorithm may be understood as a program preprocessor that converts Haskell expressions without type information into expressions in some explicitly typed intermediate language with templates. From this point of view, the difference between explicit and implicit polymorphism is that a programming language processor (such as the Haskell compiler) takes the simpler implicit syntax and automatically inserts explicit type information, converting from implicit to explicit form, before programs are compiled and executed.

Finishing this example, suppose we declare a less-than function on integers:

```
less :: (Int, Int) -> Bool
less(x,y) = x < y
```

In the following application of the polymorphic sort function, the sort template is automatically instantiated to type Int, so sort can be applied to an integer list:

```
sort (less, [1,4,5,3,2])
[1,2,3,4,5]
it :: [Int]
```

6.4.2 Implementation of Parametric Polymorphism

C++ templates and Haskell polymorphic functions are implemented differently. The reason for the difference is not related to the difference between explicitly polymorphic syntax and implicitly polymorphic syntax. The need for different implementation techniques arises from the difference between data representation in C and data representation in Haskell.

C++ Implementation

C++ templates are instantiated at link time. More specifically, suppose the swap function template is stored in one file and a program calling swap is stored in another file and these files are compiled separately. The so-called relocatable object files produced by compilation of the calling program will include information indicating that the compiled code calls a function swap of a certain type. The program linker combines the two program parts by linking the calls to swap in the one program part to the definition of swap in the other. It does so by instantiating the compiled code for swap in a form that produces code appropriate for the calls to swap.

If a program calls swap with several different types, then several different instantiated copies of swap will be produced. One reason that a different copy is needed for each type of call is that function swap declares a local variable tmp of type T. Space for tmp must be allocated in the activation record for swap. Therefore the compiled code for swap must be modified according to the size of a variable of type T. If T is a structure or object, for example, then the size might be fairly large. On the other hand, if T is int, the size will be small. In either case, the compiled code for swap must "know" the size of the datum so that addressing into the activation record can be done properly.

The linking process for C++ is relatively complex. We will not study it in detail. However, it is worth noting that if < is an overloaded operator, then the correct version of < must be identified when the compiled code for sort is linked with a calling program. For example, consider the following generic sort function:

```
template <typename T>
void sort( int count, T * A[count] ) {
   for (int i=0; i <count-1; i++)
         for (int j=i+1; j<count-1; j++)
           if (A[j] < A[i]) swap(A[i],A[j]);
}
```

If A is an array of type T, then sort(n, A) will work only if operator < is defined on type T. This requirement of sort is not declared anywhere in the C++ code. However, when the function template is instantiated, the actual type T must have an operator < defined or a link-time error will be reported and no executable object code will be produced.

Haskell Implementation

In Haskell, there is one sequence of compiled instructions for each polymorphic function. There is no need to produce different copies of the code for different types of arguments because related types of data are represented in similar ways. More specifically, pointers are used in parameter passing and in the representation of data structures such as lists so that when a function is polymorphic, it can access all necessary data in the same way, regardless of its type. This property of Haskell is called *uniform data representation.*

A simple example of uniform data representation is the polymorphic Haskell pair function:

```
pair :: t -> t1 -> (t, t1)
pair x y = (x,y)
```

As the type indicates, this pair function can be applied to any two values. Haskell represents all values as pointers; therefore, when two values are passed to the pair function, the pair function receives two pointers. The two pointers are the same size (typically 32 bits), regardless of what type of value is being passed. In fact, the compiler can implement the entire computation by only manipulating the pointers, not the values they point to. As a result, none of the compiled code for pair depends on the size of the data referred to by arguments x and y.

Uniform data representation has its advantages and disadvantages. Because there is no need to duplicate code for different argument types, uniform data representation leads to smaller code size and avoids complications associated with C++-style linking. On the other hand, the resulting code can be less efficient, as uniform data representation often involves using pointers to data instead of storing data directly in structures.

For polymorphic list functions to work properly, all lists must be represented in exactly the same way. Because of this uniformity requirement, small values that would fit directly into the car part of a list cons cell cannot be placed there because large values do not fit. Hence we must store pointers to small values in lists, just as we store pointers to large values. Haskell programmers and compiler writers call the process of making all data look the same by means of pointers *boxing.*

Comparison

Two important points of comparison are efficiency and reporting of error messages. As far as efficiency, the C++ implementation requires more effort at link time and produces a larger code size, as instantiating a template several times will result in several copies of the code. The Haskell implementation will run more slowly unless special optimizations are applied; uniform data representation involves more extensive use of pointers and these pointers must be stored and followed.

As a general programming principle, it is more convenient to have program errors reported at compile time than at link time. One reason is that separate program modules are compiled independently, but are linked together only when the entire system is assembled. Therefore, compilation is a "local" process that can be carried out by the designer or implementer of a single component. In contrast, link-time errors represent global system properties that are not known until the entire system is assembled. For this reason, C++ link-time errors associated with operations in templates can be irritating and a source of frustration.

Somewhat better error reporting for C++ templates could be achieved if the template syntax included a description of the operations needed on type parameters. However, this is relatively complicated in C++, because of overloading and other properties of the language. In contrast, Haskell has a more principled overloading mechanism and includes more information in parameterized constructs, allowing all type errors to be reported as a program unit is compiled. We will discuss overloading in more detail in Chapter 7.

6.5 TYPE DECLARATIONS AND TYPE EQUALITY

Many kinds of type declarations and many kinds of type equality have appeared in programming languages over the years. Type declarations and type equality are related because when a type name is declared, it is important to decide whether this is a "new" type that is different from all other types or a new name whose meaning is equal to some other type that may be used elsewhere in the program.

There are two basic forms of type declaration:

- *transparent*, meaning an alternative name is given to a type that can also be expressed without this name,
- *opaque*, meaning a new type is introduced into the program that is not equal to any other type.

Two historical names for these two forms of type equality are *structural type equality* and *name type equality.* Intuitively, structural equality means that two type names are the same if the types they name are the same (i.e., have the same structure). Name equality means that two type names are considered equal in type checking only if they are the same name.

Although these terms may seem simple and innocuous, there are lots of confusing phenomena associated with the use of structural and name type equivalence in programming languages. Instead of discussing many of the possible forms, we simply look at a few rational possibilities.

6.5.1 Transparent Type Declarations

In the Haskell form of transparent type declaration,

```
type <type_identifier> = <type_expression>
```

the identifier becomes a synonym for the type expression. For example, the code

```
type Celsius = Float
type Fahrenheit = Float
```

declares two type names, `Celsius` and `Fahrenheit`, whose meaning is the type `Float`, just the way that the two value declarations

```
x = 3
y = 3
```

declare two identifiers whose value is `3`. (Remember that Haskell identifiers are not assignable variables; the identifier `x` will have value `3` wherever it is used.) If we declare a Haskell function to convert from Fahrenheit to Celsius, this function will have type `Float -> Float` (if we ignore operator overloading):

```
toCelsius x = ((x-32.0)* 0.555556)
```

This fact should not be surprising because there is no indication that the function argument or return value has any type other than `Float`. However, because types `Celsius` and `Fahrenheit` are both equal to type `Float`, the function `toCelsius` also has type `Fahrenheit -> Celsius`. The programmer can indicate this fact by specifying the type of the argument and result:

```
toCelsius :: Fahrenheit -> Celsius
toCelsius x = ((x-32.0)* 0.555556)
```

This version of the `toCelsius` function is more informative to read, as the types indicate the intended purpose of the function. However, because `Fahrenheit` and `Celsius` are synonyms for `Float`, this function can be applied to any `Float` argument:

```
toCelsius 74.5
23.61113
it :: Celsius
```

The Haskell type checker gives the result type `Celsius`, but because `Celsius` = `Float`, the result can be used in `Float` expressions.

In addition to transparent type declarations, Haskell also supports opaque type declarations. We will discuss one such example in Subsection 6.5.3.

6.5.2 C Declarations and Structs

The basic type declaration construct in C is typedef. Here are some simple examples:

```
typedef char byte;
typedef byte ten_bytes[10];
```

the first declaring a type byte that is equal to char and the second an array type ten_bytes that is equal to arrays of 10 bytes. Generally speaking, the C typedef construct works similarly to the transparent Haskell type declaration discussed in the preceding subsection. However, when structs are involved, the C type checker considers separately declared type names to be unequal, even if they are declared to name the same struct type. Here is a short program example illustrating this behavior:

```
typedef struct {int m;} A;
typedef struct {int m;} B;
A x;
B y;
x=y;  /* incompatible types in assignment */
```

Here, although the two struct types used in the two declarations are the same, the C type checker does not treat A and B as equal types. However, if we replace the two declarations with typedef int A; typedef int B;, using int in place of structs, then the assignment is considered type correct.

6.5.3 Haskell Data-Type Declarations

The Haskell data-type declaration, discussed in Section 5.4, is a form of type declaration that simultaneously defines a new type name and operations for building and making use of elements of the type. Because all of the examples in Section 5.4 were monomorphic, we take a quick look at a polymorphic declaration before discussing type equality.

Here is an example of a polymorphic data type of trees. You may wish to compare this with the monomorphic (nonpolymorphic) example in Subsection 5.4:

```
data Tree a = Leaf a | Node (Tree a, Tree a)
```

This declaration defines a polymorphic type Tree a, with instances Tree Int, Tree String, and so on, together with polymorphic constructors Leaf and Node:

```
Leaf :: a -> Tree a
```

```
Node :: (Tree a , Tree a) -> Tree a
```

The following function checks to see if an element appears in a tree.

```
inTree :: (Eq t) => (t, Tree t) -> Bool
inTree (x, Leaf y) = x == y
inTree (x, Node(left,right)) =
               inTree (x, left) || inTree(x, right)
```

The type indicates that `inTree` will work for any type `t` that belongs to the `Eq` type class, meaning the type `t` must have an equality operator defined for it. This qualification on the type of `t` enables us to compare `x` and `y` for equality in the first clause of the function. We will study type classes in more detail in the next chapter.

Each Haskell data-type declaration is considered to define a new type different from all other types. Even if two data types have the same structure, they are not considered equivalent.

The design of Haskell makes it hard to declare similar data types, as each constructor has only one type. For example, the two declarations

```
data A = C Int
data B = C Int
```

declare distinct types `A` and `B`. Both declarations introduce a constructor function named `C`, the first with type `Int -> A` and the second with type `Int -> B`. Because both constructors are in the same scope, the Haskell compiler reports an error, saying that the constructor `C` has been defined more than once. We can solve this problem by putting the two declarations into two different modules, `M1` and `M2`. We can then see that `M1.A` and `M2.B` are considered different by writing a function that attempts to treat a value of one type as the other,

```
f :: M1.A -> M2.B
f x = x
```

which leads to the message: `Couldn’t match expected type ‘M1.B’ against inferred type ‘M2.A’`.

6.6 CHAPTER SUMMARY

In this chapter, we studied reasons for using types in programming languages, methods for type checking, and some typing issues such as polymorphism and type equality.

Reasons for Using Types

There are three main uses of types in programming languages:

- *Naming and organizing concepts:* Functions and data structures can be given types that reflect the way these computational constructs are used in a program. This helps the programmers and anyone else reading a program figure out how the program works and why it is written a certain way.
- *Making sure that bit sequences in computer memory are interpreted consistently:* Type checking keeps operations from being applied to operands in incorrect ways. This prevents a floating-point operation from being applied to a sequence of bits that represents a string, for example.
- *Providing information to the compiler about data manipulated by the program:* In languages in which the compiler can determine the type of a data structure, for example, the type information can be used to determine the relative location of a part of this structure. This compile-time type information can be used to generate efficient code for indexing into the data structure at run time.

Type Inference

Type inference is the process of determining the types of expressions based on the known types of some of the symbols that appear in them. For example, we saw how to infer that the function `g` declared by

```
g x = 2 + x
```

has type `Int -> Int`. The difference between type inference and compile-time type checking is a matter of degree. A type-checking algorithm goes through the program to check that the types declared by the programmer agree with the language requirements. In type inference, the idea is that some information is not specified and some form of logical inference is required for determining the types of identifiers from the way they are used.

The following steps are used to infer the type of an expression `e`:

1. Assign a type variable to `e` and each subexpression. Each such variable represents the unknown type of the corresponding expression.
2. Generate a set of constraints on these type variables from the form of the parse tree of the expression.
3. Solve these constraints by using unification, which is a substitution-based algorithm for solving systems of equations.

In a series of examples, we saw how to apply this algorithm to a variety of expressions. Type inference has many characteristics in common with the kind of algorithms that are used in compilers and programming environments to determine properties of programs. For example, some useful alias analysis algorithms that try to determine whether two pointers might point to the same location have the same general outline as that of type inference.

Parametric Polymorphism

There are three forms of polymorphism, which literally means “many shapes”: parametric polymorphism, ad hoc polymorphism (another term for overloading), and subtype polymorphism. We studied the first of these in this chapter. We will examine ad hoc polymorphism in the next chapter and subtype polymorphism in later chapters on object-oriented languages. Parametric polymorphism can be either implicit, as in Haskell, or explicit, as with C++ templates. There are two ways of implementing parametric polymorphism, one in which the same data representation is used for all types of data and one in which different data representations are are used for different types of data. In this second approach, the parametric code is instantiated (ie, slightly different versions of the code are used) to manage different data representations.

Type Declarations and Type Equality

We discussed opaque and transparent type declarations. In opaque type declarations, the type name stands for a distinct type different from all other types. In transparent type declarations, the declared name is a synonym for another type. Both forms are used in many programming languages.

EXERCISES

6.1 Haskell Types

Assuming that integer literals have type `Int` and binary arithmetic operators `+` and `*` have type `Int -> Int -> Int`, explain the uHaskell type for each of the following declarations:

```
(a)  a (x,y) = x + 2 * y
(b)  b (x,y,f) = if f y then x else y
(c)  c f = \y -> f y
(d)  d (f,x) = f(f x)
```

Because you can simply type these expressions into a Haskell interpreter to determine the type, be sure to write a short explanation to show that you understand why each function has the type you give.

6.2 Polymorphic Sorting

This function performing insertion sort on a list takes as arguments a comparison function `less` and a list `l` of elements to be sorted. The code compiles and runs correctly:

```
sort (less, []) = []
sort (less, a : l) =
   let insert(a, []) = a : []
       insert(a, b:l) = if less(a,b) then a : b : l
                        else b : insert (a,l)
   in insert(a, sort(less, l))
```

What is the type of this `sort` function? Explain briefly, including the type of the subsidiary function `insert`. You do not have to run the Haskell algorithm on this code; just explain why an ordinary Haskell programmer would expect the code to have this type.

6.3 Types and Garbage Collection

Language D allows a form of “cast” in which an expression of one type can be treated as an expression of any other. For example, if *x* is a variable of type integer, then *(string)x* is an expression of type string. No conversion is done. Explain how this might affect garbage collection for language D.

For simplicity, assume that D is a conventional imperative language with integers, reals (floating-point numbers), pairs, and pointers. You do not need to consider other language features.

6.4 Polymorphic Fixed Point

A *fixed point* of a function f is some value x such that $x \mathcal{D} f(x)$. There is a connection between recursion and fixed points that is illustrated by this Haskell definition of the factorial function `factorial :: Integer -> Integer`:

```
y f x = f (y f) x
g f x = if x == 0 then 1 else x*f(x-1)
factorial = y g
```

The first function, `y`, is a fixed-point operator. The second function, `g`, is a function on functions whose fixed point is `factorial`. Both of these are curried functions; using the Haskell syntax `\x -> . . .` for λx . . . , we could also write the function `g` as

```
g f = \x ->
      if x == 0 then 1 else x*f(x-1)
```

This `g` is a function that, when applied to argument `f`, returns a function that, when applied to argument `x`, has the value given by the expression `if x == 0 then 1 else x*f(x-1)`.

(a) What type will the Haskell compiler deduce for `g`?

(b) What type will the Haskell compiler deduce for `y`?

Explain your answers in a few sentences.

6.5 Type Inference 1

Use the parse graph in Figure 6.11 to calculate the uHaskell type for the function

```
f(g,h) = g(h) + 2
```

Assume that `2` has type `Integer` and `+` has type `Integer -> Integer -> Integer`.

6.6 Type Inference 2

Use the following parse graph to follow the steps of the Haskell type-inference algorithm on the function declaration

```
f g = (g g) + 2
```

Assume that `2` has type `Integer` and `+` has type `Integer -> Integer -> Integer`. What is the output of the type checker?

6.7 Type Inference and Bugs

What is the type of the following Haskell function?

```
append([], l) = l
append(x:l, m) = append(l, m)
```

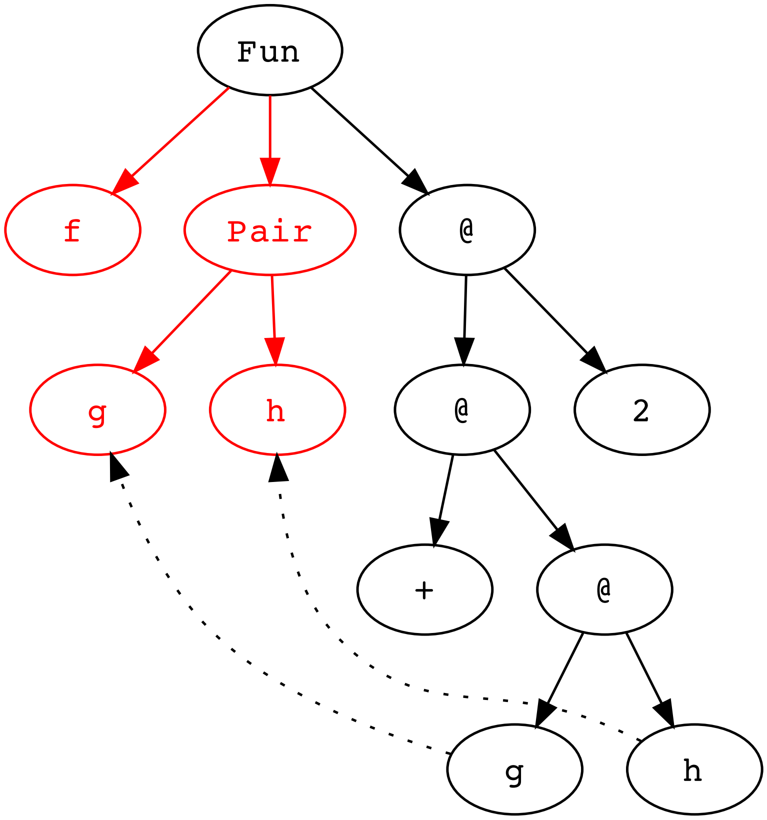

Figure 6.10. Parse tree for problem 6.5.

Write one or two sentences to explain succinctly and informally why `append` has the type you give. This function is intended to append one list onto another. However, it has a bug. How might knowing the type of this function help the programmer to find the bug?

6.8 Type Inference and Debugging

The `reduce` function takes a binary operation, in the form of a function `f`, and a list, and produces the result of combining all elements in the list by using the binary operation. For example;

```
reduce plus [1,2,3] = 1 + 2 + 3 = 6
```

if `plus` is defined by

```
plus(x,y::Int) = x + y
```

A friend of yours is trying to learn Haskell and tries to write a `reduce` function. Here is his incorrect definition:

```
reduce(f, x) = x
reduce(f, (x : y)) = f(x, reduce(f, y))
```

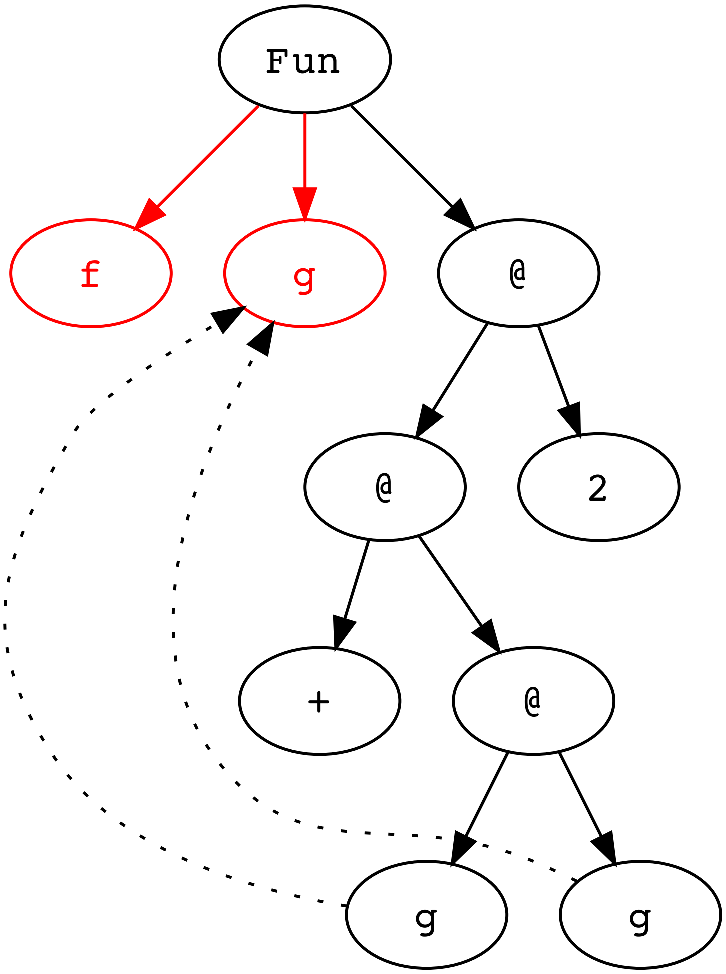

Figure 6.11. Parse tree for problem 6.6.

He tells you that he does not know what to return for an empty list, but this should work for a nonempty list: If the list has one element, then the first clause returns it. If the list has more than one element, then the second clause of the definition uses the function `f`. This sounds like a reasonable explanation, but the type checker gives you the following output:

```
reduce :: ((t, [t]) -> [t], [t]) -> [t]
```

How can you use this type to explain to your friend that his code is wrong?

6.9 Polymorphism in C

In the following C `min` function, the type `void` is used in the types of two arguments.

However, the function makes sense and can be applied to a list of arguments in which `void` has been replaced with another type. In other words, although the C type of this function is not polymorphic, the function could be given a polymorphic type if C had a polymorphic type system. Using Haskell notation for types, write a type for this `min` function that captures the way that `min` could be meaningfully applied to arguments of various types. Explain why you believe the function has the type you have written.

```
int min (
   void *a[],       /* a is an array of pointers to data of
unknown type */
   int n,          /*  n is the length of the array */
   int (*less)(void*, void*)    /* parameter less is a pointer to
function */
  )                          /* that is used to compare array
elements */
{
   int i;
   int m;
   m=0;
   for (i=1; i < n; i++)
      if (less(a[i], a[m])) m=i;
   return(m);
}
```

6.10 Typing and Run-Time Behavior

The following Haskell functions have essentially identical computational behavior,

```
f x = not (f x)
g y = (g y) * 2
```

because except for typing differences, we could replace one function with the other in any program without changing the observable behavior of the program. In more detail, suppose we turn off the Haskell type checker and compile a program of the form $\mathcal{P}$`[f x = not (f x)]`. Whatever this program does, the program $\mathcal{P}$`[g y = (g y) * 2]` we obtain by replacing one function definition with the other will do exactly the same thing. In particular, if the first does not lead to a run-time type error such as adding an integer to a string, neither will the second.

(a) What is the Haskell type for `f`?

(b) What is the Haskell type for `g`? (Assume that `2` has type `Integer` and `+` has type `Integer -> Integer -> Integer`).

(c) Give an informal explanation of why these two functions have the same run-time behavior.

(d) Because the two functions are equivalent, it might be better to give them the same type. Why do you think the designers of the Haskell typing algorithm did not work harder to make it do this? Do you think they made a mistake?

6.11 Dynamic Typing in Haskell

Many programmers believe that a run-time typed programming language like Lisp or Scheme is more expressive than a compile-time typed language like Haskell, as there is no type system to “get in your way.” Although there are some situations in which the flexibility of Lisp or Scheme is a tremendous advantage, we can also make the opposite argument. Specifically, Haskell is more expressive than Lisp or Scheme because we can define an Haskell data type for Lisp or Scheme expressions.

Here is a type declaration for pure historical Lisp:

```
data LISP = Nil
          | Symbol String
          | Number Int
          | Cons (LISP, LISP)
          | Function (LISP -> LISP)
```

Although we could have used `(Symbol ‘‘nil’’)` instead of a primitive `Nil`, it seems convenient to treat `Nil` separately.

(a) Write a Haskell declaration for the Lisp function `atom` that tests whether its argument is an atom. (Everything except a cons cell is an `atom` – The word `atom` comes from the Greek word *atomos*, meaning indivisible. In Lisp, symbols, numbers, nil, and functions cannot be divided into smaller pieces, so they are considered to be atoms.) Your function should have type `LISP -> LISP`, returning atoms `Symbol(‘‘T’’)` or `Nil`.

(b) Write a Haskell declaration for the Lisp function `islist` that tests whether its argument is a *proper* list. A proper list is either `Nil` or a cons cell whose `cdr` is a proper list. Note that not all list-like structures built from cons cells are proper lists. For instance, `(Cons (Symbol(‘‘A’’), Symbol(‘‘B’’)))` is not a proper list (it is instead what is known as a dotted list), and so `(islist (Cons (Symbol(‘‘A’’), Symbol(‘‘B’’))))` should evaluate to `Nil`. On the other hand, `(Cons (Symbol(‘‘A’’), (Cons (Symbol(‘‘B’’), Nil))))` is a proper list, and so your function should evaluate to `Symbol(‘‘T’’)`. Your function should have type `LISP -> LISP`, as before.

(c) Write a Haskell declaration for the Lisp `car` function and explain briefly. The function should have type `LISP -> LISP`.

(d) Write the Lisp expression `(lambda (x) (cons x ’A))` as a Haskell expression of type `LISP -> LISP`. Explain briefly.