6

Type Systems and Type Inference

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, overloading, 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 int channel 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 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. Because the bits used to represent the floating-point number 4.5 represent an integer that is not mathematically related to 4.5, the operation it is not meaningful. More specifically, int_add is intended to perform 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,

```
fun f(x) = 3+x;
```

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

```
float z = 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 is a type error because integer addition is applied to a noninteger 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 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 Pascal 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. In a program that also has records for undergraduate students, these might be represented as related type that also contains a field for the year in school of the student. Both types are written here as ML-style type expressions:

```
Student = {name : string, number : int}
Undergrad = {name : string, number : int, year : int}
```

In a program that manipulates records, there might be an expression of the form r.name, 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 r.name 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 r.name 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, ML, Smalltalk, 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. 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 ML code is two to four times faster than Lisp code. The primary reason for this speed increase is that static type checking of ML 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. To be more specific about it, 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	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 ML are not usually declared to have a specific type. The type system *infers* the types of ML 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 can be applied to a variety of programming languages. For example, type inference could, in principle, be applied to C or other programming languages. 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, ML 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.

6.3.1 First Examples of Type Inference

Here are two ML type-inference examples to give you some feel for how ML 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

```
- fun f1(x) = x + 2;
val f1 = fn : int → int
```

The function f1 adds 2 to its argument. In ML, 2 is an integer constant; the real number 2 would be written as 2.0. The operator + is overloaded; it can be either integer addition or real addition. In this function, however, it must be integer addition because 2 is an integer. 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

```
- fun f2(g,h) = g(h(0));
val f2 = fn : ('a → 'b) * (int → 'a) → 'b
```

The type ('a → 'b) * (int → 'a) → 'b inferred by the compiler is parsed as (('a → 'b) * (int → 'a)) → 'b.

The type-inference algorithm figures out that, because h is applied to an integer argument, h must be a function from int to something. The algorithm represents "something" by introducing a type variable, which is written as 'a. (This is unrelated to Lisp 'a, which would be syntax for a Lisp atom, not a variable.) 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, as shown in the preceding compiler output.

6.3.2 Type-Inference Algorithm

The ML type-inference algorithm uses the following three steps:

1. A 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. Generates 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.)

As the compiler output shows, this function is typeable; there is no type error in this declaration. However, look carefully at the type of reverse. The type 'a list → 'b list 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 AND OVERLOADING

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 'a list → int for every type 'a.

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.

Parametric and ad hoc polymorphism (overloading) are discussed in this section. Subtype polymorphism is considered 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, an ML function that sorts lists might have the ML type

```
sort : ('a * 'a → bool) * 'a list → 'a list
```

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 'a * 'a -> bool, in which the type 'a 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. ML 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 you 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 name `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.

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. ML 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, as instantiating a template several times will result in several copies of the code. The ML 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, the ML has simpler overloading and includes more information in parameterized constructs, allowing all type errors to be reported as a program unit is compiled.

6.4.3 Overloading

Parametric polymorphism can be contrasted with overloading. A symbol is *overloaded* if it has two (or more) meanings, distinguished by type, and resolved at compile time. In an influential historical paper, Christopher Strachey referred to ML-style polymorphism as *parametric polymorphism* (although ML had not been invented yet) and overloading as *ad hoc polymorphism*.

Example. In standard ML, as in many other languages, the operator + has two distinct implementations associated with it, one of type int * int → int, the other of type real*real → real. The reason that both of these operations are given the name + is that both compute numeric addition. However, at the implementation level, the two operations are really very different. Because integers are represented in one way (as binary numbers) and real numbers in another (as exponent and mantissa, following scientific notation), the way that integer addition combines the bits of its arguments to produce the bits of its result is very different from the way this is done in real addition.

An important difference between parametric polymorphism and overloading is that parameter polymorphic functions use one algorithm to operate on arguments