9

Data Abstraction and Modularity

Computer programmers have long recognized the value of building software systems that consist of a number of program modules. In an effective design, each module can be designed and tested independently. Two important goals in modularity are to allow one module to be written with little knowledge of the code in another module and to allow a module to be redesigned and reimplemented without modifying other parts of the system. Modern programming languages and software development environments support modularity in different ways.

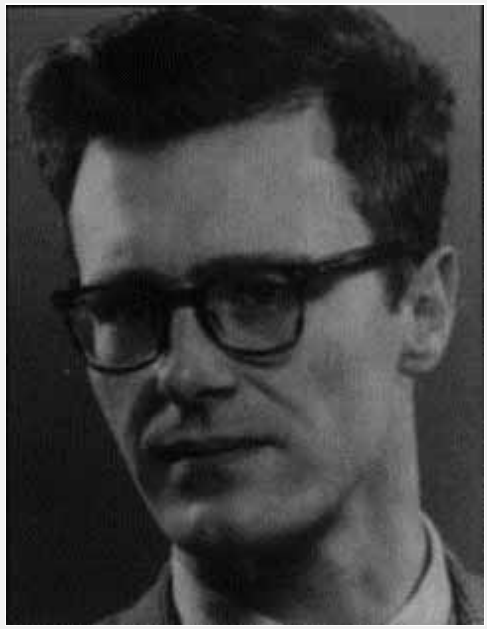
In this chapter, we look at some of the ways that programs can be divided into meaningful parts and the way that programming languages can be designed to support these divisions. Because in Chapters 10–13 we explore object-oriented languages in detail, in this chapter we are concerned with modularity mechanisms that do not involve objects. The main topics are structured programming, support for abstraction, and modules. The two examples used to describe module systems and generic programming are the standard ML module system and the C++ Standard Template Library (STL).

9.1 STRUCTURED PROGRAMMING

In an influential 1969 paper called Structured Programming, E.W. Dijkstra argued that one should develop a program by first outlining the major tasks that it should perform and then successively refining these tasks into smaller subtasks, until a level is reached at which each remaining task can be expressed easily by basic operations. This produces subproblems that are small enough to be understood and separate enough to be solved independently.

In Example 9.1, the data structures passed between separate parts of the program are simple and straightforward. This makes it possible to identify the main data structures early in the process. Because the data structures remain invariant through most of the design process, Dijkstra's example centers on refinement of procedures into smaller procedures. In more complex systems, it is necessary to refine data structures as well as procedures. This is illustrated in Example 9.2.

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press



EDSGER W DIJKSTRA

An exacting and fundamentally warm-hearted person, Edsger W. Dijkstra has made many important contributions to the field of computing science. He is known for semaphores, which are commonly used for concurrency control, algorithms such as his method for finding shortest paths in graphs, his "guarded command" language, and methods for reasoning about programs.

Over the years, Dijkstra has written a series of carefully handwritten articles, known commonly as the EWDs. As of the early 2002, he had written over 1309 EWDs, scanned and available on the web. As Dijkstra now says on his web page,

> My area of interest focuses on the streamlining of the mathematical argument so as to increase our powers of reasoning, in particular, by the use of formal techniques.

His interest in streamlining mathematical argument is evident in the EWDs, each developing an elegant solution to an intriguing problem in a few pages.

Like many old-school Europeans, and unlike most Americans, Dijkstra has impeccable handwriting. In part as a joke and in part as a tribute to Dijkstra, a programming language researcher named Luca Cardelli carefully copied the handwriting from a set of EWDs and produced the EWD font. If you can find the font on the web, you can try writing short notes in Dijkstra's famous handwriting.

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

Example 9.1

Dijkstra considered the problem of computing and printing the first 1000 prime numbers. The first version of the program contains a little bit of syntax to get us thinking about writing a program. Otherwise, it just looks like an English description of the problem we want to solve.

Program 1:

```
begin
   print first thousand prime numbers
end
```

This task can now be refined into subtasks. To divide the problem in two, some data structure must be selected for passing the result of the first subtask onto the second. In Dijkstra's example, the data structure is a table, which will be filled with the first 1000 primes.

Program 2:

```
begin variable table p
  fill table p with first thousand primes
  print table p
end
```

In the next refinement, each subtask is further elaborated. One important idea in structured programming is that each subtask is considered independently. In the example at hand, the problem of filling the table with primes is independent of the problem of printing the table. Therefore, each subtask could be assigned to a different programmer, allowing the problems to be solved at the same time by different people. Even if the program were going to be written by a single person, there is an important benefit of separating a complex problem into independent subproblems. Specifically, a single person can think about only so many details at once. Dividing a task into subtasks makes it possible to think about one task at a time, reducing the number of details that must be considered at any one time.

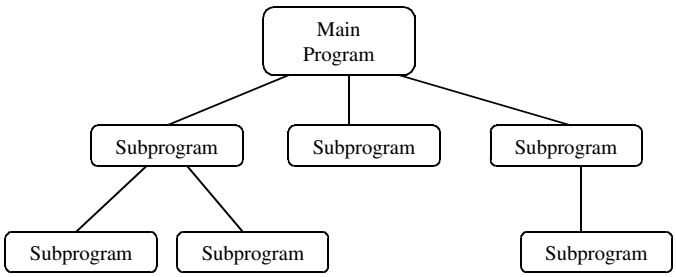
Program 3:

```
begin integer array p[1:1000]
    make for k from 1 through 1000
       p[k] equal to the kth prime number
    print p[k] for k from 1 through 1000
end
```

At this point, the basic program structure has been determined and the programmer can concentrate on the algorithm for computing successive primes. Although this example is extremely simple, it should give some idea of the basic idea of programming

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

by stepwise refinement. Stepwise refinement generally leads to programs with a tree-like conceptual structure.

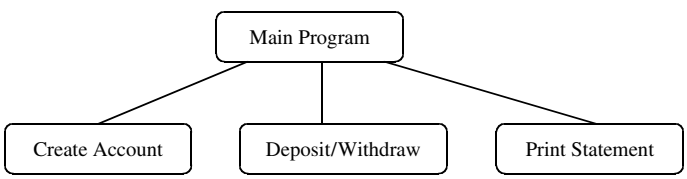


One difficult aspect of top-down program development is that it is important to make the problem simpler on each refinement step. Otherwise, it might be possible to refine a task and produce a list of programming problems that are each more difficult than the original task. This means that a designer who uses stepwise refinement must have a good idea in advance of how tasks will eventually be accomplished.

9.1.1 Data Refinement

In addition to refining tasks into simpler subtasks, evolution in a system design may lead to changes in the data structures that are used to combine the actions of independent modules.

Example 9.2
Consider the problem of designing a simple banking program. The goal of this program is to process account deposits, process withdrawals, and print monthly bank statements. In the first pass, we might formulate a system design that looks something like this:



In this design, the main program receives a list of input transactions and calls the appropriate subprograms. If we assume that statements only contain the account number and balance, then we could represent a single bank account by an integer value and store all bank accounts in a single integer array.

If we later refine the task "Print Statement" to include the subtask "Print list of transactions," then we will have to maintain a record of bank transactions. For this refinement, we will have to replace the integer array with some other data structure that records the sequence of transactions that have occurred since the last statement. This may require changes in the behavior of all the subprograms, as all of them perform operations on bank accounts.

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

9.1.2 Modularity

Divide and conquer is one of the fundamental techniques of computer science. Because software systems can be exceedingly complex, it is important to divide programs into separate parts that can be treated independently. Top-down program development, when it works, is one method for producing programs that consist of separable parts. In some cases, it is also useful to work bottom-up, designing basic parts that will be needed in a large software system and then combining them into larger subsystems. Since the 1970s, a number of other program-development methods have been proposed.

One useful development method, sometimes called *prototyping*, involves implementing parts of a program in a simple way to understand if the design will really work. Then, after the design has been tested in some way, one can improve parts of the program independently by reimplementing them. This process can be carried out incrementally by a series of progressively more elaborate prototypes to develop a satisfactory system. There are also related object-oriented design methods, which we will discuss in Chapter 10.

One important way for programming languages to support modular programming methods is by helping programmers keep track of the dependencies between different parts of a system. For the purposes of discussion, we call a meaningful part of a program that is meant to be partially independent of other parts a program *component*.

Two important concepts in modular program development are interfaces and specifications:

- *Interface:* A description of the parts of a component that are visible to other program components.
- *Specification:* A description of the behavior of a component, as observable through its interface.

When a program is designed modularly, it should be possible to change the internal structure of any component, as long as the behavior visible through the interface remains the same.

A simple example of a program component is a single function. The interface of a function consists of the function name, the number and types of parameters, and the type of the return result. A function interface is also called a *function header*.

A function specification usually describes the relationship between function arguments and the corresponding return result. If a function will work properly on only certain arguments, then this restriction should be part of the function specification. For example, the interface of a square-root function might be

```
function sqrt (float x) returns float
```

A specification for this function can be written as

```
If x>0, then sqrt(x)*sqrt(x) ≈ x.
```

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

where the squiggly approximation sign, ≈, is used to mean "approximately equal," as computation with floating-point numbers is carried out to only limited precision.

In some forms of modular programming, system designers write a specification for each component. When a component is implemented, it should be designed to work correctly when all of the components it interacts with satisfy their specifications. In other words, the correctness of one component should not depend on any hidden implementation details of any other component. One reason for striving to achieve this degree of independence is that it allows components to be reimplemented independently. Specifically, in a system in which each component relies on only stated specifications of other components, we can replace any component with another that satisfies the same specification. This allows us to optimize components independently or to add functionality that does not violate the original specification.

There are many different languages and methods for writing specifications, ranging from English and graphical notations that have little structure to formal languages that can be manipulated by specification tools. A basic problem associated with program specification is that there is no algorithmic method for testing that a module satisfies its specification. This is a consequence of a fundamental mathematical limitation, similar to the undecidability of the halting problem. As a result, programming with specifications requires substantial effort and discipline.

To illustrate the use of data structures and specifications, we look at a sorting algorithm that uses a general data structure that also serves other purposes.

Example 9.3 A Modular Sorting Algorithm

An integer priority queue is a data structure with three operations:

```
empty : pqueue
insert : int * pqueue ⟶ pqueue
deletemax: pqueue ⟶ int * pqueue
```

In words, there is a way of representing an empty priority queue, an insert operation that adds an integer to a priority queue, and a deletemax operation that removes an element from a priority queue. These three operations form the interface to priority queues. To give more detail, we have the following specifications:

- Each priority queue has a multiset of elements. There must be an ordering ≤ on the elements that may be placed in a priority queue. (For integer priority queues, we may use the ordinary ≤ ordering on integers.)
- An empty priority queue has no elements.
- insert(elt, pq) returns a priority queue whose elements are elt plus the elements of pq.
- deletemax(pq) returns an element of pq that is ≥ all other elements of pq, together with a data structure representing the priority queue obtained when this element is removed.

These specifications do not impose any restrictions on the implementation of priority queues other than properties that are observable through the interface of priority queues.

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

Knowing in advance that we would like to use priority queues in our sorting algorithm, we can begin the top-down design process by stating the problem in a program form:

Program 1:

```
function sort
    begin
        sort an array of integers
    end
```

The next step is to refine the statement sort an array of integers into subtasks. One way to do this, using priority queues, is to transfer the elements of the array into a priority queue and then remove them one at a time. In addition, we can make the decision at this point that the function will take an array and its integer length as separate arguments.

Program 2:

```
function sort(n:int, A : array [1..n] of int)
    begin
        place each element of array A in a priority queue
        remove elements in decreasing order and place in array A
    end
```

Finally, we can translate these English descriptive statements into some form of program code. (Here, the program is written in a generic Algol- or Pascal-like notation.)

Program 3:

```
function sort(n:int, A : array [1..n] of int)
    begin priority queue s;
        s := empty;
        for i := 1 to n do s := insert(A[i], s);
        for i := n downto 1 do (A[i],s) := deletemax(s);
end
```

One advantage of this sorting algorithm is that there is a clear separation between the control structure of the algorithm and the data structure for priority queues. We could implement priority queues inefficiently to begin with, by using an algorithm that is easy to code, and then optimize the implementation later if this turns out to be needed.

As written, it seems difficult to sort an array in place by this algorithm. However, it is possible to come close to the conventional heapsort algorithm.

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

9.2 LANGUAGE SUPPORT FOR ABSTRACTION

Programmers and software designers often speak about "finding the right abstraction" for a problem. This means that they are looking for general concepts, such as data structures or processing metaphors, that will make a complex, detailed problem seem more orderly or systematic. One way that a programming language can help programmers find the right abstraction is by providing a variety of ways to organize data and computation. Another way that a programming language can help with finding the right abstraction is to make it possible to build program components that capture meaningful patterns in computation.

9.2.1 Abstraction

In programming languages, an abstraction mechanism is one that emphasizes the general properties of some segment of code and hides details. Abstraction mechanisms generally involve separating a program into parts that contain certain details and parts where these details are hidden. Common terms associated with abstraction are

- *client:* the part of a program that uses program component
- *implementation:* the part of a program that defines a program component.

The interaction between the client of an abstraction and the implementation of the abstraction is usually restricted to a specific interface.

Procedural Abstraction

One of the oldest abstraction mechanisms in programming languages is the procedure or function. The client of a function is a program making a function call. The implementation of a function is the function body, which consists of the instructions that will be executed each time the function is called.

If we have a few lines of code that store the square root of a variable `x` in the variable `y`, for example, then we can *encapsulate* this code into a function. This accomplishes several things:

1. The function has a well-defined interface, made explicit in the code. The interface consists of the function name, which is used to call the function, the input parameters (and their types, if it is a typed programming language) and the type of the output.
2. If the code for computing the function value uses other variables, then these can be made local to the function. If variables are declared inside the function body, then they will not be visible to other parts of the program that use the function. In other words, no assignment or other use of local variables has any effect on other parts of the program. This provides a form of *information hiding*: information about how the function computes a result is contained in the function declaration, but hidden from the program that uses the function.
3. The function may be called on many different arguments. If code to carry out a computation is written in-line, then the computation is performed on specific variables. By enclosing the code in a function declaration, we obtain an *abstract*

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

entity that makes sense apart from its specific use on these specific variables. In grandiose terms, enclosing code inside a function makes the code generic and reusable.

This is an idealistic description of the advantages of enclosing code inside a function. In most programming languages, a function may read or assign to global variables. These global variables are not listed in the function interface. Therefore, the behavior of a function is not always determined by its interface alone. For this reason, some purists in program design recommend against using global variables in functions.

Data Abstraction

Data abstraction refers to hiding information about the way that data are represented. Common language mechanisms for data abstractions are abstract data-type declarations (discussed in Subsection 9.2.2) and modules (discussed in Section 9.3).

We saw in Subsection 9.1.2 how a sorting algorithm can be defined by using a data structure called a priority queue. If a program uses priority queues, then the writer of that program must know what the operations are on priority queues and their interfaces. Therefore, the set of operations and their interfaces is called the interface of a data abstraction. In principle, a program that uses priority queues should not depend on whether priority queues are represented as binary search trees or sorted arrays. These implementation details are best hidden by an encapsulation mechanism.

As for procedural abstraction, there are three main goals of data abstraction:

1. Identifying the interface of the data structure. The interface of a data abstraction consists of the operations on the data structure and their arguments and return results.
2. Providing *information hiding* by separating implementation decisions from parts of the program that use the data structure.
3. Allowing the data structure to be used in many different ways by many different programs. This goal is best supported by generic abstractions, discussed in Section 9.4.

9.2.2 Abstract Data Types

Interest in data abstraction came to prominence in the 1970s. This led to the development of a programming language construct call the *abstract data-type* declaration.

This is a common short definition of an abstract data type:

> An *abstract data type* consists of a type together with a specified set of operations.

Good languages for programming with abstract data types not only allow a programmer to group types and operations, but also use type checking to limit access to the representation of a data structure. In other words, not only does an abstract data type have a specific interface that can be used by other parts of a program, but access is also restricted so that *the only use of an abstract data type is through its interface*.

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

If a stack is implemented with an array, then programs that use a stack abstract data type can use only the stack operations (push and pop, say), not array operations such as indexing into the array at arbitrary points. This hides information about the implementation of a data structure and allows the implementer of the data structure to make changes without affecting parts of a program that use the data structure.

We can appreciate some aspects of abstract data types by understanding a historical idea that was in the air at the time of their development. In the early 1970s, there was a movement to investigate "extensible" languages. The goal of this movement was to produce programming languages in which the programmer would be able to define constructs with the same flexibility as a language designer. For example, if some person or group of programmers wanted to write programs by using a new form of iterative loop, they could use a "loop declaration" to define one and use it in their programs. This idea turned out to be rather unsuccessful, as programs littered with all kinds of programmer-defined syntactic conventions can be extremely difficult to read or modify. However, the idea that programmers should be able to define types that have the same status as the types that are provided by the language did prove useful and has stood the test of time.

A potential confusion about abstract data types is the sense in which they are abstract. A simple distinction is that a data type whose representation and operational details are hidden from clients is abstract. In contrast, a data type whose representation details are visible to clients may be called a transparent type. ML abstype, discussed in the next subsection, defines abstract data types, whereas ML data type, discussed in Subsection 6.5.3, is a transparent type-declaration form.

9.2.3 ML abstype

We use the historical ML abstract data-type construct, called abstype, to discuss the main ideas associated with abstract data-type mechanisms in programming languages.

As discussed in the preceding subsection, an abstract data-type mechanism associates a type with a data structure in such a way that a specific set of functions has direct access to the data structure but general code in other parts of a program does not. We will see how this works in ML by considering a simple example, complex numbers.

We can represent a complex number as a pair of real numbers. The first real is the "real" part of the complex number, and the second is the "imaginary" part of the complex number. If we are going to compute with complex numbers, then we need to have a way of forming a complex number from two reals and ways of getting the real and imaginary parts of a complex number. Computation with complex numbers may also involve complex addition, multiplication, and other standard operations. Here, simply providing complex addition is discussed. Other operations could be included in the abstract data type in similar ways.

An ML declaration of an abstract data type of complex numbers may be written as follows:

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

```
abstype cmplx = C of real * real with
    fun cmplx(x,y: real) = C(x,y)
    fun x_coord(C(x,y)) = x
    fun y_coord(C(x,y)) = y
    fun add(C(x1, y1), C(x2, y2)) = C(x1+x2, y1+y2)
end
```

This declaration binds five identifiers for use outside the declaration: the type cmplx and the functions cmplx, x_coord, y_coord, and add. The declaration also binds the name C to a constructor that can be used only within the bodies of the functions that are part of the declaration. Specifically, C may appear in the code for cmplx, x_coord, y_coord, and add but not in any other part of the program.

The type name cmplx is the type of complex numbers. When a program uses complex numbers, each complex number will be represented internally as a pair of real numbers. However, because the type name cmplx is different from the ML type real*real for a pair of real numbers, a function that is meant to operate on a pair of real numbers cannot be applied to a value of type cmplx: The ML type checker will not allow this. This restriction is one of the fundamental properties of any good abstract data-type mechanism: Programs should be restricted so that only the declared operations of the abstract type can be applied.

Within the data-type declaration, however, the functions that are part of the abstract data type must be able to treat complex numbers as pairs of real numbers. Otherwise, it would not be possible to implement many operations. In ML, a constructor is used to distinguish "abstract" from "concrete" uses of complex numbers. Specifically, if z is a complex number, then matching z against the pattern C (x,y) will bind x to the real part of z and y to the imaginary part of z. This form of pattern matching is used in the implementation of complex addition, for example, in which add combines the real parts of its two arguments and the imaginary parts of its two arguments. The pair representing the complex sum is then identified as a complex number by application of the constructor C.

When ML is presented with this declaration of complex numbers, it returns the following type information:

```
type cmplx
val cmplx = fn : real * real → cmplx
val x_coord = fn : cmplx → real
val y_coord = fn : cmplx → real
val add = fn : cmplx * cmplx → cmplx
```

The first line indicates that the declaration introduces a new type, named cmplx. The next four lines list the operations allowed on expressions with type cmplx. The types of these operations involve the type cmplx, not the type real*real that is used to represent complex numbers. Be sure you understand the code for add, for example, and why the type checker gives add the type cmplx * cmplx → cmplx.

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

In general, an ML abstype declaration has the form

```
abstype t = <constructor> of <type>
    with
        val <pattern> = <body>
        ...
        fun f(<pattern>) = <body>
        ...
    end
```

The syntax

```
t = <constructor> of <type>
```

is the same notation used to define data types. The identifier t is the name of the new type, <constructor> is the name of the constructor for the new type t, and <type> gives the type used to represent elements of the abstract type. The difference between an abstype and a data type lies with the rest of the preceding syntax. The value and function declarations that occur between the with and the end keywords are the only operations that may be written with the constructor. Other parts of the program may refer to the type name t and the functions and values declared between with and end. However, other parts of the program are outside the scope of the declaration of the constructor and therefore may not convert between the abstract type and its representation. The operations declared in an abstype declaration are called the *interface* of the abstract type, and the hidden data type and associated function bodies are called its *implementation*.

As many readers know, there are two common ways of representing complex numbers. The preceding abstract type uses rectangular coordinates – each complex number is represented by a pair consisting of its real and imaginary coordinates. The other standard representation is called polar coordinates. In the polar representation, each complex number is represented by its distance from the origin and an angle indicating the direction (relative to the real axis) used to reach the point from the origin. Because the implementation of the abstract data type cmplx is hidden, a program that uses a rectangular implementation can be replaced with one that uses a polar representation without changing the behavior of any program that uses the abstract data type.

A polar representation of complex numbers is used in this abstract data-type declaration:

```
abstype cmplx = C of real * real with
    fun cmplx(x,y: real) = C(sqrt(sq(x)+sq(y)), arctan(y/x))
    fun x_coord(C(r,theta)) = r * cos(theta)
    fun y_coord(C(r,theta)) = r * sin(theta)
    fun add(C(x1,y1), C(x2,y2)) = C(..., ...)
end
```

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

where the implementation of add is filled in as appropriate.

Example 9.4 Set Abstract Type

We can also create polymorphic abstypes, as the following abstype declaration illustrates:

```
abstype 'a set = SET of 'a list with
    val empty = SET(nil)
    fun insert(x, SET(elts)) = ...
    fun union(SET(elts1), SET(elts2)) = ...
    fun isMember(x, SET(elts)) = ...
end
```

Assuming the preceding ...'s are filled in with the appropriate code to implement insert, union, and isMember, ML returns as the result of evaluating this declaration:

```
type 'a set
val empty = - : 'a set
val insert = fn : 'a * ('a set) → ('a set)
val union = fn : ('a set) * ('a set) → ('a set)
val isMember = fn : 'a * ('a set) → bool
```

Note that the value for empty is written as -, instead of nil. This hiding prevents users of the 'a set abstype from using the fact that the abstype is currently implemented as a list.

Clu Clusters

The first language with user-declared abstract types was Clu. In Clu, abstract types are declared with the cluster construct. Here is an example declaration of complex numbers:

```
complex = cluster is make_complex, real_part, imaginary_part, plus, times
        rep = struct [ re, im : real ]
        make_complex = proc (x, y : real) returns (cvt)
                return (rep${re:x, im:y})
        real_part = proc (z : cvt) returns (real)
                return (x.re)
        imaginary_part = proc (z : cvt) returns (real)
                return (x.im)
        plus = proc (z, w : cvt) returns (cvt)
                return (rep${re: z.re + w.re, z.im + w.im})
        mult = ...
end complex
```

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

BARBARA LISKOV

Barabara Liskov's research and teaching interests include programming languages, programming methodology, distributed computing, and parallel computing. She was the developer of the programming language Clu, which was described this way when it was developed in the 1970s: "The programming language Clu is a practical vehicle for study and development of approaches in structured programming. It provides a new linguistic mechanism, called a cluster, to support the use of data abstractions in program construction."

When I was a graduate student at MIT, Barbara had huge piles of papers, many three or four feet high, covering the top of her desk. Whenever I went in to talk with her, I imagined I might find her unconscious under a fallen heap of printed matter.

In this code, the line rep = struct [re, im : real] specifies that each complex number is represented by a struct with two real (float) parts, called re and im. Inside the implementations of operations of the cluster, the keywords cvt and rep are used to convert between types complex and struct [re, im : real], in the same way that pattern matching and the constructor C are used in the ML abstype declaration for cmplex.

9.2.4 Representation Independence

We can understand the significance of abstract type declarations by considering some of the properties of a typical built-in type such as int:

- We can declare variables x : int of this type.
- There is a specific set of built-in operations on the type +, -, *, etc.
- Only these built-in operations can be applied to values of type int; it is not type correct to apply string or other types of operations to integers.

Because ints can be accessed only by means of the built-in operations, they enjoy a property called *representation independence*, which means that different computer representations of integers do not affect program behavior. One computer could represent ints by using 1's complement, and another by using 2's complement, and the same program run on the two machines will produce the same output (assuming all else is equal).

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

A type has the *representation-independence* property if different (correct) underlying representations or implementations for the values of that type are indistinguishable by clients of the type. This property implies that implementations for such types may be changed without breaking any client code, a useful property for software engineering.

In a type-safe programming language with abstract data types,

- we can declare variables of an abstract type,
- we define operations on any abstract type,
- the type-checking rules guarantee that only these specified operations can be applied to values of the abstract type.

For the same reasons as those for built-in types such as int, these properties of abstract data types imply representation independence for user-defined type. Representation independence means that we can change the representation for our abstract type without affecting the clients of our abstraction.

In practice, different programming languages provide different degrees of representation independence. In Clu, ML, and other type-safe programming languages with an abstract data-type mechanism, it is possible to prove a form of representation independence as a theorem about the ideal implementation of the language. The proof of this theorem relies on the way the programming language restricts access to implementation of an abstract data type. In languages like C or C++ that have type loopholes, representation independence is an ideal that can be achieved through good programming style. More specifically, if a program uses only a specific set of operations on some data structure, then the data structure and implementations of these operations can be changed in various ways without changing the behavior of the program that uses them. However, C does not enforce representation independence. This is true for built-in types as well as for user-declared types. For example, C code that examines the bits of an integer can distinguish 1's complement from 2's complement implementations of integer operations.

9.2.5 Data-Type Induction

Data-type induction is a useful principle for reasoning about abstract data types. We are not interested here in the formal aspects of this principle, only the intuition that it provides for thinking about programming and data-type equivalence. Data-type equivalence is an important relation between abstract data types: We can replace a data type with any equivalent one without changing the behavior of any client program. This principle is used informally in program development and maintenance. In particular, it is common to first build a software system with potentially inefficient prototype implementations of a data type and then to replace these with more efficient implementations as time permits.

Partition Operations

For many data types, it is possible to partition the operations on the type into three groups:

1. *Constructors*: operations that build elements of the type.
2. *Operators*: operations that map elements of the type that are definable only

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

with constructors to other elements of the type that are definable with only constructors.

3. *Observers*: operations that return a result of some other type.

The main idea is that all elements of the data type can be defined with constructors; operators are useful for computing with elements of the type, but do not define any new values. Observers are the functions that let us distinguish one element of the data type from another. They give us a notion of *observable equality*, which is usually different from equality of representation.

Example 9.5 Equivalence of Integer Sets Implementations
For the data type of integer sets with the signature

```
empty : set
insert : int * set ⟶ set
union : set * set ⟶ set
isMember: int * set ⟶ bool
```

the operations can be partitioned as follows:

1. *Constructors*: empty and insert
2. *Operator*: union
3. *Observer*: isMember

We may understand some of the intuition behind this partitioning of the operations by thinking about how sets might be used in a program. Because there is no print operation on sets, a program cannot produce a set directly as output. Instead, if any printable output of a program depends on the value of some set expression, it can be only because of some membership test on sets. Therefore, if two sets, s_1 and s_2, have the property that

For all integers n, isMember(n,s_1) = isMember(n,s_2)

then no program will be able to distinguish one from the other in any observable way. This actually gives us a useful equivalence relation on sets: Two sets s_1 and s_2 are *equivalent* if isMember(n,s_1) = isMember(n,s_2) for every integer n. For sets, this equivalence principle is actually the usual *extensionality axiom* from set theory: Two sets are equal if they have precisely the same elements.

Given extensionality of sets, it is easy to see that any set can be defined by insertion of some number of elements into the empty set. More specifically, for every set s, there is a sequence of elements $n_1, n_2, \ldots, n_k$, with

$$s \approx \text{insert}(n_1, \text{insert}(n_2, \ldots \text{insert}(n_k, \text{empty}) \ldots)).$$

This demonstrates that insert and empty are in fact constructors for the data type of sets: Every set can be defined with only these two operations.

To formally show that a given method is an operator, we need to demonstrate that, for any given use of an operator, there exists a sequence of constructor calls that produces the same result. As we expect, union is a useful operation on sets, but if s_1 and s_2 are definable with the operations of this data type, then union(s_1,s_2) can be

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

defined with only insert and empty. For this reason, union is classified as an operator, not a constructor.

In practice, it is not always easy to partition the operations of a data type into these three groups. Some functions might appear to fit into two groups. However, the principle of data-type induction still provides a useful guide for reasoning about arbitrary abstract data types.

Induction over Constructors

Because all elements of a given abstract type are given by a sequence of constructor operations, we may prove properties of all elements of an abstract type by induction on the number of uses of constructors necessary to produce a given element. Once we show that some function of the signature is an operator, we can generally eliminate it from further consideration.

Example 9.6

As an illustration of data-type induction, we will go through the outline of a proof that two different implementations of integer sets are equivalent. The term equivalent means that, if we replace one implementation with another, then no client program can detect the change.

Let us begin with a definition of equivalence, the property we are trying to prove:

> Two implementations set and set′ are equivalent if, for all values of all parameters, all corresponding applications of observers to set expressions are equal.

We refer to the operations of set by empty, insert, union, and isMember and the operations of set′ by empty′, insert′, union′, and isMember′. Some examples of corresponding applications of observers are

```
isMember(6, insert(n1, ...insert(nk, empty) ...)) and
isMember′(6, insert′(n1, ...insert′(nk, empty′) ...))
```

These expressions correspond in the sense that all the nonset arguments are the same, but we have replaced the operations of one implementation with another.

The intuition behind this definition of data-type equivalence is similar to the equivalence relation on set expressions we previously discussed. Specifically, suppose we have two different implementations of sets. The only way a client program can use one of them to produce a printable (or observable) output is to use the set constructors and operators to build up some potentially complicated sets and then to "observe" the resulting sets by using the observer operations.

Because we have established that union is an operator, not a constructor, and the only observer function is isMember, proving the equivalence of two different implementations of sets boils down to showing that

```
for all z, isMember(z, aSet) = isMember′(z, aSet′)
```

where aSet and aSet′ are corresponding expressions in which only the constructors empty and insert (or empty′ and insert′) are used.

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

We have now reduced the problem of establishing data-type equivalence to the problem of showing that

```
isMember(n, insert(n1, ... insert(nk, empty) ...))
= isMember'(n, insert'(n1, ... insert'(nk, empty') ...))
```

for all sequences of natural numbers n, n1, . . . , nk. We can do this by induction on k, the number of insert operations required for constructing the sets.

The inductive proof proceeds as follows.

Base Case: Zero Insert Operations. In this case, we must show that

```
for all n, isMember(n, empty) = isMember'(n, empty')
```

We must do this by looking at the actual implementations of the data type. However, in a correct implementation of sets, the empty set has no elements. Therefore, if both implementations are correct, then isMember(n, empty) = isMember'(n, empty') = false.

Induction Step. We assume that equivalence holds when k insert operations are used and consider the case of k+1 insert operations. This reduces to showing that, for all n, m, we have

```
isMember(n, insert(m, s)) = isMember'(n, insert'(m, s'))
```

under the assumption that for all n we have isMember(n,s) = isMember'(n,s'). Again, we must do this by looking at the actual implementations. However, if both implementations are correct, then we should have isMember(n,s) = isMember'(n,s').

An interesting aspect of this argument is that we have proved something about all possible programs that use a data type by using only ordinary induction over the constructors. The reason this is possible is the assumption that, in a language with abstract data types, only the operations of the data type can be applied to values of the type. It would be impossible to use this form of proof if type-checking rules did not guarantee that only set operations may be applied to a set. In practice, however, the ideas illustrated here may be useful for programming in languages such as C that do not enforce data abstraction, as long as the actual programs that are built do not operate on data structures except through operations designed for this purpose.

The "proof" previously described is actually just a proof outline that assumes some properties of each implementation of sets. To understand how data-type induction really works, you may work through the equivalence proof with two specific implementations in mind. For example, you may use data-type induction to prove the equivalence of a linked-list implementation and a doubly linked-list implementation of sets.

9.3 MODULES

Early abstract data-type mechanisms, like Clu clusters, declared only one type. If you want only an abstract data type of stacks, queues, trees, or other common data structures, then this form is sufficient: In each of these examples, there is one kind of data structure that is being defined, and this can be the abstract type. However, there are situations in which it is useful to define several related structures. More

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

generally, a set of types, functions, exceptions, and other user-definable entities may be conceptually related and have implementations that depend on each other.

A *module* is a programming language construct that allows a number of declarations to be grouped together. Early forms of modules, such as in the language Modula, provide minimal information hiding. However, a good module mechanism will allow a programmer to control the visibility of items declared in a module. In addition, parameterized modules, as discussed in the next subsection and in more detail in Section 9.4, make it possible to generalize a set of declarations and instantiate them together in different ways for different purposes.

9.3.1 Modula and Ada

As mentioned briefly in Subsection 5.1.4, the Modula programming language was a descendent of Pascal, developed by Pascal designer Niklaus Wirth in Switzerland in the late 1970s. The main innovation of Modula over Pascal is a module system. We will use Modula-2, a successful version of the language, to discuss Modula modules.

The basic form for Modula-2 modules is

```
module <module_name>;
    import specifications;
    declarations;
begin
    statements;
end <module_name>.
```

The declarations may be constant, type and procedure declarations, as in Pascal. The statements are Pascal-like statements. An import specification lists another module name and lists the constants, types, and procedures used from that other module; for example,

```
from Trig import sin, cos, tan
```

It is also possible to write the module name only, importing all declarations from that module.

The basic form of the preceding module may be used as a main program, with the statements performing some task. However, the basic form does not have any parts that are visible externally. To make declarations of one module visible to another, a module interface must be given.

In Modula terminology, a module interface is called a *definition module* and an implementation an *implementation module.* An implementation module has the form given at the beginning of this section, and a definition module contains only the names and types of the parts of an implementation module that are to be visible to other modules.

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

Example 9.7 Modula-2 Definition of Fractions

```
definition module Fractions;
    type fraction = ARRAY [1 .. 2] OF INTEGER;
    procedure add (x, y : fraction) : fraction;
    procedure mul (x, y : fraction) : fraction;
end Fractions.
implementation module Fractions;
    procedure Add (x, y : Fraction) : Fraction;
        VAR temp : Fraction;
        BEGIN
            temp [1] := x [1] * y [2] + x [2] * y [1];
            temp [2] := x [2] * y [2]; RETURN temp;
    END Add;
    procedure Mul (x, y : Fraction) : Fraction;
        . . .
    END Mul;
end Fractions.
```

In this example, a complete type declaration is included in the interface. As a result, the client code can see that a fraction is an array of integers. The following example hides the implementation of a type. In Modula terminology, the type declaration in Example 9.7 is *transparent* whereas the declaration in Example 9.8 is abstract or *opaque*.

Example 9.8 Modula-2 Stack Module

```
definition module Stack_module
    type stack (* an abstract type *)
    procedure create_stack ( ) : stack
    procedure push( x:integer, var s:stack ) : stack
    . . .
end Stack_module

implementation module Stack_module
    type stack =array [1..100] of integer
    . . .
end Stack_module
```

This example code defines stacks of integers. For stacks of various kinds, we would either need to repeat this definition with other types of elements or to build a generic stack module that takes the element type as a parameter. Mechanisms for defining generic modules are included in Modula-2, Ada, and most modern languages (except Java!). As representative examples, we discuss C++ templates and ML functors in Section 9.4.

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

Ada Packages

The Ada programming language was designed in the late 1970s and early 1980s as the result of an initiative by the U.S. Department of Defense (DoD). The DoD wanted to standardize its software procurement around a common language that would provide program structuring capabilities and specific features related to real-time programming. A competitive process was used to design the language. Four teams, each assigned a color as its code name, were each funded to produce a tentative "strawman" design. One of these designs, selected by a process of elimination, eventually led to the language Ada.

By some measures, Ada has been a successful language. Many Ada programs have been written and used. Some Ada design issues led to research studies and improvements in the state of the art of programming language design. However, in spite of some practical and scientific success, adoption of the language outside of suppliers of the U.S. government has been limited. One limitation was the lack of easily available implementations. Most companies who produced Ada compilers, especially at the height of the language's popularity, expected to sell them for high prices to military contractors. As a result, the language received little acceptance in universities, research laboratories, or in companies concerned primarily with civilian rather than military markets.

Ada modules are called *packages*. Packages can be written with a separate interface, called a package specification, and implementation, called a package body. Here is a sketch of how the fraction package in Example 9.7 would look if translated into Ada:

```
package FractionPkg is
    type fraction is array ... of integer;
    procedure Add_...
end FractionPkg;
package body FractionPkg is
    procedure Add ...
end FractionPkg;
```

9.3.2 ML Modules

The standard ML module system was designed in the mid-1980s as part of a redesign and standardization effort for the ML programming language. The principal architect of the ML module system was David MacQueen, who drew on concepts from type theory as well as his experience with previous programming languages.

The three main parts of the standard ML module system are *structures*, *signatures*, and *functors*. An ML structure is a module, which is a collection of type, value, and structure declarations. Signatures are module interfaces. In standard ML, signatures behave as a form of "type" for a structure, in the sense that a module may have more than one signature and a signature may have more than one associated module. If a structure satisfies the description given in a signature, the structure "matches" the signature.

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

Functors are functions from structures to structures. Functors are used to define generic modules. Because ML does not support higher-order functors (functors taking functors as arguments or yielding functors as results), there is no need for functor signatures.

Structures are defined with structure expressions, which consist of a sequence of declarations between keywords struct and end. Structures are not "first class" in that they may only be bound to structure identifiers or passed as arguments to functors. The following declaration defines a structure with one type and one value component:

```
structure S =
    struct
        type t = int
        val x : t = 3
    end
```

In this example, the structure expression following the equal sign has type component t equal to int and value component x equal to 3. In standard ML this structure is "time stamped" when the declaration is elaborated, marking it with a unique name that distinguishes it from any other structure with the same type and value components. Structure expressions are therefore said to be "generative" because each elaboration may be thought of as "generating" a new one. The reason for making structure expressions generative is that the module language provides a form of version control based on specifying that two possibly distinct structures or types must be equal.

The components of a structure are accessed by qualified names, written in a form used for record access in many languages. For instance, given our preceding structure declaration for S, above, the name S.x refers to the x component of S, and hence has value 3. Similarly, S.t refers to the t component of S and is equivalent to the type int during type checking. In other words, type declarations in structures are transparent by default. As in Modula and Ada, the distinction between transparent and opaque type declarations appears in the interface.

ML signatures are structure interfaces and may declared as follows:

```
signature SIG =
    sig
        type t
        val x : t
    end
```

This signature describes structures that have a type component t and a value component x, whose type is the type bound to t in the structure. Because the structure S previously introduced satisfies these conditions, it is said to *match* the signature SIG. The structure S also matches the following signature SIG′:

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

```
signature SIG' =
    sig
        type t
        val x : int
    end
```

This signature is matched by any structure providing a type t and a value x of type int such as the structure S. However, there are structures that match SIG, but not SIG', namely any structure that provides a type other than int and a value of that type. In addition to ambiguities of this form, there is another, more practically motivated, reason why a given structure may match a variety of distinct signatures: Signatures may be used to provide distinct views of a structure. The main idea is that the signature may specify fewer components than are actually provided. For example, we may introduce the signature

```
signature SIG'' =
    sig
       val x : int
    end
```

and subsequently define a view T of the structure S by declaring

```
structure T : SIG'' = S
```

It should be clear that S matches the signature SIG'' because it provides an x component of type int. The signature SIG'' in the declaration of T causes the t component of S to be hidden, so that subsequently only the identifier T.x is available.

Example 9.9 ML Geometry Signatures and Structures

This example gives signatures and structures for a simple geometry program. An associated functor, for which structure parameterization is used, appears in Example 9.11. The three following signatures describe points, circles, and rectangles, with each signature containing a type name and names of associated operations. Two signatures use the SML include statement to include a previous signature. The effect of include is the same as copying the body of the named signature and placing it within the signature expression containing the include statement:

```
signature Point =
    sig
        type point
        val mk_point : real * real → point
        val x_coord : point - real
```

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

```
        val y_coord : point - real
        val move_p : point * real * real → point
    end;
signature Circle =
    sig
        include Point
        type circle
        val mk_circle : point * real → circle
        val center : circle → point
        val radius : circle → real
        val move_c : circle * real * real → circle
    end;
signature Rect =
    sig
        include Point
        type rect
    (* make rectangle from lower right, upper left corners *)
        val mk_rect : point * point → rect
        val lleft : rect → point
        val uright : rect → point
        val move_r : rect * real * real → rect
    end;
```

Here is the code for the Point, Circle, and Rect structures:

```
structure pt : Point =
    struct
        type point = real*real
        fun mk_point(x,y) = (x,y)
        fun x_coord(x,y) = x
        fun y_coord(x,y) = y
        fun move_p((x,y):point,dx,dy) = (x+dx, y+dy)
    end;
structure cr : Circle =
    struct
        open pt
        type circle = point*real
        fun mk_circle(x,y) = (x,y)
        fun center(x,y) = x
        fun radius(x,y) = y
        fun move_c(((x,y),r):circle,dx,dy) = ((x+dx, y+dy),r)
    end;
structure rc : Rect =
    struct
        open pt
```

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

```
    type rect = point * point
    fun mk_rect(x,y) = (x,y)
    fun lleft(x,y) = x
    fun uright (x,y) = y
    fun move_r(((x1,y1),(x2,y2)):rect,dx,dy) =
        ((x1+dx,y1+dy),(x2+dx,y2+dy))
end;
```

9.4 GENERIC ABSTRACTIONS

Abstract data types such as stacks or queues are useful for storing many kinds of data. In typed programming languages, however, the code for stacks of integers is different from the code for stacks of strings. The two different versions of stacks are written with different type declarations and may be compiled to code that allocates different amounts of space for local variables. However, it is time consuming to write different versions of stacks for different types of elements and essentially pointless because the code for the two cases is almost identical. Thus, over time, most typed languages that emphasize abstraction and encapsulation have incorporated some form of type parameterization.

9.4.1 C++ Function Templates

For many readers, the most familiar type-parameterization mechanism is the C++ template mechanism. Although some C++ programmers associate templates with classes and object-oriented programming, function templates are also useful for programs that do not declare any classes. We look at function templates briefly before considering module-parameterization mechanisms from other languages.

Simple Polymorphic Function

Suppose you write a simple function to swap the values of two integer variables:

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

Although this code is useful for exchanging values of integer variables, it is not written in the most general way possible. If you wish to swap values of variables of other types, then you can define a function template that uses a type variable `T` in place of the type name int:

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

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

The main idea is to think of the type name T as a parameter to a function from types to functions. When applied, or *instantiated*, to a specific type, the result is a version of swap that has int replaced with another type. In other words, swap is a general function that would work perfectly well for many types of arguments, except for the fact that the code contains the specific type int. Templates allow us to treat swap as a function with a type argument.

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

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

You may have noticed that the C++ keyword associated with a type variable is class. In C++, some types are classes and some, like int and float, are not. As illustrated here, the keyword class is misleading, because a template may be used with nonclass types such as int and float.

C++ templates are instantiated at program link time. More specifically, suppose that the swap function template is stored in one file and compiled and a program calling swap is stored in another file and compiled separately. The so-called relocatable object files produced when the calling program is compiled will include information indicating that the compiled code calls a function swap of a certain type. The program linker is designed to combine the two program parts by linking the calls to swap in the calling program to the definition of swap in a separate compilation unit. It does so by instantiating the compiled code for swap in a form that produces code appropriate for the calls to swap. If the calling program calls swap with several different types, then several different instantiated copies of swap will be produced. A different copy is needed for each type of call because swap declares a local variable tmp of type T. Space for tmp must be allocated in the activation record for swap. Therefore, the compiled code for swap must be modified according to the size of a variable of type T. If T is a structure or object, for example, then the size might be fairly large. On the other hand, if T is int, the size will be small. In either case, the compiled code for swap must "know" the size of the datum so that addressing into the activation record can be done properly.

Operations on Type Parameters

The swap example is simpler than most generic functions in several respects. The most important is that the body of swap does not require any operations on the type parameter T, other than variable declaration and assignment. A more representative example of a function template is the following generic sort function:

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

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

```
    for (int j=i+1; j<count-1; j++)
        if (A[j] < A[i]) swap(A[i],A[j]);
}
```

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

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

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

Somewhat better error reporting for C++ templates could be achieved if the template syntax included descriptions of the operations needed on type parameters. However, this is relatively complicated in C++ because of overloading and other properties of the language. In contrast, ML has simpler overloading and includes more information in parameterized constructs, as described in Subsection 9.4.2.

Comparison with ML Polymorphism

Here is the ML code for a polymorphic sort function. The algorithm used in this code is insertion sort, which uses the subsidiary function insert that appears first:

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

In the ML polymorphic sort function, the less-than operation is passed as a function argument to sort. No instantiation is needed because all lists are represented in the same way (with cons cells, as in Lisp).

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

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

9.4.2 Standard ML Functors

In the ML module system, structures may be parameterized in very flexible ways. We look at this part of the ML module system as a good example of the ways that modules can usefully be parameterized in programming languages. Module parameterization is different from ML polymorphism and is essentially independent of it. More specifically, the ML module design could be adapted to languages that do not have polymorphism or type inference.

In standard ML, a parameterized structure is called a functor. Functors are written in a functionlike form, with a parameter list containing structure names and signatures. Here is an example that uses the signature SIG defined in Subsection 9.3.2:

```
functor F ( S : SIG ) : SIG =
    struct
        type t = S.t * S.t
        val x : t = (S.x,S.x)
    end
```

This code declares a functor F that takes as argument a structure matching the signature SIG and yields a structure matching the same signature. When applied to a suitable structure S, the functor F yields as its result the structure whose type component t, is bound to the product of S.t with itself and whose value component x, is the pair whose components evaluate to the value of S.x.

When free structure variables in signatures are used, certain forms of dependency of functor results on functor arguments may be expressed. For example, the following declaration specifies the type of y in the result signature of G in terms of the type component t of the argument S:

```
functor G ( S : SIG ) : sig val y : S.t * S.t end =
    struct
        val y = (S.x,S.x)
    end
```

The rest of this section consists of examples that show how SML module parameterization can be used to solve some programming problems. Example 9.10 discusses a form of container structure and Example 9.11 completes the geometry example started in Example 9.9.

Example 9.10 Parameterized Priority Queue Module

Container structures such as sets, queues, dictionaries, and so on are used in many programs. This example shows how to define a container structure, priority queues, when one or more container operations require some operation on the type of objects stored in the container. Because priority queues require an ordering on the elements stored in a queue, our definition of priority queues is parameterized by a structure consisting of a type and order relation.

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

We begin by defining the signature of a type with an order relation:

```
signature Ordered =
   sig
      type t;
      val lesseq : t * t ⟶ bool;
   end;
```

Any structure that matches signature Ordered must have a type t and a binary relation lesseq. We assume in the definition of priority queue that this relation is a total order on the type t. In this functor definition, we declare a type-implementing priority queue operations and a set of operations on priority queues. Because the insert or deletemax operations must test whether an element is less than other elements, at least one of these operations will use the operation elem.lesseq provided by the parameter structure elem:

```
functor PQ(elem:Ordered) =
    struct
       type pq = . . . elem.t . . . ;              (* representation uses element type
*)
       val empty : pq                = . . . ;
       fun insert: elem.t * pq ⟶ pq   = . . . ; (* operations may use elem.lesseq
*)
       fun deletemax: pq ⟶ elem.t * pq = . . . ;
    end;
```

Example 9.11 Completion of Geometry Example

This example shows how the Point, Circle, and Rect structures in Example 9.9 can be combined to form a geometry structure with operation on all three kinds of geometric objects. To show how additional operations can be added when modules are combined, the Geom signature contains all of the geometric types and operations, plus a bounding box function, bbox, intended to map a circle to the smallest rectangle containing it:

```
signature Geom =
   sig
      include Circle
(* add signature of Rect, except Point since Point is already defined in Circle*)
      type rect
      val mk_rect : point * point ⟶ rect
      val lleft : rect ⟶ point
      val uright : rect -⟶ point
```

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

```
      val move_r : rect * real * real → rect
(* include function computing bounding box of circle *)
      val bbox : circle → rect
  end;
```

The geometry structure is constructed with a functor that takes Circle and Rect structures as arguments. The signatures Circle and Rect both name a type called point. The two point types must be implemented in the same way. Otherwise, operations that involve both circles and rectangles will not work properly. The constraint that two structures (Circle and Rect) must share a common substructure (point) is stated explicitly in the sharing constraint in the parameter list of this functor:

```
functor geom(
    structure c:Circle
    structure r:Rect
    sharing type r.point = c.point (*constraint requires same type in both
structures *)
) : Geom =
     struct
        type point = c.point
        val mk_point = c.mk_point
        val x_coord = c.x_coord
        val y_coord = c.y_coord
        val move_p = c.move_p
        type circle = c.circle
        val mk_circle = c.mk_circle
        val center = c.center
        val radius = c.radius
        val move_c = c.move_c
        type rect = r.rect
        val mk_rect = r.mk_rect
        val lleft = r.lleft
        val uright = r.uright
        val move_r = r.move_r
(* Bounding box of circle *)
        fun bbox(c) =
            let val x = x_coord(center(c))
            and y = y_coord(center(c))
            and r = radius(c)
            in
                mk_rect(mk_point(x-r,y-r),mk_point(x+r,y+r))
            end
     end;
```

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

9.4.3 C++ Standard Template Library

The C++ Standard Template Library (STL) is a large program library developed by Alex Stepanov and collaborators and adopted as part of the C++ standard. The STL provides a collection of container classes, such as lists, vectors, sets, and maps, and a collection of related algorithms and other components. Because STL is a good example of a large, practical library with generic structures, we survey some of the main concepts. The goal in this section is not to teach you enough about STL so that you can use it easily, but rather to explain the concepts so you can appreciate some of the design ideas and the power of the system.

One striking feature of STL is the run-time efficiency of the code that is generated. STL not only makes it easy to write certain programs by providing useful structures, STL makes it possible to write library-based code that is as fast or faster than the code you might produce if you worked much harder and did not use STL. One reason for the efficiency of STL is that C++ templates are expanded at compile/link time, with a separate code generated (and optimized) for each instantiation. Another reason is the use of overloading, which is resolved at compile time, similarly allowing the compiler to optimize code that is selected for exactly the type of data manipulated in the program. On the other hand, because of code duplication resulting from template expansion, the compiled code of programs that use STL may be large.

In this subsection, the term *object* is used to mean a value stored in memory. This is consistent with the use of the term in C, C++, and STL documentation. STL relies on C++ templates, overloading, and the typing algorithms that figure out type parameters to templates. STL does not use virtual functions and, according to its developer, is not an example of object-oriented programming.

There are six kinds of entities in STL:

- *Containers*, each a collection of typed objects.
- *Iterators*, which provide access to objects of a container. Intuitively, an iterator is a generalization of pointer or address to some position in a container.
- *Algorithms*.
- *Adapters*, which provide conversions between one form of entity and another. An example is a reverse iterator, which reverses the direction associated with an iterator.
- *Function objects*, which are a form of closure (function code and associated environment). These are used extensively in a form that allows function arguments to templates to be in-lined. More specifically, a function object is passed as two separate arguments, a template argument carrying the code and a run-time argument carrying the state of the function. The type system is used to make sure that the code and state match.
- *Allocators*, which encapsulate the memory pool. Different allocators may provide garbage-collected memory, reference-counted memory, persistent memory, and so on.

An example will give you some feel for how STL works.

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

Example 9.12 STL Lists

List is one of the simplest container types defined in STL. Here is sample code that declares a list of strings called wordlist:

```
#include <string>
#include <list>
int main (void) {
    list<string> wordlist;
}
```

The code list<string> wordlist instantiates a template class list<string> and then creates an object of that type. STL gives you operations for putting elements on the front or back of a list. Here is sample code to add words to our word list:

```
wordlist.push_back("peppercorn");
wordlist.push_back("funnybone");
wordlist.push_front("nosegay");
wordlist.push_front("supercilious");
```

We now have a list with four strings in it. As the names suggest, the list member function push_back() places an object onto the back of the list and push_front() puts one on the front. We can print the words in our list by using an iterator. Here is the declaration of an iterator for lists of strings:

```
list<string>::iterator wordlistIterator;
```

As suggested earlier in this section, an STL iterator is like a pointer – an iterator points to an element of a container, and dereferencing a nonnull iterator produces an object from the container. An STL iterator is declared to have a specific container type; this is important because moving a "pointer" through a container works differently for different types of containers. Here is sample code that prints the words in the list:

```
for (wordlistIterator=wordlist.begin();
    wordlistIterator!=wordlist.end();
    ++wordlistIterator) {
// dereference the iterator to get the list element
    cout << *wordlistIterator << endl;
}
```

We determine the bounds of this for loop by calling wordlist.begin(), which returns an

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

iterator pointing to the beginning of the container, and wordlist.end(), which returns an iterator pointing to the position one past the end of the container. The loop variable is incremented by the operator ++, which is defined on each iterator and moves the iterator from one list element to the next. In the body of the loop, we dereference the iterator to obtain the list element at the current position.

As Example 9.12 shows, iterators are essential for programming with STL containers. Most STL algorithms use iterators of some kind, and there are several different kinds of iterators. The simplest form, so-called *trivial iterators*, may be dereferenced to refer to some object, but operations such as increment and comparison are not guaranteed to be supported. Input iterators may be dereferenced to refer to some object and may be incremented to obtain the next iterator in a container. Output iterators are a fairly restricted form of iterator that may be used for storing (but not necessarily accessing) a sequence of values. The other forms of iterators are forward iterator, bidirectional iterator, and random-access iterator. Specific STL algorithms may require specific kinds of iterators, depending on whether the algorithm reads from a container or stores values into a container and whether the access is sequential (like reading or writing a tape) or random access.

Another basic concept in STL is a *range*. A range is the portion of a container defined by two iterators. The elements of a range are all the objects that may be reached by incrementing the first interator until it reaches the value of the second iterator. If you think of an array as an example container, then a range is a pair of integers, the first less than the second and both less than the length of the array, representing the portion of the array lying between the first index and the second.

The following example discusses the STL merge function, which uses ranges as inputs and outputs. In general, a merge operation combines two sorted sequential structures and produces a third. In merge sort, for example, the merge phase combines two sorted lists into a longer sorted list that contains all the elements of the two input lists.

Example 9.13 Generic Merge Function

The generic STL merge function takes two ranges as inputs and returns another range. Conceptually, the type of merge is

```
merge : range(s) x range(t) x comparison(u) → range(u)
```

where range(s) means a range of a container that contains objects of type s. Because C++ has subtyping (explained in Chapters 10 and 12), the types s, t, and u need not be identical. However, types s and t must be subtypes of u, meaning that every object of type s or t can be viewed as or converted to an object of type u. The informal notation comparison(u) means that merge requires a comparison operation (less-than-or-equal) on type u, so that merge can combine structures in order.

In fact, each input range in STL is passed as two iterators and an output range is passed as one iterator. The reason we need two iterators for each input range is that we need to know how many elements in the range. Because the output of merge is determined by the input, the output range is represented by a single output iterator

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

that marks the place to begin writing the merged container. Therefore, merge has the following parameters:

```
merge(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator1 last2,
        OutputIterator result, Compare comp)
```

In words, the first two arguments provide the first input range, the next two arguments the second input range, the fifth argument the output range, and the sixth argument must provide a comparison function.

Merge is a polymorphic function, merging ranges that contain any types of objects, as long as the types are consistent with each other. The polymorphism of merge is implemented by a template with four parameters. The template requires two iterator types, one for each of the two sequential structures that will be merged, together with the type of output structure and the type of the comparison. This brings us to the full header of merge:

```
template<typename InputIterator1,
        typename InputIterator2,
        typename OutputIterator,
        typename Compare> inline
OutputIterator merge( InputIterator1 first1,
              InputIterator1 last1,
              InputIterator2 first2,
              InputIterator2 last2,
              OutputIterator result,
              Compare comp)
```

It is possible to call merge without supplying a comparison function object. In this case, operator< is used by default. Here is a short program that uses merge, taken from the *Standard Template Library Programmer's Guide*:

```
int main()
{
int A1[ ] = { 1, 3, 5, 7 };
int A2[ ] = { 2, 4, 6, 8 };
const int N1 = sizeof(A1) / sizeof(int);
const int N2 = sizeof(A2) / sizeof(int);

merge(A1, A1 + N1, A2, A2 + N2,
    ostream_iterator<int>(cout, " "));
// The output is "1 2 3 4 5 6 7 8"
}
```

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

As you can see from this sample code, even though the concepts and design of STL are subtle and complex, using STL can be very easy and direct once the concepts are understood.

9.5 CHAPTER SUMMARY

In this chapter, we studied some of the ways that programs can be divided into meaningful parts and the way that programming languages support these divisions. The main topics were structured programming, support for abstraction, and modules. Two examples were used to investigate module systems and generic programming: the standard ML module system and the C++ Standard Template Library.

Structured Programming

Historically, interest in program organization grew out of the movement for structured programming. In the late 1960s, Dijkstra and others advocated a top-down approach. Although most programs of that day were small, currently many of the early ideas continue to be influential. Most important, we are still concerned with the decomposition of a complex programming task into smaller subproblems that can be solved independently.

Top-down programming has its limitations. Top-down design is difficult when the interfaces and data structures used to communicate between modules must be changed. In addition, the concept of top-down programming does not provide any help when programmers discover that the initial design must be modified to account for phenomena that were not fully appreciated in the initial design. As other design methods have emerged, the principle of decomposing a design problem into separable parts remains central to managing large software development projects.

Two important concepts in modular program development are interfaces and specifications:

Interface:

A description of the parts of a component that are visible to other program components.

Specification:

A description of the behavior of a component, as observable through its interface.

When a program is designed modularly, it should be possible to change the internal structure of any component as long as the behavior visible through the interface remains the same.

Language Support for Abstraction

Procedural abstraction and data abstraction are programming language concepts that allow sequences of statements, or data structures and code to manipulate them, to be grouped into meaningful program units. Here are three features of data-abstraction mechanisms:

- Identifying the interface of the data structure. The interface of a data abstraction consists of the operations on the data structure and their arguments and return results.

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

- Providing *information hiding* by separating implementation decisions from parts of the program that use the data structure.
- Allowing the data structure to be used in different ways by many different programs.

In typed programming languages, the typing rules associated with abstract data types guarantee *representation independence*, which means that changing the implementation of a data type does not affect the observable behavior of client programs. Another principle associated with abstract data types is *data-type induction*, which may be used to reason about the behavior of abstract data types or their equivalence.

Modules and Generic Programming

A *module* is a programming language construct that allows a number of declarations to be grouped together. Early forms of modules provided minimal information hiding. However, good module mechanisms allow a programmer to control the visibility of items declared in a module. In addition, parameterized modules make it possible to instantiate a set of declarations in different ways for different purposes.

The three main parts of the standard ML module system are *structures*, *signatures*, and *functors*. An ML structure is a module, which is a collection of type, value, and structure declarations. Signatures are module interfaces that behave as a form of "type" for a structure. Functors are parameterized modules, requiring one or more types or structures for producing a structure. We looked at a set of structures that provide points, circles, and rectangles, and a functor `Geom` that combines circles and rectangles into a geometry structure. One interesting aspect of this example is that the circle and the rectangle structures that are combined must have the same point representation. In standard ML, this is enforced by a sharing constraint in the parameter list of the `Geom` functor.

The C++ Standard Template Library is a large program library, adopted as part of the C++ standard, that provides container classes, such as lists, vectors, sets, and maps. The main concepts in STL are *Containers*, *Iterators*, *Algorithms*, *Adapters*, *Function objects*, and *Allocators*. We looked at lists, as an example container, and a `merge` function that combines two ranges from sorted collections into a single range. In STL, a range is represented as a pair of iterators, indicating a starting and ending "address" inside a collection.

Polymorphic functions such as `merge` are represented as function templates, instantiated and optimized before program execution. Function objects, which we did not study, are a form of closure used to pass functions to generic STL algorithms. A function object is passed as two separate arguments, a template argument carrying the code and a run-time argument carrying the state of the function. The type system is used to make sure that the code and the state match.

The concept explored in this chapter can be used in the design of general-purpose programming languages or special-purpose languages developed to solve specialized programming problems (such as specifying a chemical simulation or specifying the security policy of a large organization). Because you are much more likely to design a special-purpose input language in your career as a practicing computer scientist than you are to design a general-purpose object-oriented language, the non-object-oriented methods explained here may be useful to you in the future.

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

EXERCISES

9.1 Efficiency vs. Modularity

The text describes the following form of heapsort:

```
function heapsort1(n:int, A:array[1..n])
begin
    s := empty;
    for i := 1 to n do          s := insert(A[i], s);
    for i := n downto 1 do      (A[i],s) := deletemax(s);
end
```

where heaps (also known as priority queues) have the following signature

```
empty : heap
insert : elt * heap -> heap
deletemax: heap -> elt * heap
```

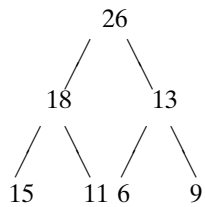
and specification:

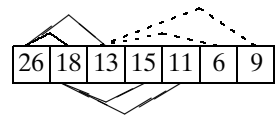
- Each heap has an associated multiset of elements. There is an ordering < on the elements that may be placed in a heap.
- empty has no elements.
- insert(elt, heap) returns a heap whose elements are elt plus the elements of heap.
- deletemax(heap) returns an element of heap that is ≥ all other elements of heap, together with a heap obtained when this element is removed. If heap is empty, deletemax(heap) raises an exception.

This problem asks you to reformulate the signature of heaps and the sorting algorithm so that the algorithm is more efficient, without destroying the separation between the sorting algorithm and the implementation of heaps. You may assume in this problem that we use pass-by-reference everywhere.

The standard nonmodular heapsort algorithm uses a clever way of representing a binary tree by an array. Specifically, we can think of an array A of length n as a tree by regarding A[1] as the root of the tree and elements A[2*k] and A[2*k+1] as the left and the right children of A[k]. One efficient aspect of this representation is that the links between tree nodes do not need do be stored; we only need memory locations for the data stored at the tree nodes.

We say a binary tree is a *heap* if the value of the root of any subtree is the maximum of all the node values in that subtree. For example, the following tree is a heap and can be represented by the array next to it:





The standard heapsort algorithm has the form

```
function heapsort2(n:int, A:array[1..n])
begin
    variable heap_size:int := n;
```

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

```
    build_heap(n,A);
    for i := n downto 2 do
        swap(A[1], A[i]);
        heap_size := heap_size - 1;
        heapify(heap_size,A);
end
```

where the procedure build_heap(n,A) reorders the elements of array A so that they form a heap (binary search tree) and procedure heapify(k,A) restores array elements $A[1], \ldots, A[k]$ to heap form, assuming that only $A[1]$ was out of place.

(a) Assume that procedures insert, deletemax, and heapify all take time $O(\log n)$, i.e., time proportional to the logarithm of the number of elements in the heap, and that build_heap takes time $O(n)$, i.e., time linear in the number of elements in the heap.* Compare the time and space requirements of the two algorithms, heapsort1 and heapsort2, and explain what circumstances might make you want to choose one over the other.

(b) Suppose we change the signature of heaps to

```
make_heap : array * int -> heap
deletemax : heap -> elt
```

where deletemax may have a side effect on the heap and there is no longer empty or insert.

In addition to the specification of a return value, which is needed for a normal function, a function f with side effects should specify the before and after values of any variables that change, using the following format:

f: If $\mathtt{x}_{before}$ is ... then $\mathtt{x}_{after}$ is ...

The idea here is that the behavior of a function that may change the values of its arguments can be specified by describing the relationship between values before the call and values after the call. This "if ... then ..." form also allows you to state any preconditions you might need, such as that the array A has at least n elements.

Write a specification for this modified version of heaps.

(c) Explain in words (or some kind of pseudocode if you prefer) how you might implement the modified form of heap with imperative operations described in part(b). You may assume that procedures such as build_heap, heapify, and swap are available. Assume that you will use your heaps for some form of heapsort. Try to make your operations efficient, at least for this application if not for all uses of heaps in general.
[*Hint*: You will need to specify a representation for heaps and an implementation of each function. Try representing a heap by a pair $\langle i, A \rangle$, where A is an array and $0 < i \leq \text{length}(A)$.]

(d) Write a heapsort algorithm (heapsort3) using the modified form of heap with imperative operations described in part (b). How will the time and the space compare with the other two algorithms?

* Function build_heap works by a form of iterated insertion. This might require $O(n \log n)$, but analysis of the actual code for heapify allows us to show that it takes $O(n)$ time. If this interests or puzzles you, see *Introduction to Algorithms*, by Cormen, Leiserson, and Rivest (MIT Press, 1990), Section 7.3.

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

(e) In all likelihood, your algorithm in part (d) (heapsort3) is not as time and space efficient as the standard heapsort algorithm (heapsort2). See if you can think of some way of modifying the definition or implementation of heaps that would let you preserve modularity and meet the optimum efficiency for this algorithm, or explain some of the obstacles for achieving this goal. (Alternatively, argue that your algorithm *is* as efficient as the standard heapsort.)

9.2 Equivalence of Abstract Data Types

Explain why the following two implementations of a point abstract data type are equivalent. More explicitly, explain why any program using the first would give exactly the same result (except possibly for differences in the speed of computation and any consequences of round-off error) as the other.

Appeal to the principle of data-type induction, or related ideas, from this chapter. You need not give a detailed formal inductive proof; a simple informal argument referring to the principles discussed in class will be sufficient. You may explain what needs to be calculated or what conditions need to be checked without doing the calculations. The point of this problem is not to review trigonometry. An answer consisting of three to five sentences should suffice.

```
(* ———————— Trig function for arctan(y/x) ———————— *)
fun atan(x,y) =
                    let val pi = 3.14159265358979323844
                     in
                      if (x > 0.0) then arctan(y / x)
                                   else (if (y > 0.0) then (arctan(y / x) + pi)
                                                      else (arctan(y / x) - pi))
                      handle Div ⇒ (if (y > 0.0) then pi/2.0 else ~pi/2.0)
                    end;
(* ———————— Two point abstract types ———————— *)
abstype point = Pt of (real ref)*(real ref) (* rectilinear coordinates *)
      with fun mk_Point(x,y) = Pt(ref x, ref y)
      and x_coord (Pt(x, y)) = !x
      and y_coord (Pt(x, y)) = !y
      and direction (Pt(x, y)) = atan(!x, !y)
      and distance (Pt(x, y)) = sqrt(!x * !x + !y * !y)
      and move (Pt (x, y), dx, dy) = (x := !x + dx; y := !y + dy)
end;

abstype point = Pt of (real ref)* (real ref) (* polar coordinates *)
with fun mk_Point(x,y) = Pt(ref (sqrt(x*x + y*y)), ref (atan(x,y)))
      and x_coord (Pt(r, t)) = !r * cos(!t)
      and y_coord (Pt(r, t)) = !r * sin(!t)
      and direction (Pt(r, t)) = !t
      and distance (Pt(r, t)) = !r
      and move (Pt(r, t), dx, dy) =
            let val x = !r * cos(!t) + dx
                and y = !r * sin(!t) + dy
            in r := sqrt(x*x + y*y); t := atan(x,y) end
end;
```

The types of the functions given in either declaration are listed in the following

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

output from the ML compiler:

```
type point
val mk_Point = fn : real * real -> point
val x_coord = fn : point -> real
val y_coord = fn : point -> real
val direction = fn : point -> real
val distance = fn : point -> real
val move = fn : point * real * real -> unit
```

9.3 Equivalence of Closures

Explain why the following two functions that return a form of point "objects," represented as ML closures of the same type, are equivalent. More explicitly, explain why any program using the first would give exactly the same result (except possibly for differences in the speed of computation and any consequences of round-off error) as the other.

We did not cover any induction or equivalence principle for closures, but you might think about whether you can use the same sort of reasoning you used for abstract data types in Problem 9.2.

```
(* ———————— Two point objects (as closures) ———————— *)
fun mk_point(xval,yval) =
    let val x = ref xval and y = ref yval
    in {
          x_coord = fn () ⇒ !x,
          y_coord = fn () ⇒ !y,
          direction = fn () ⇒ atan(!x, !y),
          distance = fn () ⇒ sqrt(!x * !x + !y * !y),
          move = fn (dx, dy) ⇒ (x := !x + dx; y := !y + dy)
        }
    end;

fun mk_point(x,y) =
    let val r = ref (sqrt(x*x + y*y)) and t = ref (atan(x,y))
    in {
          x_coord = fn () ⇒ !r * cos(!t),
          y_coord = fn () ⇒ !r * sin(!t),
          direction = fn () ⇒ !t,
          distance = fn () ⇒ !r,
          move = fn (dx, dy) ⇒
              let val x = !r * cos(!t) + dx
                  and y = !r * sin(!t) + dy
              in r := sqrt(x*x + y*y); t := atan(x,y) end
        }
    end;
```

9.4 Modularity of Concrete Data Types

Given a grammar for a language, we can define ML data types for parse trees of expressions in that language. Consider a grammar for expressions involving binary (base 2) numbers and the left shift operator (<<) from C. We will use 0^b and 1^b to

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

distinguish the syntax of bits from the numerals 0 and 1.

$B ::= 0^b \mid 1^b$
$N ::= B \mid N\,B$
$E ::= N \mid E << E$

In ML, we can create one data type for each nonterminal in the grammar:

```
datatype B = zero | one
datatype N = bit of B | many of N * B
datatype E = num of N | leftshift of E * E
```

A parse tree for an expression in our language can be written in ML with the appropriate constructors. For example, the expression $0^b1^b \ll 1^b1^b$ would be written `leftshift(many(num(bit(zero), one)), many(num(bit(one), one)))`.

We may have a denotational semantics for expressions in this language. For example, a function to evaluate binary expressions could be written as

$$\begin{aligned} \text{eval}\,[\![0^b]\!] &= 0, \\ \text{eval}\,[\![1^b]\!] &= 1, \\ \text{eval}\,[\![nb]\!] &= 2 \cdot \text{eval}\,[\![n]\!] + \text{eval}\,[\![b]\!], \\ \text{eval}\,[\![e_1 \ll e_2]\!] &= \text{eval}\,[\![e_1]\!] \cdot 2^{\text{eval}[\![e_2]\!]}. \end{aligned}$$

In ML, we would have to write one function per data type:

```
fun evalB(zero) = 0
  | evalB(one) = 1
fun evalN(bit(b)) = evalB(b)
  | evalN(many(n,b)) = 2 * evalN(n) + evalB(b)
```

(a) Write `evalE`, which has type `E` $\longrightarrow$ `int`. Assume you can use the `power` function in ML, e.g. `power(2,3)` will return `8`.

If you want to try your program in SML, you may define `power` by

```
fun power(b,0) = 1
  | power(b,e) = b*power(b,e-1)
```

(b) Discuss what has to be changed to add an operator (such as `>>`) to expressions. Do not worry about how `>>` actually works.

(c) Discuss what has to be changed to add a new denotational semantics function (such as `odd`, a function that takes a binary expression and returns `true` if there are an odd number of 1^b bits in it). Do not worry about how `odd` actually works.

(d) If Alice modified the program to add a function `bitcount` and Bob modified it to add a function `odd`, how hard is it to combine the two modifications into the same program? What has to be changed? Explain.

(e) If instead, Alice modified the program to add an operator `>>` and at the same time Bob modified the program to add the `odd` function, how hard is it to combine the two modifications into the same program? What has to be changed? Explain.

(f) If instead, Alice modified the program to add an operator `>>` and Bob modified the program to add an operator `xor`, how hard is it to combine the two modifications into the same program? What has to be changed? Explain.

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press

In your discussion of changes, assume that the parts of the program can be classified as one of four kinds:

- data-type declarations for binary expressions
- functions that perform pattern matching on binary expressions
- functions that use binary expressions but do not perform pattern matching
- functions that do not use binary expressions

For each kind of program component, you should discuss whether it has to be changed, what has to be changed, and why. If it does not have to be changed, explain why it can be left unchanged.

9.5 Templates and Polymorphism

ML and C++ both have mechanisms for creating a generic stack implementation that can be used for stacks with any type of element. In ML, polymorphic stacks could be written as

```
datatype 'a stack = Empty | Push of 'a * 'a stack

fun top(Empty) = raise EmptyStack
|   top(Push(x,s)) = x

fun pop(Empty) = raise EmptyStack
|   pop(Push(x,s)) = x
```

To achieve a similar effect in C++, we could write a template for stack objects of the following form:

```
template <typename A> class node {
public:
    node(A v,node<A>* n) {val=v; next=n;}
    A val;
    node<A>* next;
};

template <typename A> class stack {
     node<A>* first;
public:
     stack () { first=0; }
     void push(A x) { node<A>* n = new node<A>(x,first); first=n; }
     void pop() { node<A>* n=first; first=first->next; delete n; }
     A top() { return(first->val); }
};
```

Assume we are writing a program that uses five or six different types of stacks.

(a) For which language will the compiler generate a larger amount of code for stack operations? Why?

(b) For which language will the compiler generate more efficient run-time representations of stacks? Why?

https://doi.org/10.1017/CBO9780511804175.010 Published online by Cambridge University Press