Chapter 3

Types

The evolution of modern programming languages is closely coupled with the development (and formalization) of the concept of data type. At the machine-language level, all values are untyped (that is, simply bit patterns). Assembler-language programmers, however, usually recognize the fundamental differences between addresses (considered relocatable) and data (considered absolute). Hence they recognize that certain combinations of addresses and data (for example, the sum of two addresses) are ill defined.

This assembler-language view of typing is flawed, however, because it views type as a property of a datum rather than a property of the cell containing the datum. That is, whether or not an operation is meaningful can usually be determined only at runtime when the actual operand values are available. An assembler will probably recognize the invalidity of an expression that adds two labels, while it will accept a code sequence that computes exactly the same thing! This weakness has led to the introduction of tagged architectures that include (at runtime) type information with a datum. Such architectures can detect the label-addition error, because the add instruction can detect that its operands are two addresses. Unfortunately, the type information included with data is usually limited to the primitive types provided by the architecture. Programmer-declared data types cannot receive the same sort of automatic correctness checking.

FORTRAN and later high-level languages improved upon assembler languages by associating type information with the locations holding data rather than the data itself. More generally, languages associate type information with identifiers, which may be variables or formal parameters. When an attribute such as a type is associated with an identifier, we say the the identifier is **bound** to the attribute. Binding that takes place at compile time is usually called **static**, and binding that takes place at runtime is called **dynamic**. **Static-typed** languages are those that bind types to identifiers at compile time. Since types are known at compile time, the compiler can detect a wide range of type errors (for example, an attempt to multiply two Boolean variables).

High-level languages prior to Pascal usually limited their concepts of data types to those provided directly by hardware (integers, reals, double precision integers and reals, and blocks of contiguous locations). Two objects had different types if it was necessary to generate different code to manipulate them. Pascal and later languages have taken a rather different approach, based on the concept of abstract data types. In Pascal, the programmer can give two objects different types even if they have the same representation and use the same generated code. Type rules have shifted from concentrating on what makes sense to the computer to what makes sense to the programmer.

1 ◆ DYNAMIC-TYPED LANGUAGES

It is possible to delay the binding of types to identifiers until runtime, leading to dynamic-typed languages. Interpreted languages (like SNOBOL, APL, and Awk) often bind types only at runtime. These languages have no type declarations; the type of an identifier may change dynamically. These are different from **typeless** languages, such as Bliss or BCPL, which have only one type of datum, the cell or word.

Delaying the binding of a type to an identifier gains expressiveness at the cost of efficiency, since runtime code must determine its type in order to manipulate its value appropriately. As an example of expressiveness, in dynamic-typed languages, arrays need not be homogeneous. As an example of loss of efficiency, even in static-typed languages, the values of choice types require some runtime checking to ensure that the expected variant is present.

2 ◆ STRONG TYPING

One of the major achievements of Pascal was the emphasis it placed on the definition of data types. It viewed the creation of programmer-declared data types as an integral part of program development. Pascal introduced the concept of strong typing to protect programmers from errors involving type mismatches. A **strongly typed language** provides rules that allow the compiler to determine the type of every value (that is, every variable and every expression).[1] Assignments and actual-formal parameter binding involving inequivalent types are invalid, except for a limited number of automatic conversions. The underlying philosophy is that different types represent different abstractions, so they ought to interact only in carefully controlled and clearly correct ways.

[1] Actually, Pascal is not completely strongly typed. Procedure-valued parameters do not specify the full procedure header, so it is possible to provide an actual parameter that does not match the formal in number or type of parameters. Untagged record variants are another loophole.

3 ◆ TYPE EQUIVALENCE

The concept of strong typing relies on a definition of exactly when types are equivalent. Surprisingly, the original definition of Pascal did not present a definition of type equivalence. The issue can be framed by asking whether the types T1 and T2 are equivalent in Figure 3.1:

Figure 3.1

```
type
     T1, T2 = array[1..10] of real;
     T3 = array[1..10] of real;
```

Structural equivalence states that two types are equivalent if, after all type identifiers are replaced by their definitions, the same structure is obtained. This definition is recursive, because the definitions of the type identifiers may themselves contain type identifiers. It is also vague, because it leaves open what "same structure" means. Everyone agrees that T1, T2, and T3 are structurally equivalent. However, not everyone agrees that records require identical field names in order to have the same structure, or that arrays require identical index ranges. In Figure 3.2, T4, T5, and T6 would be considered equivalent to T1 in some languages but not others:

Figure 3.2

```
type
     T4 = array[2..11] of real; -- same length
     T5  = array[2..10] of real; -- compatible index type
     T6 = array[blue .. red] of real; -- incompatible
          -- index type
```

Testing for structural equivalence is not always trivial, because recursive types are possible. In Figure 3.3, types TA and TB are structurally equivalent, as are TC and TD, although their expansions are infinite.

Figure 3.3

```
type
     TA = pointer to TA;
     TB = pointer to TB;
     TC =
          record
               Data : integer;
               Next : pointer to TC;
          end;
     TD =
          record
               Data : integer;
               Next : pointer to TD;
          end;
```

In contrast to structural equivalence, **name equivalence** states that two variables are of the same type if they are declared with the same type name, such as `integer` or some declared type. When a variable is declared using a **type constructor** (that is, an expression that yields a type), its type is given a new internal name for the sake of name equivalence. Type constructors include the words **array**, **record**, and **pointer to**. Therefore, type equivalence says that T1 and T3 above are different, as are TA and TB. There are different interpretations possible when several variables are declared using a single type constructor, such as T1 and T2 above. Ada is quite strict; it calls T1 and T2 different. The current standard for Pascal is more lenient; it calls T1 and T2 identical [ANSI 83]. This form of name equivalence is also called **declaration equivalence**.

Name equivalence seems to be the better design because the mere fact that two data types share the same structure does not mean they represent the same abstraction. T1 might represent the batting averages of ten members of the Milwaukee Brewers, while T3 might represent the grade-point average of ten students in an advanced programming language course. Given this interpretation, we surely wouldn't want T1 and T3 to be considered equivalent!

Nonetheless, there are good reasons to use structural equivalence, even though unrelated types may accidentally turn out to be equivalent. Applications that write out their values and try to read them in later (perhaps under the control of a different program) deserve the same sort of type-safety possessed by programs that only manipulate values internally. Modula-2+, which uses name equivalence, outputs both the type name and the type's structure for each value to prevent later readers from accidentally using the same name with a different meaning. Anonymous types are assigned an internal name. Subtle bugs arise if a programmer moves code about, causing the compiler to generate a different internal name for an anonymous type. Modula-3, on the other hand, uses structural equivalence. It outputs the type's structure (but not its name) with each value output. There is no danger that rearranging a program will lead to type incompatibilities with data written by a previous version of the program.

A language may allow assignment even though the type of the expression and the type of the destination variable are not equivalent; they only need to be **assignment-compatible**. For example, under name equivalence, two array types might have the same structure but be inequivalent because they are generated by different instances of the **array** type constructor. Nonetheless, the language may allow assignment if the types are close enough, for example, if they are structurally equivalent. In a similar vein, two types may be compatible with respect to any operation, such as addition, even though they are not type-equivalent. It is often a quibble whether to say a language uses name equivalence but has lax rules for compatibility or to say that it uses structural equivalence. I will avoid the use of "compatibility" and just talk about equivalence.

Modula-3's rules for determining when two types are structurally equivalent are fairly complex. If every value of one type is a value of the second, then the first type is a called a "subtype" of the second. For example, a record type TypeA is a subtype of another record type TypeB only if their fields have the same names and the same order, and all of the types of the fields of TypeA are subtypes of their counterparts in TypeB. An array type TypeA is a subtype of another array type TypeB if they have the same number of dimensions of the same size (although the range of indices may differ) and the same index and component types. There are also rules for the subtype relation between procedure and pointer types. If two types are subtypes of each other, they are equivalent. Assignment requires that the value being assigned be of a subtype of the target variable.[2]

After Pascal became popular, a weakness in its type system became apparent. For example, given the code in Figure 3.4,

Figure 3.4

```
type
     natural = 0 .. maxint;
```

you would expect natural numbers (which are a subrange of integers) to be equivalent to integers, so that naturals and integers might be added or assigned. On the other hand, given the code of Figure 3.5,

Figure 3.5

```
type
     feet   = 0 .. maxint;
     meters = 0 .. maxint;
```

you would probably expect `feet` and `meters` to be inequivalent. It turns out in Pascal that subranges of an existing type (or a type identifier defined as equal to another type identifier) are equivalent (subject to possible range restrictions). But I don't want `feet` and `meters` to be equivalent.

Successors to Pascal (especially Ada) have attempted to generalize type rules to allow types derived from an existing type to be considered inequivalent. In such languages, one can declare a type to be a **subtype** of an existing type, in which case the subtype and original type are type-equivalent. One can also declare a type to be **derived** from an existing type, in which case the derived and original types are not type equivalent. To implement `feet` and `meters` as inequivalent, I could therefore create types as follows:

Figure 3.6

```
type                                                    1
     feet = derived integer range 0..maxint;            2
     meters = derived integer range 0..maxint;          3
variable                                                4
     imperial_length : feet;                            5
     metric_length   : meters;                          6
begin                                                   7
     metric_length := metric_length * 2;                8
end;                                                    9
```

In order to make sure that values of a derived type that are stored by one program and read by another maintain their type, Modula-3 **brands** each derived type with a string literal. Branded values may only be read into variables with the same brand. In other words, the programmer may control which derived types are considered structurally equivalent to each other.

There is a slight problem in line 8 in Figure 3.6. The operator `*` is defined on `integer` and `real`, but I intentionally made `meters` a new type dis-

[2] The actual rule is more complex in order to account for range types and to allow pointer assignments.

tinct from `integer`. Similarly, 2 is a literal of type `integer`, not `meters`. Ada solves this problem by overloading operators, procedures, and literals associated with a derived type. That is, when `meters` was created, a new set of arithmetic operators and procedures (like `sqrt`) was created to take values of type `meters`. Similarly, `integer` literals are allowed to serve also as `meters` literals. The expression in line 8 is valid, but `metric_length * imperial_length` involves a type mismatch.[3]

The compiler determines which version of an overloaded procedure, operator, and literal to use. Intuitively, it tries all possible combinations of interpretations, and if exactly one satisfies all type rules, the expression is valid and well defined. Naturally, a smart compiler won't try all possible combinations; the number could be exponential in the length of the expression. Instead, the compiler builds a collection of subtrees, each representing a possible overload interpretation. When the root of the expression tree is reached, either a unique overload resolution has been found, or the compiler knows that no unique resolution is possible [Baker 82]. (If no appropriate overloaded procedure can be found, it may still be possible to coerce the types of the actual parameters to types that are accepted by a declared procedure. However, type coercion is often surprising to the programmer and leads to confusion.)

The concept of subtype can be generalized by allowing **extensions** and **reductions** to existing types [Paaki 90]. For example, array types can be extended by increasing the index range and reduced by decreasing the index range. Enumeration types can be extended by adding new enumeration constants and reduced by removing enumeration constants. Record types can be extended by adding new fields and reduced by removing fields. (Oberon allows extension of record types.) Extending record types is very similar to the concept of building subclasses in object-oriented programming, discussed in Chapter 5.

The resulting types can be interconverted with the original types for purposes of assignment and parameter passing. Conversion can be either by casting or by coercion. In either case, conversion can ignore array elements and record fields that are not needed in the target type and can set elements and fields that are only known in the target type to an error value. It can generate a runtime error if an enumeration value is unknown in the target type.

The advantage of type extensions and reductions is much the same as that of subclasses in object-oriented languages, discussed in Chapter 5: the new type can make use of the software already developed for the existing type; only new cases need to be specifically addressed in new software. A module that extends or reduces an imported type does not force the module that exports the type to be recompiled.

[3] In Ada, a programmer can also overload operators, so one can declare a procedure that takes a metric unit and an imperial unit, converts them, and then multiplies them.

4 ◆ DIMENSIONS

The example involving meters and feet shows that types alone do not prevent programming errors. I want to prohibit multiplying two feet values and assigning the result back into a feet variable, because the type of the result is square feet, not feet.

The AL language, intended for programming mechanical manipulators, introduced a typelike attribute of expressions called **dimension** to prevent such errors [Finkel 76]. This concept was first suggested by C. A. R. Hoare [Hoare 73], and it has been extended in various ways since then. Recent research has shown how to include dimensions in a polymorphic setting like ML [Kennedy 94]. (Polymorphism in ML is discussed extensively later in this chapter.) AL has four predeclared base dimensions: time, distance, angle, and mass. Each base dimension has predeclared constants, such as second, centimeter, and gram. The values of these constants are with respect to an arbitrary set of units; the programmer only needs to know that the constants are mutually consistent. For example, 60*second = minute. New dimensions can be declared and built from the old ones. AL does not support programmer-declared base dimensions, but such an extension would be reasonable. Other useful base dimensions would be electrical current (measured, for instance, in amps), temperature (degrees Kelvin), luminous intensity (lumens), and currency (florin). In retrospect, angle may be a poor choice for a base dimension; it is equivalent to the ratio of two distances: distance along an arc and the radius of a circle. Figure 3.7 shows how dimensions are used.

Figure 3.7

```
dimension                                                  1
     area = distance * distance;                           2
     velocity = distance / time;                           3
constant                                                   4
     mile = 5280 * foot; --  foot is predeclared           5
     acre = mile * mile / 640;                             6
variable                                                   7
     d1, d2 : distance real;                               8
     a1 : area real;                                       9
     v1 : velocity real;                                   10
begin                                                      11
     d1 := 30 * foot;                                      12
     a1 := d1 * (2 * mile) + (4 * acre);                   13
     v1 := a1 / (5 * foot * 4 * minute);                   14
     d2 := 40; -- invalid: dimension error                 15
     d2 := d1 + v1; -- invalid: dimension error            16
     write(d1/foot, "d1 in feet",                          17
          v1*hour/mile, "v1 in miles per hour");           18
end;                                                       19
```

In line 13, a1 is the area comprising 4 acres plus a region 30 feet by 2 miles. In line 14, the compiler can check that the expression on the right-hand side has the dimension of velocity, that is, distance/time, even though it is hard for a human to come up with a simple interpretation of the expression.

In languages lacking a dimension feature, abstract data types, introduced in the next section, can be used instead. The exercises explore this substitution.

5 ◆ ABSTRACT DATA TYPES

An **abstract data type** is a set of values and a set of procedures that manipulate those values. An abstract data type is analogous to a built-in type, which is also a set of values (such as integers) and operations (such as addition) on those values. Once a program has introduced an abstract data type, variables can be declared of that type and values of the type can be passed to the procedures that make up the type. The **client** of an abstract data type (that is, a part of a program that uses that type, as opposed to the part of the program that defines it) can create and manipulate values only by using procedures that the abstract data type allows. The structure of an abstract data type (usually a **record** type) is hidden from the clients. Within the definition of the abstract data type, however, procedures may make full use of that structure.

An abstract data type can be seen as having two parts: the specification and the implementation. The specification is needed by clients; it indicates the name of the type and the headers of the associated procedures. It is not necessary for the client to know the structure of the type or the body of the procedures. The implementation includes the full description of the type and the bodies of the procedures; it may include other procedures that are used as subroutines but are not needed directly by clients.

This logical separation allows a programmer to concentrate on the issues at hand. If the programmer is coding a client, there is no need to worry about how the abstract data type is implemented. The implementer may upgrade or even completely redesign the implementation, and the client should still function correctly, so long as the specification still holds.

A popular example of an abstract data type is the stack. The procedures that manipulate stacks are `push`, `pop`, and `empty`. Whether the implementation uses an array, a linked list, or a data file is irrelevant to the client and may be hidden.

Abstract data types are used extensively in large programs for modularity and abstraction. They put a barrier between the implementor of a set of routines and its clients. Changes in the implementation of an abstract data type will not influence the clients so long as the specification is preserved. Abstract data types also provide a clean extension mechanism for languages. If a new data type is needed that cannot be effectively implemented with the existing primitive types and operations (for example, bitmaps for graphics), it can be still specified and prototyped as a new abstract data type and then efficiently implemented and added to the environment.

In order to separate the specification from the implementation, programming languages should provide a way to hide the implementation details from client code. Languages like C and Pascal that have no hiding mechanism do not cater to abstract data types, even though they permit the programmer to declare new types. CLU, Ada, C++, and Modula-2 (as well as numerous other languages) provide a name-scope technique that allows the programmer to group the procedures and type declarations that make up an abstract data type and to give clients only a limited view of these declarations. All declarations that make up an abstract data type are placed in a **module**.[4] It is a name scope in which the programmer has control over what identifiers are imported from and exported to the surrounding name scope. Local identifiers that are to be seen outside a module are **exported**; all other local identifiers are invisible outside the module, which allows programmers to hide implementation details from the clients of the module. Identifiers from surrounding modules are not automatically inherited by a module. Instead, those that are needed must be explicitly **imported**. These features allow name scopes to selectively import identifiers they require and provide better documentation of what nonlocal identifiers a module will need. Some identifiers, like the predeclared types `integer` and `Boolean`, may be declared **pervasive**, which means that they are automatically imported into all nested name scopes.

Languages that support abstract data types often allow modules to be partitioned into the specification part and the implementation part. (Ada, Modula-2, C++, and Oberon have this facility; CLU and Eiffel do not.) The **specification part** contains declarations intended to be visible to clients of the module; it may include constants, types, variables, and procedure headers. The **implementation part** contains the bodies (that is, implementations) of procedures as well as other declarations that are private to the module. Typically, the specification part is in a separate source file that is referred to both by clients and by the implementation part, each of which is in a separate source file.

Partitioning modules into specification and implementation parts helps support libraries of precompiled procedures and separate compilation. Only the specification part of a module is needed to compile procedures that use the module. The implementation part of the module need not be supplied until link time. However, separating the parts can make it difficult for implementation programmers to find relevant declarations, since they might be in either part. One reasonable solution is to join the parts for the convenience of the implementor and extract just the specification part for the benefit of the compiler or client-application programmer.

Figure 3.8 shows how a stack abstract data type might be programmed.

Figure 3.8

```
module Stack;                                          1

export                                                 2
     Push, Pop, Empty, StackType, MaxStackSize;        3

constant                                               4
     MaxStackSize = 10;                                5
```

[4] You can read a nice overview of language support for modules in [Calliss 91]. Modules are used not only for abstract data types, but also for nesting name scopes, separate compilation, device control (in Modula, for example), and synchronization (monitors are discussed in Chapter 7).

```
type                                                        6
     private StackType =                                    7
          record                                            8
               Size : 0..MaxStackSize := 0;                 9
               Data : array 1..MaxStackSize of integer;     10
          end;                                              11

     -- details omitted for the following procedures        12
     procedure Push(reference ThisStack : StackType;        13
          readonly What : integer);                         14
     procedure Pop(reference ThisStack) : integer;          15
     procedure Empty(readonly ThisStack) : Boolean;         16

end; -- Stack                                               17
```

In Figure 3.8, line 3 indicates that the module exports the three procedures. It also exports the constant MaxStackSize, which the client may wish to consult, and StackType, so the client may declare variables of this type. I assume that integer and Boolean are pervasive. The code does not export enumeration types or record types. Generally, exporting these types implies exporting the enumeration constants and the record field names as well.

In Ada, the programmer can control to what extent the details of an exported type are visible to the module's clients. By default, the entire structure of an exported type, such as its record field names, is visible. If the exported type is declared as **private**, as in line 7, then only construction, destruction, assignment, equality, and inequality operations are available to the client. Even these can be hidden if the exported type is declared **limited private**. The only way the client can manipulate objects of **limited private** types is to present them as actual parameters to the module's procedures. The programmer of the implementation may change the details of private types, knowing that the change will not affect the correctness of the clients. In Oberon, record types can be partly visible and partly private.

Languages differ in how programmers restrict identifier export. In some languages, like Simula, all identifiers are exported unless explicitly hidden. Others, like Eiffel, provide for different clients (which are other modules) to import different sets of identifiers from the same module. The **export** line for Eiffel might look as shown in Figure 3.9:

Figure 3.9

```
export
     Push, Pop, Empty, StackType {ModuleA},
          MaxStackSize {ModuleA, ModuleB};
```

Here, Push, Pop, and Empty are exported to all clients. Only ModuleA may import StackType, and only two modules may import MaxStackSize. The module can thereby ensure that no client makes unauthorized use of an exported identifier. However, this approach of restricting exports requires that a module be recompiled any time its client set changes, which can be cumbersome.

An alternative, found in Modula-2, is for client modules to selectively import identifiers, as in Figure 3.10.

Figure 3.10

```
from Stack import
     Push, Pop, Empty, StackType;
```

This client has chosen not to import MaxStackSize. This approach of restricting imports is not as secure but requires less recompilation when programs change.[5]

Very large programs sometimes face confusion when importing from several modules; the same identifier may be imported from more than one module. Languages often permit or require qualified identifiers to be used in order to remove any ambiguity.

The **principle of uniform reference** suggests that clients should not be able to discover algorithmic details of exporting modules. In particular, they should not be able to distinguish whether an exported identifier is a constant, a variable, or a parameterless function. However, in many languages, the client *can* distinguish these identifiers. In C++, for example, parameterless functions are special because they are invoked with parentheses surrounding an empty list. Variables are special in that only they may be used on the left-hand side of an assignment. In Eiffel, however, the syntax is the same for all three, and exported variables are readonly, so the principle of uniform reference is upheld.

6 ◆ LABELS, PROCEDURES, AND TYPES AS FIRST-CLASS VALUES

You are used to thinking of integers as values. But to what extent is a label or a procedure a value? Can a type itself be a value? One way to address these questions is to categorize values by what sort of manipulation they allow. The following chart distinguishes **first**, **second**, and **third-class values**.

Manipulation	**Class of value**		
	First	**Second**	**Third**
Pass value as a parameter	yes	yes	no
Return value from a procedure	yes	no	no
Assign value into a variable	yes	no	no

Languages differ in how they treat labels, procedures, and types. For example, procedures are third-class values in Ada, second-class values in Pas-

[5] The difference between restricting exports and restricting imports is identical to the difference between access lists and capability lists in operating systems.

cal, and first-class values in C and Modula-2. Labels are generally third-class values, but they are second-class values in Algol-60.

Labels and procedures are similar in some ways. If a label is passed as a parameter, then jumping to it must restore the central stack to its situation when the label was elaborated. The value of a label passed as a parameter must therefore include a reference to the central stack as well as a reference to an instruction. In other words, a label is passed as a closure. Similarly, procedures that are passed as parameters generally are passed as closures, so that when they are invoked, they regain their nonlocal referencing environments. In both cases, the closure points to an activation record deeper on the central stack than the called procedure's activation record. Jumping to a passed label causes the central stack to be unwound, removing intermediate activation records. Invoking a passed procedure establishes its static chain to point somewhere deep in the central stack.

Allowing labels and procedures to be first-class values is trickier. Such values may be stored in variables and invoked at a time when the central stack no longer contains the activation record to which they point. Figure 3.11 demonstrates the problem.

Figure 3.11

```
variable                                        1
    ProcVar : procedure();                      2

procedure Outer();                              3
    variable OuterVar : integer;                4
    procedure Inner();                          5
    begin -- Inner                              6
        write(OuterVar);                        7
    end; -- Inner                               8
begin -- Outer                                  9
    ProcVar := Inner; -- closure is assigned   10
end; -- Outer                                  11

begin -- main program                          12
    Outer();                                   13
    ProcVar();                                 14
end;                                           15
```

By the time `Inner` is invoked (as the value of the procedure variable `ProcVar` in line 14), its nonlocal referencing environment, the instance of `Outer`, has been deactivated, because `Outer` has returned. I call this the **dangling-procedure problem**. Languages take various stances in regard to the dangling-procedure problem:

1. Treat any program that tries to invoke a closure with a dangling pointer as erroneous, but don't try to discover the error.
2. Prevent the bad situation from arising by language restrictions. Top-level procedures do not need a nonlocal referencing environment. In C, all procedures are top-level, so bad situations cannot arise. Modula-2 disallows assigning any but a top-level procedure as a value to a variable; it forbids the assignment in line 10 above. Neither language treats labels as first-class values.

3. Prevent the bad situation from arising by expensive implementation. The nice aspect of a central stack is that allocation and deallocation are inexpensive and occur in a strict stack order as procedures are invoked and return. This inexpensive mechanism can be replaced by activation records that are allocated from the heap and are linked together. A reference-count mechanism suffices for reclamation, since there will be no cycles. Activation, deactivation, and access to referencing environments is likely to be slower than if a stack were used.

Labels as first-class values are frightening for another reason: they can be stored in a variable and repeatedly invoked. Therefore, the procedure that elaborates a label (that is, that defines the label) can return more than once, because that label may be invoked repeatedly. Multiply-returning procedures are certain to be confusing.

So far, I have only dealt with labels and procedures, but the same questions can also be asked about types. Types as parameters, type variables and procedures that return types could be very useful. For example, an abstract data type implementing stacks really ought to be parameterized by the type of the stack element, rather than having it simply "wired in" as `integer`, as in Figure 3.8 (page 63). Ada and C++ allow a limited form of **type polymorphism**, that is, the ability to partially specify a type when it is declared and further specify it later. They implement polymorphism by permitting modules (that is, the name scopes that define abstract data types) to accept type parameters. Such modules are called **generic modules**.[6] A declaration of a generic module creates a template for a set of actual modules. The stack example can be rewritten as in Figure 3.12.

Figure 3.12

```
generic(type ElementType) module Stack;

export
     Push, Pop, Empty, StackType, MaxStackSize;

constant
     MaxStackSize = 10;

type
     private StackType =
          record
               Size : 0..MaxStackSize := 0;
               Data : array 1..MaxStackSize of ElementType;
          end;

```

[6] Ada and C++ allow generic procedures in addition to generic modules.

```
        -- details omitted for the following procedures          13
        procedure Push(reference ThisStack : StackType;            14
             readonly What : ElementType);                         15
        procedure Pop(reference ThisStack) : ElementType;          16
        procedure Empty(readonly ThisStack) : Boolean;             17

end; -- Stack                                                      18

module IntegerStack = Stack(integer);                              19
```

To create an instance of a generic module, I **instantiate** it, as in line 19. Instantiation of generic modules in Ada and C++ is a compile-time, not a runtime, operation — more like macro expansion than procedure invocation. Compilers that support generic modules need to store the module text in order to create instances.

The actual types that are substituted into the formal generic parameters need not be built-in types like `integer`; program-defined types are also acceptable. However, the code of the generic module may require that the actual type satisfy certain requirements. For example, it might only make sense to include pointer types, or array types, or numeric types. Ada provides a way for generic modules to stipulate what sorts of types are acceptable. If the constraint, for example, is that the actual type be numeric, then Ada will permit operations like + inside the generic module; if there the constraint only requires that assignment work, Ada will not allow the + operation. Now, program-defined types may be numeric in spirit. For example, a complex number can be represented by a record with two fields. Both Ada and C++ allow operators like + to be overloaded to accept parameters of such types, so that generic modules can accept these types as actual parameters with a "numeric" flavor.

More general manipulation of types can also be desirable. Type constructors like **array** or **record** can be viewed as predeclared, type-valued procedures. It would be nice to be able to allow programmers to write such type constructors. Although this is beyond the capabilities of today's mainstream languages, it is allowed in Russell, discussed later in this chapter. A much fuller form of polymorphism is also seen in the ML language, the subject of the next section.

7 ◆ ML

Now that I have covered some issues surrounding types, I will present a detailed look at one of the most interesting strongly typed languages, ML [Harper 89; Paulson 92]. ML, designed by Robin Milner, is a functional programming language (the subject of Chapter 4), which means that procedure calls do not have any **side effects** (changing values of variables) and that there are no variables as such. Since the only reason to call a procedure is to get its return value, all procedures are actually functions, and I will call them that. Functions are first-class values: They can be passed as parameters, returned as values from procedures, and embedded in data structures. **Higher-order functions** (that is, functions returning other functions) are used extensively. Function application is the most important control construct, and it is extremely uniform: all functions take exactly one parameter and return exactly one result. Parameters and results can, however, be arbitrary structures, thereby achieving the effect of passing many parameters and producing many results. Parameters to functions are evaluated before invocation and are passed in value mode.

ML is an **interactive** language. An ML session is a dialogue of questions and answers. Interaction is achieved by an **incremental compiler**, which translates new code (typically new functions) and integrates it into already-compiled code. Incremental compilers have some of the advantages of interpreted languages (fast turnaround for dialogues) and of compiled languages (high execution speed).

ML is statically scoped. All identifiers are associated with meanings according to where they occur in the program text, not according to runtime execution paths. This design avoids name conflicts in large programs, because identifier names can be hidden in local scopes, and it prevents accidental damage to existing programs. Static scoping greatly improves the security and, incidentally, the efficiency of an interactive language.

ML is strongly typed. Every ML expression has a statically determined type. The type of an expression is usually inferred from the way it is used so that type declarations are not necessary. This **type inference** property is very useful in interactive use, when it would be distracting to have to provide type information. However, it is always possible for the programmer to specify the type of any value. Adding redundant type information can be a good documentation practice in large programs. Strong typing guarantees that expressions will not generate type errors at runtime. Static type checking promotes safety; it detects at compile time a large proportion of bugs in programs that make extensive use of the ML data-structuring capabilities (type checking does not help so much in numerical or engineering programs, since there is no concept of dimension). Usually, only truly “logical” bugs are left after compilation.

ML has a **polymorphic type** mechanism. Type expressions may contain **type identifiers**, which stand for arbitrary types. With such expressions, the ML programmer can express the type of a function that behaves uniformly on a class of parameters of different (but structurally related) types. For example, the `length` function, which computes the length of a list, has type `'a` **`list`** `-> int`, where `'a` is a type identifier standing for any type. `Length` can work on lists of any type (lists of integers, lists of functions, lists of lists, and so forth), because it disregards the elements of the list. The polymorphic type mechanism gives ML much of the expressiveness of dynamic-typed languages without the conceptual cost of runtime type errors or the computational cost of runtime type checking.

ML has a rich collection of data types, and the programmer can define new abstract data types. In fact, arrays are not part of the language definition; they can be considered as a predeclared abstract data type that just happens to be more efficient than its ML specification would lead one to believe.

ML has an exception-handling mechanism that allows programs to uniformly handle predeclared and programmer-declared exceptions. (Exception handling is discussed in Chapter 2.) Exceptions can be selectively trapped, and handlers can be specified.

ML programs can be grouped into separately compiled modules. Dependencies among modules can be easily expressed, and the sharing of common submodules is automatically guaranteed. ML keeps track of module versions to detect compiled modules that are out of date.

I will describe some (by no means all!) of the features of SML, the Standard ML of New Jersey implementation [Appel 91]. I should warn you that I have intentionally left out large parts of the language that do not pertain directly to the concept of type. For example, programming in the functional style, which is natural to ML, is discussed in Chapter 4. If you find functional programming confusing, you might want to read Chapter 4 before the rest of this chapter. In addition, I do not discuss how ML implements abstract data types, which is mostly similar to what I have covered earlier. I do not dwell on one very significant type constructor: **ref**, which represents a pointer to a value. Pointer types introduce variables into the language, because a program can associate an identifier with a pointer value, and the object pointed to can be manipulated (assigned into and accessed). ML is therefore not completely functional. The examples are syntactically correct SML program fragments. I mark the user's input by `in` and ML's output by `out`.

7.1 Expressions

ML is an expression-based language; all the standard programming constructs (conditionals, declarations, procedures, and so forth) are packaged as expressions yielding values. Strictly speaking, there are no statements: even operations that have side effects return values.

It is always meaningful to supply an arbitrary expression as the parameter to a function (when the type constraints are satisfied) or to combine expressions to form larger expressions in the same way that simple constants can be combined.

Arithmetic expressions have a fairly conventional appearance; the result of evaluating an expression is presented by ML as a value and its type, separated by a colon, as in Figure 3.13.

Figure 3.13

```
in:  (3 + 5) * 2;
out: 16 : int
```

String expressions are straightforward (Figure 3.14).

Figure 3.14

```
in:  "this is it";
out: "this is it" : string
```

Tuples of values are enclosed in parentheses, and their elements are separated by commas. The type of a tuple is described by the type constructor `*`.[7]

[7] The `*` constructor, which usually denotes product, denotes here the set-theoretic Cartesian product of the values of the component types. A value in a Cartesian product is a compound formed by selecting one value from each of its underlying component types. The number of values is the product of the number of values of the component types, which is one reason this set-theoretic operation is called a product.

Figure 3.15

```
in:  (3,4);
out: (3,4) : int * int

in:  (3,4,5);
out: (3,4,5) : int * int * int
```

Lists are enclosed in square brackets, and their elements are separated by commas, as in Figure 3.16. The list type constructor is the word **list** after the component type.

Figure 3.16

```
in:  [1,2,3,4];
out: [1,2,3,4] : int list

in:  [(3,4),(5,6)];
out: [(3,4),(5,6)] : (int * int) list
```

Conditional expressions have ordinary **if** syntax (as usual in expression-based languages, **else** cannot be omitted), as in Figure 3.17.

Figure 3.17

```
in:  if true then 3 else 4;
out: 3 : int

in:  if (if 3 = 4 then false else true)
          then false else true;
out: false : bool
```

The **if** part must be a Boolean expression. Two predeclared constants true and false denote the Boolean values; the two binary Boolean operators **orelse** and **andalso** have short-circuit semantics (described in Chapter 1).

7.2 Global Declarations

Values are bound to identifiers by **declarations**. Declarations can appear at the top level, in which case their scope is global, or in blocks, in which case they have a limited local scope spanning a single expression. I will first deal with global declarations.

Declarations are not expressions. They establish bindings instead of returning values. Value bindings are introduced by the keyword **val**; additional value bindings are prefixed by **and** (Figure 3.18).

Figure 3.18

```
in:  val  a = 3 and
          b = 5 and
          c = 2;
out: val c = 2 : int
     val b = 5 : int
     val a = 3 : int

in:  (a + b) div c;
out: 4 : int
```

In this case, I have declared the identifiers a, b, and c at the top level; they will be accessible from now on unless I redeclare them. Value bindings printed by ML are always prefixed by **val**, to distinguish them from type bindings and module bindings.

Identifiers are not variables; they are named constants. All identifiers must be initialized when introduced. The initial values determine their types, which need not be given explicitly.

Value declarations are also used to declare functions, with the syntax shown in Figure 3.19.

Figure 3.19

```
in:  val f = fn x => x + 1;                                   1
out: val f = fn : int -> int                                  2

in:  val g = fn (a,b) => (a + b) div 2;                       3
out: val g = fn : (int * int) -> int                          4

in:  (f 3, g(8,4));                                           5
out: (4,6) : int * int                                        6
```

The function `f` declared in line 1 has one formal parameter, `x`. The result of a function is the value of its body, in this case `x+1`. The arrow `->` (lines 2 and 4) is a type constructor that takes two types (the left operand is the type of the parameter of the function, and the right operand is the type of its return value) and returns the type that describes a function that takes such a parameter and returns such a result. In other words, functions have types that can be described by a constructor syntax, which is necessary if functions are to be first-class values and if all values are to have describable types. The function `g` declared in line 3 has a single parameter, the tuple of integers formally named `a` and `b`.

Parameters to functions do not generally need to be parenthesized (both in declarations and applications): the simple juxtaposition of two expressions is interpreted as a **function application**, that is, as invoking the function (the first expression) with the given parameter (the second expression). Function application is an invisible, high-precedence, binary operator; expressions like `f 3 + 4` are parsed like `(f 3) + 4` and not like `f (3 + 4)`. Parentheses are needed in line 5, because `g 8,4` would be interpreted as `(g 8),4`.

The identifiers `f` and `g` are bound to functions. Since functions are first-class values, they can stand alone, without being applied to parameters (Figure 3.20).

Figure 3.20

```
in:  (f, f 3);                                                1
out: (fn,4) : (int -> int) * int                              2

in:  val h = g;                                               3
out: val h = fn : (int * int) -> int                          4
```

In line 1, `f` is both presented alone and applied to 3. Functional values, shown in line 2, are always printed **fn** without showing their internal structure. In line 3, `h` is mapped to the function `g`. I could also have written line 3 as **val** `h =` **fn** `(a,b) => g(a,b)`.

Identifiers are statically scoped, and their values cannot change. When new identifiers are declared, they may override previously declared identifiers having the same name, but those other identifiers still exist and still retain their old values. Consider Figure 3.21.

Figure 3.21

```
in:  val a = 3;                                1
out: val a = 3 : int                           2

in:  val f = fn x => a + x;                    3
out: val f = fn : int -> int                   4

in:  val a = [1,2,3];                          5
out: val a = [1,2,3] : int list                6

in:  f 1;                                      7
out: 4 : int                                   8
```

The function f declared in line 3 uses the top-level identifier a, which was bound to 3 in line 1. Hence f is a function from integers to integers that returns its parameter plus 3. Then a is redeclared at the top level (line 5) to be a list of three integers; any subsequent reference to a will yield that list (unless it is redeclared again). But f is not affected at all: the old value of a was frozen in f at the moment of its declaration, and f continues to add 3 to its actual parameter. The nonlocal referencing environment of f was bound when it was first elaborated and is then fixed. In other words, ML uses deep binding.

Deep binding is consistent with static scoping of identifiers. It is quite common in block-structured programming languages, but it is rarely used in interactive languages like ML. The use of deep binding at the top level may sometimes be counterintuitive. For example, if a function f calls a previously declared function g, then redeclaring g (for example, to correct a bug) will not change f, which will keep calling the old version of g.

The **and** keyword is used to introduce sets of independent declarations: None of them uses the identifiers declared by the other bindings in the set; however, a declaration often needs identifiers introduced by previous declarations. The programmer may introduce such declarations sequentially, as in Figure 3.22.

Figure 3.22

```
in:  val a = 3; val b = 2 * a;                 1
out: val a = 3 : int                           2
     val b = 6 : int                           3
```

A function that expects a pair of elements can be converted to an infix operator for convenience, as seen in line 2 of Figure 3.23.

Figure 3.23

```
in:  val plus = fn (a,b) => a + b : int;
in:  infix plus;
in:  4 plus 5;
out: 9 : int;
```

7.3 Local Declarations

Declarations can be made local by embedding them in a block (see Figure 3.24), which is formed by the keywords **let** (followed by the declarations), **in** (followed by a single expression, the body), and **end**. The scope of the declaration is limited to this body.

Figure 3.24

```
in:  let
         val a = 3 and b = 5
     in
         (a + b) div 2
     end;
out: 4 : int
```

Here the identifiers a and b are mapped to the values 3 and 5 respectively for the extent of the expression (a + b) **div** 2. No top-level binding is introduced; the whole **let** construct is an expression whose value is the value of its body.

Just as in the global scope, identifiers can be locally redeclared, hiding the previous declarations (whether local or not). It is convenient to think of each redeclaration as introducing a new scope. Previous declarations are not affected, as demonstrated in Figure 3.25.

Figure 3.25

```
in:  val a = 3 and b = 5;
out: val b = 5 : int;
     val a = 3 : int;

in:  (let val a = 8 in a + b end, a);
out: (13,3) : int * int
```

The body of a block can access all the identifiers declared in the surrounding environment (like b), unless they are redeclared (like a).

Declarations can be composed sequentially in local scopes just as in the global scope, as shown in Figure 3.26.

Figure 3.26

```
in:  let
         val a = 3;
         val b = 2 * a
     in
         (a,b)
     end;
out: (3,6) : int * int
```

7.4 Lists

Lists are homogeneous; that is, all their components must have the same type. The component type may be anything, such as strings, lists of integers, and functions from integers to Booleans.

Many functions dealing with lists can work on lists of any kind (for example to compute the length); they do not have to be rewritten every time a new kind of list is introduced. In other words, these functions are naturally polymorphic; they accept a parameter with a range of acceptable types and return a result whose type depends on the type of the parameter. Other functions are more restricted in what type of lists they accept; summing a list makes sense for integer lists, but not for Boolean lists. However, because ML allows functions to be passed as parameters, programmers can generalize such restricted functions. For example, summing an integer list is a special case of a more general function that accumulates a single result by scanning a list and applying a commutative, associative operation repeatedly to its elements. In particular, it is not hard to code a polymorphic `accumulate` function that can be used to sum the elements of a list this way, as in Figure 3.27.

Figure 3.27

```
in:     accumulate([3,4,5], fn (x,y) => x+y, 0);      1
out:    12 : int                                      2
```

Line 1 asks for the list `[3,4,5]` to be accumulated under integer summation, whose identity value is `0`. Implementing the `accumulate` function is left as an exercise.

The fundamental list constructors are `nil`, the empty list, and the right-associative binary operator `::` (pronounced "cons," based on LISP, discussed in Chapter 4), which places an element (its left operand) at the head of a list (its right operand). The square-brackets constructor for lists (for example, `[1,2,3]`) is an abbreviation for a sequence of `cons` operations terminated by `nil`: `1 :: (2 :: (3 :: nil))`. `Nil` itself may be written `[]`. ML always uses the square-brackets notation when printing lists.

Expression	**Evaluates to**
`nil`	`[]`
`1 :: [2,3]`	`[1,2,3]`
`1 :: 2 :: 3 :: nil`	`[1,2,3]`

Other predeclared operators on lists include

- `null`, which returns `true` if its parameter is `nil`, and `false` on any other list.
- `hd`, which returns the first element of a nonempty list.
- `tl`, which strips the first element from the head of a nonempty list.
- `@` (append), which concatenates lists.

`Hd` and `tl` are called **selectors**, because they allow the programmer to select a component of a structure. Here are some examples that use the predeclared operators.

Expression	Evaluates to
`null []`	`true`
`null [1,2,3]`	`false`
`hd [1,2,3]`	`1`
`tl [1,2,3]`	`[2,3]`
`[1,2] @ []`	`[1,2]`
`[] @ [3,4]`	`[3,4]`
`[1,2] @ [3,4]`	`[1,2,3,4]`

Lists are discussed in greater depth in Chapter 4, which discusses functional languages. They are interesting to us in this chapter because of their interaction with ML's type rules and with patterns.

7.5 Functions and Patterns

Because all functions take exactly one parameter, it is often necessary to pass complicated structures in that parameter. The programmer may want the formal parameter to show the structure and to name its components. ML patterns provide this ability, as shown in Figure 3.28.

Figure 3.28

```
in:  val plus = fn (a,b) => a + b;
```

I need to say `a + b : int`, as I will show later. Here, the function `plus` takes a single parameter, which is expressed as a pattern showing that the parameter must be a tuple with two elements, which are called formally `a` and `b`.[8] This pattern does not force the actual parameter to be presented as an explicit tuple, as Figure 3.29 shows.

Figure 3.29

```
in:  plus(3,4)                                                 1
out: 7 : int                                                   2

in:  let                                                       3
          val x = (3,4)                                        4
     in                                                        5
          plus x                                               6
     end;                                                      7
out: 7 : int                                                   8
```

The first example (line 1) builds the actual parameter to `plus` explicitly from two components, 3 and 4. The comma between them is the tuple constructor. The syntax is contrived to remind the programmer that the intent is to provide two parameters.[9] The second example presents a single variable `x` as the actual parameter (line 6); the compiler can tell that it has the right type,

[8] The declaration is actually ambiguous; ML cannot determine which meaning of + is meant.

[9] In practice, a compiler can usually optimize away the extra pair constructions.

namely `int * int`.

Figure 3.30 shows how the declaration of `plus` can be written in single-parameter form.

Figure 3.30

```
in:  val plus = fn x =>                                   1
          let                                             2
               val (a,b) = x                              3
          in                                              4
               a + b                                      5
          end;                                            6
out: val plus = fn : int * int -> int
```

This example avoids a pattern for the formal parameter, now called `x` (line 1). However, it introduces a pattern in line 3 to produce the same effect. This pattern constrains `x` (retroactively) to be a pair, and it binds `a` and `b` to the two components. Figure 3.31 also uses patterns, both in specifying formal parameters and in declaring identifiers.

Figure 3.31

```
in:  val f = fn [x,y,z] => (x,y,z);                       1
out: val f = fn : 'a list -> 'a * 'a * 'a                 2

in:  val (a,b,c) = f[1,2,3];                              3
out: val c = 3 : int                                      4
     val b = 2 : int                                      5
     val a = 1 : int                                      6
```

The function `f` (line 1) returns three values packaged as a tuple. The pattern `a,b,c` in line 3 is used to unpack the result of `f[1,2,3]` into its components.

Patterns in ML come in many forms. For example, a pattern `[a,b,c]` matches a list of exactly three elements, which are mapped to `a`, `b`, and `c`; a pattern `first::rest` matches a nonempty list with its first element associated with `first`, and its other elements to `rest`. Similarly, `first::second::rest` matches a list with at least two elements, and so forth. The most common patterns are tuples like `(a,b,c)`, but more complicated patterns can be constructed by nesting, such as `([a,b],c,(d,e)::_)`. The don't-care pattern `_` matches any value without establishing any binding. Patterns can conveniently replace selector operators for unpacking data.

Patterns allow functions to be coded using **case analysis**, that is, testing the value of the parameter to determine which code to execute. This situation is most common in recursive functions, which must first test if the parameter is the base case, which is treated differently from other cases. ML programs seldom need to use the **if** expression for this purpose. Instead, pattern alternatives are used, as in Figure 3.32.

Figure 3.32

```
in:  val rec summation =                                  1
          fn nil => 0                                     2
          | (head :: tail) => head + summation tail;      3
out: val summation = fn : int list -> int                 4
```

The **rec** declaration in line 1 indicates that the scope of the declaration of summation starts immediately, not after the declaration. This wide scope allows the invocation of summation in line 3 to refer to this function itself. The formal parameter is presented as a series of alternatives separated by the | symbol. Each alternative gives a different pattern, thereby restricting the allowable values of the actual parameter and naming its formal components. The patterns are evaluated sequentially when the function is invoked. If a pattern matches the actual parameter, the identifiers in the pattern act as formal parameters that are bound to the respective parts of the actual parameter, and the corresponding action is executed. If several patterns match the actual parameter, only the first matching one is activated. If all patterns fail to match the actual parameter, a runtime exception occurs. In this case, the first pattern requires that the parameter be an empty list; the second matches any nonempty list and names its components head and tail.[10]

Patterns used for case analysis should obey several properties. First, they must all be of the same type. In Figure 3.32, both nil and (head :: tail) are of a list type with unspecified component type. ML disallows a declaration in which the formal-parameter patterns cannot be unified into a single type. Second, they should be exhaustive, covering all possible cases. The ML compiler will issue a warning if it detects a nonexhaustive match. (In the example, omitting either of the two cases elicits such a warning.) Invoking a function with a nonexhaustive match can lead to a Match exception being raised (exceptions are discussed in Chapter 2). Third, good style dictates that they should not overlap. The ML compiler issues a warning if it detects a redundant match. The first matching pattern will be used when the function is invoked.

Patterns are found in other languages as well. CSP (Chapter 7) and Prolog (Chapter 8) use patterns both for unpacking parameters and for introducing restrictions on their values. String-processing languages (Chapter 9) use patterns for testing data and extracting components.

7.6 Polymorphic Types

A function is **polymorphic** when it can work uniformly over parameters of different data types. For example, the function in Figure 3.33 computes the length of a list.

Figure 3.33

```
in:  val rec length =                                 1
           fn nil => 0                                 2
           | (_ :: tail) => 1 + length tail;           3
out: val length = fn : 'a list -> int                  4

in:  (length [1,2,3], length ["a","b","c","d"]);       5
out: (3,4) : int * int                                 6
```

The type of length inferred by the compiler (line 4) contains a type identifier ('a), indicating that any kind of list can be used, such as an integer list or a

[10] The parentheses in the second pattern are not needed; I put them in for the sake of clarity. Parentheses are required in tuples, however.

string list. A type identifier is any ordinary identifier prefixed by one or more tic marks ('). For convenience, we can pronounce 'a as "alpha" and 'b as "beta."

A type is **polymorphic** if it contains type identifiers; otherwise it is **monomorphic**. A type identifier can be mapped to any ML type and thereby form an instance of that type. For example, int **list** is a monomorphic instance of 'a **list**. Instances of polymorphic types may themselves be polymorphic. For example, ('b * 'c) **list** is a polymorphic instance of 'a **list**.

Several type identifiers can be used in a type, and each identifier can appear several times, expressing contextual relationships between components of a type. For example, 'a * 'a is the type of all pairs having components of the same type. Contextual constraints can also be expressed between parameters and results of functions, as in the identity function, which has type 'a -> 'a, or the function in Figure 3.34, which swaps pairs:

Figure 3.34

```
in:  val swap = fn (x,y) => (y,x);
out: val swap = fn : ('a * 'b) -> ('b * 'a)

in:  swap ([],"abc");
out: ("abc",[]) : string * ('a list)
```

The empty list [] is a polymorphic expression of type 'a **list**, because it can be considered an empty integer list, an empty string list, or some other empty list.

In printing out polymorphic types, ML uses the type identifiers 'a, 'b, and so on in succession, starting again from 'a at every new top-level declaration.

Several primitive functions are polymorphic. For example, you have already encountered the list operators, whose types appear in the following table.

Operator	Type
nil	'a **list**
::	('a * 'a **list**) -> 'a **list**
null	('a **list**) -> bool
hd	('a **list**) -> 'a
tl	('a **list**) -> ('a **list**)
@	('a **list** * 'a **list**) -> ('a **list**)

If these operators were not polymorphic, a program would need different primitive operators for all possible types of list elements. The 'a shared by the two parameters of :: (cons) prevents any attempt to build lists containing expressions of different types.

The user can always determine the type of any ML function or expression by typing its name at the top level; the expression is evaluated and, as usual, its type is printed after its value, as in Figure 3.35.

Figure 3.35

```
in:  [];
out: [] : 'a list

in:  hd;
out: fn : ('a list) -> 'a
```

7.7 Type Inference

A type can be a type identifier ('a, 'b, ...), or it can be constructed with type constructors. Predeclared type constants, like `int` and `bool`, are actually nullary type constructors. Polymorphic type constructors include `->` , `*` , and **`list`**.

As a simple example of type inference, if I declare `Identity = fn x => x`, then `Identity` has type `'a -> 'a`, because it returns unchanged expressions of any type. If I have the application `Identity 0`, then since `0` is of type `int`, this application of `Identity` is specialized to `int -> int`, and hence the value of the application is of type `int`.

The following table summarizes the types assumed for a variety of literals and operators, some of which are naturally polymorphic.

Expression	**Type**
`true`	`bool`
`false`	`bool`
`1`	`int`
`+`	`(int * int) -> int`
`=`	`('a * 'a) -> bool`
`nil`	`'a list`
`::`	`('a * 'a list) -> 'a list`
`hd`	`'a list -> 'a`
`tl`	`'a list -> 'a list`
`null`	`'a list -> bool`

A type expression may contain several occurrences of the same type identifier, allowing the programmer to specify type dependencies. Thus `'a -> 'a` represents a function whose parameter and result type are the same, although it does not specify what that type is. In a type expression, all occurrences of a type identifier must represent the same type. Discovering that type is done by an algorithm called **unification**; it finds the strongest common type constraint for (possibly polymorphic) types. For example, `int -> int` and `(int -> bool) -> (int -> bool)` can be unified to `'a -> 'a`. They can also be unified to `'a -> 'b`, but that is a weaker constraint. In fact, they can be unified to the weakest possible type, `'a`.

To perform polymorphic type inference, ML assigns a type identifier to each expression whose type is unknown and then solves for the type identifiers. The algorithm to solve for the type identifiers is based on repeatedly applying constraints:

1. All occurrences of the same identifier (under the scoping rules) have the same type.
2. In a **let rec** declaration, all free occurrences of the declared identifier (that is, those that are not bound by new declarations in nested name scopes) have the same type.
3. In a conditional expression such as **if** B **then** branch1 **else** branch2, B must have type bool, and branch1, branch2, and the total expression have the same type. A shorthand expression for this constraint would be **if** bool **then** 'a **else** 'a : 'a.
4. Function application: ('a -> 'b) 'a : 'b. This means that applying a function to a parameter yields a result of the appropriate type. This constraint can be used to derive the type of the parameter, the type of the result, or the type of the function.
5. Function abstraction: **fn** 'a => 'b : 'a -> 'b. This means that an anonymous function has a type based on the type of its parameter and its result. Again, this constraint can be used to derive the type of the parameter, the type of the result, or the type of the function.

Let me now illustrate type inference based on the code in Figure 3.36.

Figure 3.36

```
in:  val rec length = fn AList =>
          if null AList then
               0
          else
               1 + length(tl AList);
out: val length = fn : 'a list -> int
```

To begin, the following type constraints hold:

Expression	Type
length	't1 -> 't2
AList	't3
null AList	bool
1 + length(tl AList))	int

Using the type of null, that is, 'a **list** -> bool, it must be that 't3 = 'a **list**, and because + returns int, it must be that length(tl AList) : int; hence 't2 = int. Now tl : 'a **list** -> 'a **list**, so tl AList : 'a **list**. Therefore, length : 'a **list** -> int, which agrees with the intuitive declaration of a length function.

Although type inference may appear trivial, interesting problems can arise. Consider first self-application, as shown in Figure 3.37.

Figure 3.37

```
in:  val F = fn x => x x;
out: Type clash  in:  (x x)
     Looking  for a:  'a
     I have found a:  'a -> 'b
```

Here, F : 'a -> 'b, so x : 'a. Since (x x) : 'b, therefore x : 'a -> 'b, which leads to the conclusion that 'a = 'a -> 'b, which has no (finite) solution. Under the type inference rules, F has an invalid type.

Another interesting problem is illustrated in Figure 3.38.

Figure 3.38

```
in:  val f1 = fn x => (x 3, x true);
out: Type clash in:  (x true)
     Looking for a:  int
     I have found a:  bool
```

The problem is how to type the parameter x, which clearly is a function. ML treats functions as first-class values, so passing a function as a parameter isn't a problem. The first application of x to 3 suggests a type of int -> 'a, while the second application to true suggests bool -> 'b. You might be tempted to generalize the type of x to 'c -> 'a, so any function would be valid as a parameter to f. But, for example, not (of type bool -> bool) matches 'c -> 'a, but not can't take an integer parameter, as required in the first application of x. Rather, ML must conclude that f can't be typed using the rules discussed above and hence is invalid.

Now consider the valid variant of f1 shown in Figure 3.39.

Figure 3.39

```
in:  let                                  1
           val f2 = fn x => x             2
     in                                   3
           ((f2 3), (f2 true))            4
     end;                                 5
out: (3,true) : int * bool                6
```

Now f2's type is 'a -> 'a, so both calls of f2 are valid. The significance is that a parameter to a function, like x in f1, must have a single type that works each time it appears. In this case, neither int -> 'a nor bool -> 'a works. On the other hand, polymorphic functions like f2 can acquire different inferred types each time they are used, as in line 4 of Figure 3.39.

Even with its polymorphism, ML is strongly typed. The compiler knows the type of every value, even though that type may be expressed with respect to type identifiers that are not yet constrained. Furthermore, ML is **type-safe**; that is, whenever a program passes the compile-time type-checking rules, no runtime type error is possible. This concept is familiar in monomorphic languages, but not in polymorphic languages.

The type mechanism of ML could be enhanced to allow f1 in Figure 3.38 to be typed. ML could provide a choice type, composed of a fixed number of alternatives, denoted by **alt**. Then f1 could be typed as ((int **alt** bool) -> 'a) -> ('a * 'a). I could use **datatype** for this purpose, but it would not be as elegant.

7.8 Higher-Order Functions

ML supports higher-order functions, that is, functions that take other functions as parameters or deliver functions as results. Higher-order functions are particularly useful to implement **partial application**, in which an invocation provides only some of the expected parameters of a function, as in Figure 3.40.

Figure 3.40

```
in:  val times = fn a => (fn b : int => a * b);          1
out: val times = fn : int -> (int -> int)                2

in:  times 3 4;                                          3
out: 12 : int                                            4

in:  val twice = times 2;                                5
out: val twice = fn : int -> int                         6

in:  twice 4;                                            7
out: 8 : int                                             8
```

The type of times (lines 1–2) is unexpectedly complex, because I have chosen to split the two parameters. (I explicitly indicate that b is of type int to resolve the * operator.) In line 3, times 3 4 is understood as (times 3) 4. Times first takes the actual parameter 3 and returns a function from integers to integers; this anonymous function is then applied to 4 to give the result 12. This unusual definition allows me to provide only the first parameter to times if I wish, leading to partial application. For example, I declare twice in line 5 by calling times with only one parameter. When I wish to supply the second parameter, I can do so, as in line 7.

The function-composition function, declared in Figure 3.41, is a good example of partial application. It also has an interesting polymorphic type.

Figure 3.41

```
in:  val compose = fn (f,g) => (fn x => f (g x));        1
out: val compose = fn :                                  2
          (('a -> 'b) * ('c -> 'a)) -> ('c -> 'b)        3

in:  val fourTimes = compose(twice,twice);               4
out: val fourTimes = fn : int -> int                     5

in:  fourTimes 5;                                        6
out: 20 : int                                            7
```

Compose takes two functions f and g as parameters and returns a function that when applied to a parameter x returns f (g x). Composing twice with itself, by partially applying compose to the pair (twice,twice), produces a function that multiplies numbers by four. Function composition is actually a predeclared binary operator in ML written as o. The composition of f and g can be written f o g.

Suppose now that I need to partially apply a function f that, like plus, takes a pair of parameters. I could redeclare f as in Figure 3.42.

Figure 3.42

```
val f = fn a => (fn b => f(a,b))
```

Since I did not say **rec**, the use of f inside the declaration refers to the preexisting function f. The new f can be partially applied and uses the old f as appropriate.

To make this conversion more systematic, I can write a function that transforms any function of type ('a * 'b) -> 'c (that is, it requires a pair of parameters) into a function of type 'a -> ('b -> 'c) (that is, it can be partially applied). This conversion is usually called **currying** the function.[11] Figure 3.43 declares a curry function.

Figure 3.43

```
in:  val curry = fn f => (fn a => (fn b => f(a,b)));          1
out: val curry = fn :                                          2
          (('a * 'b) -> 'c) -> ('a -> ('b -> 'c))              3

in:  val curryPlus = curry plus;                               4
out: val curryPlus = fn : int -> (int -> int)                  5

in:  val successor = curryPlus 1;                              6
out: val successor = fn : int -> int                           7
```

The higher-order function curry (line 1) takes any function f defined on pairs and two parameters a and b, and applies f to the pair (a,b). I have declared curry so that it can be partially applied; it needs to be provided at least with f, but not necessarily with a or b. When I partially apply curry to plus (line 4), I obtain a function curryPlus that works exactly like plus, but which can be partially applied, as in line 6.

7.9 ML Types

The type of an expression indicates the set of values it may produce. Types include primitive types (integer, real, Boolean, string) and structured types (tuples, lists, functions, and pointers). An ML type only gives information about attributes that can be computed at compile time and does not distinguish among different sets of values having the same structure. Hence the set of positive integers is not a type, nor is the set of lists of length 3. In contrast, Pascal and Ada provide subtypes that restrict the range of allowable values.

On the other hand, ML types can express structural relations within values, for example, that the right part of a pair must have the same type as the left part of the pair, or that a function must return a value of the same type as its parameter (whatever that type may be).

Types are described by recursively applied type constructors. Primitive types like int are type constructors that take no parameters. Structured types are built by type constructors like * (Cartesian product, for tuples), **list**, -> (for functions), and **ref** (for pointers). Type constructors are usually infix or suffix: int * int, int **list**, int -> int, and int **ref** are the types

[11] Haskell B. Curry was a logician who popularized this idea.

of integer pairs, lists, functions, and pointers. Type constructors can be arbitrarily nested. For example, `(int -> int) list` is the type of lists of integer-to-integer functions.

Type identifiers can be used to express polymorphic types. Polymorphic types are mostly useful as types of functions, although some nonfunctional expressions, like `[]`, of type `'a list`, are also polymorphic. A typical example of a polymorphic function is `hd`, of type `'a list -> 'a`. The type of `hd` indicates that it can accept any list and that the type of the result is the same as the type of the elements of the list.

Every type denotes a **type domain**, which is the set of all values of the given type. For example, `int * int` denotes the domain of integer pairs, and `int -> int` denotes the domain of all integer functions. An expression can have several types; that is, it can belong to several domains. For example, the identity function `fn x => x` has type `int -> int`, because it maps any expression of type integer to itself, but it also has the type `bool -> bool` for a similar reason. The most general polymorphic type for the identity function is `'a -> 'a`, because all the types of identity are instances of it. This last notation gives more information than the others, because it encompasses all the types that the identity function can have and thus expresses all the ways that the identity function can be used. Hence it is preferable to the others, although the others are not wrong. The ML type checker always determines the most general type for an expression, given the information contained in that expression.

The programmer may append a type expression to a data expression in order to indicate a **type constraint**, as in Figure 3.44.

Figure 3.44

```
in:  3 : int;
out: 3 : int

in:  [(3,4), (5,6) : int * int];
out: [(3,4),(5,6)] : (int * int) list
```

In this example, the type constraint has no effect. The compiler independently infers the types and checks them against the given constraints. Any attempt to constrain a type incorrectly will result in a type error, as shown in Figure 3.45.

Figure 3.45

```
in:  3 : bool;
out: Type clash in: 3 : bool
     Looking for a: bool
     I have found a: int
```

However, a type constraint can restrict the types inferred by ML by constraining polymorphic expressions or functions, as in Figure 3.46.

Figure 3.46

```
in:  [] : int list;
out: [] : int list

in:  (fn x => x) : int -> int;
out: fn : int -> int
```

The type normally inferred for [] is 'a **list**, and for **fn** x => x, it is 'a -> 'a.

Type constraints can be used in declarations, as in Figure 3.47.

Figure 3.47

```
in:  val (a : int) = 3;
in:  val f = fn (a : int, b : int) => a+b;
in:  val f = fn (a : int, b) => a+b;
in:  val f = fn ((a,b) : int * int) => (a + b) : int;
```

The examples in lines 2, 3, and 4 are equivalent.

7.10 Constructed Types

A **constructed type** is a type for which constructors are available. Constructors can be used in patterns later to decompose data. You have already seen examples of this dual usage with the tuple constructor and the list constructors nil and :: (cons).

A constructed type and its constructors should be considered as a single conceptual unit. Whenever a new constructed type is declared, its constructors are declared at the same time. Wherever a constructed type is known, its constructors are also known.

The programmer can introduce new constructed types in a type declaration. A type declaration introduces a new type name and the names of the constructors for that type. Each of those constructors leads to a component, whose type is also presented. The components together make up a choice type, that is, a type whose values cover all the components. Syntactically, components are separated by | . Each component starts with its constructor name, followed by the keyword **of** and then the type of the component. The keyword **of** and the component type can be omitted; in this case the constructor is a constant of the new type.

For example, money can be a coin of some value (in cents), a bill of some value (in dollars), a check drawn on some bank for some amount (in cents), or the absence of money (see Figure 3.48).

Figure 3.48

```
in:  datatype money =
          nomoney |
          coin of int |
          bill of int |
          check of string * int; -- (bank, cents)
```

```
out: datatype money =
          bill of int |
          check of string * int |
          coin of int |
          nomoney
     con nomoney : money
     con coin = fn int -> money
     con check = fn : (string * int) -> money
     con bill = fn : int -> money
```

Here nomoney, coin, bill, and check are money constructors; nomoney is also a money constant. Constructors can be used as ordinary functions in expressions, as in Figure 3.49.

Figure 3.49

```
in:  val
          nickel = coin 5 and
          dime = coin 10 and
          quarter = coin 25;
out: val
          quarter = coin 25 : money
          dime = coin 10 : money
          nickel = coin 5 : money
```

Figure 3.50 shows that they can also be used in patterns.

Figure 3.50

```
in:  val amount =                                   1
          fn nomoney => 0                           2
          | (coin cents) => cents                   3
          | (bill dollars) => 100 * dollars         4
          | (check(bank,cents)) => cents;           5
out: val amount = fn : money -> int                 6
```

Quarter is not a constructor, but an identifier with value coin 25 of type money. I cannot add, say after line 4, a clause saying quarter => 25, because quarter would be interpreted as a formal parameter, like cents.

A constructed type can be made entirely of constants, in which case it is similar to an enumeration type, except there is no ordering relation among the individual constants. A type can be composed of a single constructor, in which case the type declaration can be considered as an abbreviation for the type following **of**. Both these possibilities are shown in Figure 3.51.

Figure 3.51

```
in:  datatype color = red | blue | yellow;                          1
out: datatype color = blue | red | yellow                           2
     con yellow : color                                             3
     con red : color                                                4
     con blue : color                                               5

in:  datatype point = point of int * int;                           6
out: datatype point = point of int * int                            7
     con point = fn : (int * int) -> point                          8
```

In the second example, I have overloaded the identifier `point`, which is both the name of a type and a constructor that builds values of that type. Such overloading is conventional in ML if there is only one constructor for a type. There is no risk of ambiguity, since ML can always tell by context if a constructor or a type is intended.

A constructed-type declaration may involve type identifiers, in which case the constructed type is polymorphic. All the type identifiers used on the right side of the declaration must be listed on the left side as type parameters, as shown in Figure 3.52.

Figure 3.52

```
in:  datatype 'a predicate =                                        1
          predicate of 'a -> bool;                                  2
out: datatype 'a predicate = predicate of 'a -> bool                3
     con predicate = fn : ('a -> bool) -> ('a predicate)            4

in:  predicate null;                                                5
out: predicate null : ('a list) predicate;                          6

in:  datatype ('a,'b) leftProjection =                              7
          leftProjection of ('a * 'b) -> 'a;                        8
out: datatype ('a,'b) leftProjection =                              9
          leftProjection of ('a * 'b) -> 'a                         10
     con leftProjection = fn :                                      11
          (('a * 'b) -> 'a) -> ('a,'b) leftProjection               12
```

In lines 1–2, `predicate` is declared as a type with one constructor, also called `predicate`. This constructor turns Boolean-valued functions into objects of type `predicate`. An example is shown in line 5, which applies the constructor to `null`, which is a Boolean-valued function. The result, shown in line 6, is in fact a `predicate`, with the polymorphic type somewhat constrained to `'a` **`list`**. In lines 7–8, `leftProjection` is declared as a type with one constructor, also called `leftProjection`. This type is doubly polymorphic: it depends on two type parameters. This constructor turns functions of type `('a * 'b) -> 'a` into objects of type `leftProjection`.

ML also allows recursive constructed types. Figure 3.53 shows how the predeclared list type and the hd selector are declared:

Figure 3.53

```
in:  datatype 'a list =                                  1
           nil |                                         2
           :: of 'a * ('a list);                         3
out: datatype 'a list =                                  4
           nil |                                         5
           :: of 'a * ('a list)                          6
     con nil : 'a list                                   7
     con :: = fn : ('a * ('a list)) -> ('a list)         8

in:  val hd = fn ::(head, rest) => head;                 9
out: val hd = fn : ('a list) -> 'a                      10
```

The pattern in line 9 indicates that hd may only be called on lists constructed with the `::` constructor; it is invalid to call it on `nil`.

In addition to constructed types, ML provides abstract data types through a module mechanism. It permits the specification and the implementation parts to be separated. Modules can be parameterized by types, in much the same way as generic modules in Ada and C++.

Before leaving the subject of types, I will turn briefly to two other programming languages that are closely related to ML but show how one might extend its treatment of types.

8 ◆ MIRANDA

The Miranda language, designed by David Turner of the University of Kent, shares many features with ML [Turner 85a, 86; Thompson 86]. It is strongly typed, infers types from context, provides for abstract data types, and has higher-order functions. It provides tuples and homogeneous lists, and components of tuples are extracted by patterns. Operators are provided for cons and append, as well as for list length, selection from a list by position, and set difference.

Miranda differs from ML in some minor ways. It is purely functional; there are no pointer types. Functions of more than one parameter are automatically curried unless parentheses explicitly indicate a tuple. (ML also has a declaration form that automatically curries, but I have not shown it.) The scope rules in Miranda are dynamic, which means that functions may be referenced textually before they are declared. All declarations implicitly allow recursion; there is no need for a **rec** keyword. Binary operators may be passed as actual parameters in Miranda; they are equivalent to curried functions that take two parameters.

Miranda has a nontraditional syntax in which indentation indicates grouping and conditionals look like the one in Figure 3.54.

Figure 3.54

```
max  = a, a>=b
     = b, otherwise
```

However, my examples will follow ML syntax (modified as necessary) for consistency.

Miranda provides some novel extensions to ML. First, evaluation is normally lazy. I discuss **lazy evaluation** in detail in Chapter 4; for now, let me just say that expressions, particularly actual parameters, are not evaluated until they must be, and then only as much as necessary. As Figure 3.55 shows, I can declare a function cond that does not need to evaluate all its parameters.

Figure 3.55

```
in:  val cond =                                          1
          fn true, x, y => x                             2
          | false, x, y => y;                            3
out: val cond = fn : bool -> ('a -> ('a -> 'a))          4

in:  let val x=0 in cond x=0 0 1/x end                   5
out: 0 : int                                             6
```

If cond evaluated all its parameters, the invocation in line 5 would generate an exception as the program tries to divide 1 by 0. However, lazy evaluation prevents the suspicious parameter from being evaluated until it is needed, and it is never needed.

Miranda provides a concise syntax for specifying lists by enumerating their components. Most simply, one can build a list by a shorthand, as in Figure 3.56.

Figure 3.56

```
in:  [1..10];                                            1
out: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] : int list         2
```

Infinite lists (Figure 3.57) are a bit more sophisticated.

Figure 3.57

```
in:  [0..];                                              1
out: [0, 1, 2, ...] : int list                           2

in:  val ones = 1 :: ones;                               3
out: [1, 1, 1, ...] : int list                           4
```

Line 3 declares ones recursively. I have arbitrarily decided to let the expression printer evaluate only the first three components of an infinite list.

The next step is to filter objects, whether finite or infinite, to restrict values. ZF-expressions (named after Zermelo and Fraenkel, founders of modern set theory), also called list comprehensions, are built out of filters, as shown in Figure 3.58.

Figure 3.58

```
in:  [n*n | n <- [1..5] ];                                  1
out: [1, 4, 9, 16, 25] : int list                           2

in:  [ (a,b,c,n) | a,b,c,n <- [3..]; a^n + b^n = c^n ];     3
out: [ ... ] : (int * int * int * int) list                 4

in:  val QuickSort =                                        5
     fn [] => []                                            6
     | (a :: rest) =>                                       7
          QuickSort [ b | b <- rest; b <= a ] @             8
          [a] @                                             9
          QuickSort [ b | b <- rest; b > a];                10
out: val QuickSort = fn : 'a list -> 'a list                11
```

Line 1 evaluates to a list of 5 squares. Line 3 evaluates to an empty list (most likely), but will take forever to compute. However, if the expression is evaluated lazily, the infinite computation need not even start. Lines 5–10 represent the Quicksort algorithm concisely.

Infinite lists can be used to create lookup tables for caching the values of a function. Caching allows a programmer to use a recursive algorithm but apply caching (also called dynamic programming and memoization) to change an exponential-time algorithm into a linear-time one. For example, Fibonacci numbers can be computed efficiently as shown in Figure 3.59.

Figure 3.59

```
in:  val map =                                              1
          fn function, [] => []                             2
          | function, [a :: rest] =>                        3
               (function a) :: (map function rest);         4
out: val map = fn : ('a -> 'b) -> ('a list -> 'b list)     5

in:  val cache = map fib [0..]                              6
     and fib =                                              7
          fn 0 = 1                                          8
          | 1 => 1                                          9
          | n => cache at (n-1) + cache at (n-2)            10
out: val fib = fn : int -> int                              11
```

The map function (lines 1–4) applies a function to each member of a list, producing a new list. The fib function (lines 7–10) uses the infinite object cache (line 6), which is not evaluated until necessary. Line 10 calls for evaluating just those elements of cache that are needed. (The **at** operator selects an element from a list on the basis of its position.) The chart in Figure 3.60 shows the order of events in evaluating fib 4.

Figure 3.60

```
fib 4
     cache at 3
          cache at 0
               fib 0 returns 1; cache at 0 becomes 1
          cache at 1
               fib 1 returns 1; cache at 1 becomes 1
          cache at 2
               fib 2
                    cache at 1 returns 1
                    cache at 0 returns 1
                    fib 2 returns 2; cache at 2 becomes 2
          fib 3
               cache at 2 returns 2
               cache at 1 returns 1
               returns 3; cache at 3 becomes 3
     cache at 2 returns 2
     returns 5
```

Another way to express dynamic programming for computing Fibonacci numbers is described in Chapter 9 in the section on mathematics languages.

Lazy evaluation also makes it fairly easy to generate an infinite binary tree with 7 at each node, as in Figure 3.61.

Figure 3.61

```
in:  datatype 'a tree =
          nil |
          node of 'a * ('a tree) * ('a tree);
out: datatype 'a tree =
          nil |
          node of 'a * ('a tree) * ('a tree)
          con nil : 'a tree
          con node = fn : 'a ->
               (('a tree) -> (('a tree) -> ('a tree)))

in:  val BigTree = node 7 BigTree BigTree;
out: node 7 ... ... : int tree
```

In Miranda, the programmer may introduce a named polymorphic type much like ML's **datatype** construct but without specifying constructors, as Figure 3.62 shows.

Figure 3.62

```
in:  type 'a BinOp = 'a -> ('a -> 'a);
out: type 'a BinOp = 'a -> ('a -> 'a)

in:  BinOp int;
out: int -> (int -> int)

in:  type 'a Matrix = 'a list list;
out: type 'a Matrix = 'a list list
```

```
in:  type BoolMatrix = Matrix bool;                           7
out: type BoolMatrix = bool list list                         8

in:  val AMatrix = [[true, false] [false, false]]             9
          : BoolMatrix;                                      10
out: val AMatrix = [[true, false] [false, false]]            11
          : bool list list                                   12

in:  val FirstRow = fn                                       13
          [RowOne :: OtherRows] : BoolMatrix => RowOne;      14
out: val FirstRow = fn :                                     15
          bool list list -> bool list                        16
```

In line 1, BinOp is declared as a polymorphic type with one type parameter, 'a. Line 3 demonstrates that BinOp can be invoked with a parameter int, leading to the type int -> (int -> int). Types derived from polymorphic types may be used to constrain declarations, as seen trivially in lines 9–10 and not so trivially in lines 13–14.

Recursively defined types sometimes need to provide multiple ways of deriving the same object. For example, if I wish to declare integers as a recursive data type with constructors succ and pred, I need to indicate that zero is the same as succ(pred zero). Miranda allows the programmer to specify simplification laws, as shown in Figure 3.63.

Figure 3.63

```
in:  datatype MyInt =
          zero |
          pred of MyInt |
          succ of MyInt
     laws
          pred(succ n) => n and succ(pred n) => n;

in:  pred(pred(succ(zero));
out: pred zero : MyInt
```

Simplification laws also allow the programmer to declare a rational-number data type that stores numbers in their canonical form (see Figure 3.64).

Figure 3.64

```
in:  datatype Rational = ratio of num * num

     laws ratio (a,b) =>
          if b = 0 then
               error "zero denominator"
          elsif b < 0 then
               ratio (-a,-b)
          else                                             7
               let                                         8
                    val gcd = fn (x,y) =>                  9
                         if a < b then gcd (a,b-a)         10
                         elsif b < a then gcd (a-b,b)      11
                         else a;                           12
                    CommonPart = gcd (abs a, abs b)        13
               in                                          14
                    if CommonPart > 1 then                 15
                         ratio (a div CommonPart,          16
                         b div CommonPart);                17
                    else                                   18
                         nosimplify                        19
               end;                                        20

in:  ratio (3,2);                                          21
out: ratio (3,2) : Rational                                22

in:  ratio (12,-3);                                        23
out: ratio (-4,1) : Rational                               24
```

In line 19, **nosimplify** indicates that no law applies in that case.

9 ◆ RUSSELL

The Russell language predates ML but is quite similar in general flavor [Demers 79; Boehm 86]. It was developed to explore the semantics of types, in particular, to try to make types first-class values. Russell is strongly typed, infers types from context, provides for abstract data types, and has higher-order functions.

Russell differs from ML in some minor ways. Although it is statically scoped, new function declarations do not override old ones of the same name if the types differ; instead, the name becomes overloaded, and the number and type of the actual parameters are used to distinguish which function is meant in any particular context. (Redeclaration of identifiers other than functions is not allowed at all.) Functions may be declared to be invoked as prefix, suffix, or infix operators. Functions that take more than two parameters may still be declared to be invoked with an infix operator; a given number of the parameters are placed before the operator, and the rest after. ML only allows infix notation for binary functions. To prevent side effects in the presence of variables (**ref** types), functions do not import identifiers mapped to variables.

Russell's nomenclature is nonstandard; what ML calls a type is a signature in Russell; an abstract data type (a collection of functions) is a type in Russell. So when Russell succeeds in making types first-class values, it doesn't accomplish quite as much as we would expect. Russell's syntax is quite different from ML. For consistency, I will continue to use ML terminology and syntax as I discuss Russell.

The principal difference between Russell and ML is that in Russell abstract data types are first-class values, just like values, pointers, and functions. That is, abstract data types may be passed as parameters, returned from functions, and stored in identifiers. Abstract data type values can also be manipulated after they have been constructed.

More specifically, Russell considers an abstract data type to be a collection of functions that may be applied to objects of a particular domain. The `Boolean` abstract data type includes the nullary functions `true` and `false`, binary operators such as **and** and **or**, and even statements such as **if** and **while**, which have Boolean components. Manipulation of an abstract data type means deleting or inserting functions in its definition.

The border between data and program becomes quite blurred if we look at the world this way. After all, we are not used to treating control constructs like **while** as functions that take two parameters, a Boolean and a statement, and return a statement. We don't usually consider a statement to be data at all, since it cannot be read, written, or manipulated.[12]

The components of an abstract data type may be quite different from each other. I could declare an abstract data type MyType that includes the Boolean **false** as well as the integer 3 (both nullary functions). If I wish to distinguish which false is meant, I can qualify it by saying bool.false or MyType.false. (The . operator is a selector that extracts a given component of a given abstract data type.)

I might declare a simple abstract data type of small integers as shown in Figure 3.65.

Figure 3.65

```
val SmallInt =                                                    1
    type New = fn : void -> SmallInt -- constructor              2
    and ":=" = fn : (SmallInt ref, SmallInt) -> SmallInt          3
        -- assignment                                             4
    and ValueOf = fn : SmallInt ref -> SmallInt -- deref          5
    and alias = fn : (SmallInt ref, SmallInt ref) -> bool         6
        -- pointer equality                                       7
    and "<" = fn : (SmallInt,SmallInt) -> Boolean                 8
    ... -- other comparisons, such as <= , =, >, >=, ≠            9
    and "-" = fn : (SmallInt, SmallInt) -> SmallInt              10
    ... -- other arithmetic, such as +, *, div, mod              11
    and "0" : SmallInt -- constant                               12
    ... -- other constants 1, 2, ... , 9                         13
        -- the rest are built by concatenation                   14
    and "^" = fn : (SmallInt,SmallInt) -> SmallInt               15
        -- concatenation                                         16
    ;                                                            17
```

I use void in line 2 to indicate that the New function is nullary. The function declarations are all missing their implementations.

Generally, one builds abstract data types with shorthand forms that expand out to such lists. For example, there are shorthands for declaring enumerations, records, choices, and new copies of existing abstract data types. The lists generated by the shorthands contain functions with predefined bodies.

Since abstract data types can be passed as parameters, the programmer can build polymorphic functions that behave differently on values of different abstract data types. It is common to pass both value-containing parameters and type-containing parameters to functions. Figure 3.66 shows how to declare a polymorphic Boolean function least that tells if a given value is the smallest in its abstract data type.

[12] Some languages, like SNOBOL and APL, let strings be converted at runtime into statements and then executed. Only LISP, discussed in Chapter 4, and Tcl, discussed in Chapter 9, actually build programs out of the same stuff as data.

Figure 3.66

```
val least =                                                 1
     fn (value : bool, bool) => value = false               2
     | (value : SmallInt, SmallInt) => value = SmallInt."0" 3
     | (value : Other, Other : type) => false;              4
```

Line 2 applies when the first parameter is Boolean and the second parameter so indicates. It returns `true` only if the first parameter has value `false`. Line 3 applies when the first parameter is of type `SmallInt`. Line 4 applies to all other types, so long as the type of the first parameter matches the value of the second parameter. A call such as `least("string", int)` would fail because none of the alternatives would match.

Manipulations on an abstract data type include adding, replacing, and deleting its functions. The programmer must provide a body for all replacement functions. For example, I can build a version of the `integer` type that counts how many times an assignment has been made on its values (Figure 3.67).

Figure 3.67

```
val InstrumentedInt =
     record (Value : int, Count : int)
          -- "record" expands to a list of functions
     adding
          Alloc = fn void =>
               let
                    val x = InstrumentedInt.new
               in
                    count x := 0;
                    x -- returned from Alloc
               end
     and
          Assign = fn
               (IIVar : InstrumentedInt ref,
                    IIValue : InstrumentedInt) ->
               (    -- sequence of several statements
                    count IIValue := count IIValue + 1;
                    Value IIVar := Value IIValue;
               )
     and
          GetCount = fn (IIValue : InstrumentedInt) ->
               count IIValue
     and
          new = InstrumentedInt.Alloc -- new name
     and
          ":=" = InstrumentedInt.Assign -- new name
     and
          ValueOf = ValueOf Value
     hiding
          Alloc, Assign, -- internal functions
          Value, Count, -- fields (also functions);
```

Two abstract data types are considered to have the same type if they contain the same function names (in any order) with equivalent parameter and result types. This definition is a lax form of structural equivalence.

10 ◆ DYNAMIC TYPING IN STATICALLY TYPED LANGUAGES

It seems strange to include dynamic typing in otherwise statically typed languages, but there are situations in which the types of objects cannot be predicted at compile time. In fact, there are situations in which a program may wish to create a new type during its computation.

An elegant proposal for escaping from static types is to introduce a predeclared type named `dynamic` [Abadi 91]. This method is used extensively in Amber [Cardelli 86]. Values of this type are constructed by the polymorphic predeclared function `makeDynamic`. They are implemented as a pair containing a value and a type description, as shown in Figure 3.68 (in an ML-like syntax).

Figure 3.68

```
val A = makeDynamic 3;
val B = makeDynamic "a string";
val C = makeDynamic A;
```

The value placed in `A` is 3, and its type description is `int`. The value placed in `B` is `"a string"`, and its type description is `string`. The value placed in `C` is the pair representing `A`, and its type description is `dynamic`.

Values of dynamic type can be manipulated inside a **typecase** expression that distinguishes the underlying types and assigns local names to the component values, as in Figure 3.69.

Figure 3.69

```
val rec Stringify = fn Arg : dynamic =>                         1
     typecase Arg                                               2
     of   s : string => '"' + s + '"'                           3
     |    i : int => integerToString(i)                         4
     |    f : ’a -> ’b => "function"                            5
     |    (x, y) => "(" + (Stringify makeDynamic x) +           6
               ", " + (Stringify makeDynamic y) + ")"           7
     |    d : dynamic => Stringify d                            8
     |    _ => "unknown";                                       9
```

`Stringify` is a function that takes a dynamic-typed parameter `Arg` and returns a string version of that parameter. It distinguishes the possible types of `Arg` in a **typecase** expression with patterns both to capture the type and to assign local identifiers to the components of the type. If the underlying type is itself `dynamic`, `Stringify` recurses down to the underlying type (line 8). In lines 6–7, `makeDynamic` is invoked to ensure that the parameters to `Stringify` are of the right type, that is, `dynamic`.

Figure 3.70 shows a more complicated example that nests **typecase** expressions. The function `Apply` takes two curried dynamic parameters and invokes the first one with the second one as a parameter, checking that such an application is valid.

Figure 3.70

```
val rec Apply =                                            1
     fn Function : dynamic =>                              2
          fn Parameter : dynamic =>                        3
               typecase Function                           4
               of f : 'a -> 'b =>                          5
                    typecase Parameter                     6
                    of p : 'a => makeDynamic f(p);         7
```

Line 5 explicitly binds the type identifiers `'a` and `'b` so that `'a` can be used later in line 7 when the program checks for type equivalence. Line 7 needs to invoke `makeDynamic` so that the return value of `Apply` (namely, `dynamic`) is known to the compiler. In each **typecase** expression, if the actual type at runtime is not matched by the guard, a type error has occurred. I could use an explicit **raise** statement in a language with exception handling.

The `dynamic` type does not violate strong typing. The compiler still knows the type of every value, because all the otherwise unknown types are lumped together as the `dynamic` type. Runtime type checking is needed only in evaluating the guards of a **typecase** expression. Within each branch, types are again statically known.

It is possible to allow compile-time coercion of `dynamic` types. If a `dynamic` value is used in a context where the compiler does not have any applicable meaning, it may implicitly supply a **typecase** that distinguishes the meanings that it knows how to handle, as shown in Figure 3.71.

Figure 3.71

```
in:  write makeDynamic (4 + makeDynamic 6)                 1
out: 10 : int                                              2
```

In line 1, the `+` operator has no overloaded meaning for integers plus dynamic values. The compiler realizes this fact and inserts an explicit **typecase** to handle the one meaning it knows, integers plus integers. The predeclared `write` function cannot handle `dynamic` types, either, so another **typecase** is inserted for all the types that it can handle. In other words, the input is expanded to that shown in Figure 3.72.

Figure 3.72

```
typecase makeDynamic 4 +                                   1
     typecase makeDynamic 6                                2
     of i : int => i                                       3
     end;                                                  4
of                                                         5
     i : int => write i                                    6
     r : real => write r                                   7
     ...                                                   8
end;                                                       9
```

The **typecase** expression in lines 2–4 has type `int`, so the `+` in line 1 is well defined. In order to give `write`'s parameter a compile-time type, I had to draw the `write` function into the outer **typecase** (in lines 6–8). Drawing functions into the implicit **typecase** expressions can lead to an explosion of code.

It is much better to coerce at runtime, when the actual type is known for each dynamic type. The program in Figure 3.72 would clearly use integer addition and printing of integers. Runtime coercion is still perfectly type-safe, although some type errors won't be discovered until runtime.

11 ◆ FINAL COMMENTS

The discussion of derived types and dimensions is part of a larger issue about how restrictive a programming language needs to be in order to permit the art of programming. One way to look at this question [Gauthier 92] is to notice that on the one hand the real world is very restrictively typed, as students of physics realize. One should not add apples and oranges, much less volts and calories. On the other hand, the memory of most computers is completely untyped; everything is represented by bits (organized into equally untyped bytes or words). The programming language represents a platform for describing the real world via the computer, so it properly lies somewhere between these extremes. It needs to balance type security with simplicity. Type security demands that each different kind of value have its own type in order to match the real world. For example, lists of exactly three elements are different from lists of four elements. Integers constrained to even numbers are different from unconstrained integers. Simplicity demands that types be easy to specify and that types be efficiently checked, preferably at compile time. It is not so easy to include lengths or number-theoretic considerations in the type description of lists and integers, respectively.

It is largely a matter of personal taste where this platform should be on the spectrum ranging from restrictively typed, using strong typing and perhaps providing derived types with dimensions, to lax, with dynamic typing and easy coercion. Proponents of the restrictive style point with pride to the clarity of their programs and the fact that sometimes they run correctly the first time. Proponents of the lax style speak disparagingly of "bondage-and-discipline" languages like Ada, and prefer the relative freedom of C.

Such taste is likely to change as a programmer changes. My first experience of programming (after plug-board computers) was in machine language, not even assembler language. Later, I relished the intricacies of SNOBOL, which is quite lax about typing. Algol was a real eye-opener, with its declared types and its control structures. I now prefer strong typing; to me, an elegant program is one that is readable the first time by a novice, not one that plays unexpected tricks. Strong typing helps me to build such programs. Still, I use C heavily because it is implemented so widely, and I often need to port my programs across machines.

ML is an elegant language that shows how to make functions first-class values and how to deal with type polymorphism and still be strongly typed. Type inference relieves the programmer of careful type declarations. Miranda extends these ideas with infinite lists and lazy evaluation. (There is also a lazy variant of ML with similar extensions.) Russell even allows some types to be manipulated in fairly simple ways. None of these languages truly allows types themselves to be first-class values. Such an extension would probably require runtime type checking or lose strong typing. (The exercises explore this concept.)

Type systems are an area of active research. Integrating dimensions into polymorphic languages like ML, for example, is being studied [Kennedy 94]. The SML variant of ML includes an experimental, higher-order extension of the module system, in which generic modules can be parameterized by other (possibly generic) modules.

Although I have intentionally avoided issues of syntax in this chapter, I would like to point out that syntactic design certainly affects the ease with which programmers can learn and use a language. Compare, for example, identical types in C and ML:

C	ML
`int z`	`z : int`
`int (*a)(char)`	`a : (char -> int)` **`ref`**
`int (*((*b)(int)))(char)`	`b : (int -> ((char -> int)` **`ref`**`))` **`ref`**
`int (*c)(int (*)(char))`	`c : ((char -> int)` **`ref`** `-> int)` **`ref`**

Although the C type expressions are shorter, I find them difficult to generate and to understand.

EXERCISES

Review Exercises

3.1 What is the difference between the way Modula-2+ and Modula-3 handle type equivalence for derived types?

3.2 If two types are name-equivalent, are they necessarily structurally equivalent?

3.3 Would you consider `First` and `Second` in Figure 3.73 structurally equivalent? Why or why not?

Figure 3.73

```
type
     First =
          record
               A : integer;
               B : record
                    B1, B2 : integer;
               end;
          end;
     Second =
          record
               A: record
                    A1, A2 : integer;
               end;
               B : integer;
```

```
        end;
```

3.4 How can abstract data types be used to implement dimensions?

3.5 Why is instantiation of a generic module a compile-time operation, not a runtime operation?

3.6 What sort of type equivalence does ML use — name equivalence, structural equivalence, or something else?

Challenge Exercises

3.7 Enumerate the possible values of type `TA` in Figure 3.3 (page 57).

3.8 Given that `First` and `Second` are not structurally equivalent in exercise 3.3, suggest an algorithm for testing structural equivalence.

3.9 Suggest an algorithm for compile-time dimension checking. Is runtime dimension checking needed?

3.10 Explore adding dimensions to ML.

3.11 Write an `accumulate` procedure in ML that can be used to sum a list, as suggested on page 75.

3.12 Show a valid use of `leftProjection`, introduced in Figure 3.52 (page 88).

3.13 Program QuickSort in ML.

3.14 Show how **`datatype`** in ML could give me the effect of `f1 : ((int` **`alt`** `bool) -> 'a) -> ('a * 'a)`, as suggested on page 82.

3.15 Use the `dynamic` type in an ML framework to declare a function `BuildDeep` such that `BuildDeep 2` produces a function of dynamic type `int -> (int -> int)`, `BuildDeep 3` produces a function of dynamic type `int -> (int -> (int -> int))`, and so forth. The produced functions should return the sum of all their parameters.

3.16 Generalize types in ML so that types are true first-class values. That is, I should be able to build things of type `type`, or of type `type -> int`. Decide what the built-in functions on type `type` should be. Try to keep the language strongly typed.

3.17 Extend ML so that there is a type **`expression`**. Devise reasonable functions that use that type. These functions should have runtime (not just compile-time) significance.

3.18 What is the type (in the ML sense) of `least` in Figure 3.66 (page 96)?

3.19 Can Io constructs (see Chapter 2) be represented in ML?

3.20 Show two types in Russell that are type-equivalent, but are neither name-equivalent nor structurally equivalent.

3.21 Are all structurally equivalent types type-equivalent in Russell?

3.22 Russell prevents side effects in the presence of variables (**`ref`** types) by prohibiting functions from importing identifiers mapped to variables. Why is this rule important?

3.23 Russell also prohibits a block from exporting a value of a type declared locally in that block. Why is this rule important in Russell? Should ML also have such a rule? Can this rule be enforced at compile time?