Chapter 17

Generics

Overview

Generic domains are domains which include more than one specific type of object. They are used to express abstractions, to make code reusable, and to support top-down programming. There are four kinds of generic domains: parameterized, subdomains, polymorphic, and ad-hoc.

Several issues must be considered when implementing generic domains. They have been considered in the support for generics that has been built into various languages. The solutions vary in flexibility, preservation of semantic validity, binding time, and efficiency. Approaches to this problem include: overloaded operators, flexible arrays, polymorphic types, parameterized domains with instantiation, class hierarchies with inheritance, and declarable domain relationships.

A corresponding tool for expressing procedural abstractions is the virtual function. A virtual function is a function name, header, and description but no code. An ADT can be represented in a programming language by a generic domain together with virtual function declarations.

A generic function is a single abstract operation defined over a generic domain. Most languages have a few primitive generic functions for all primitive types, which are automatically extended to user-defined types. However, user-defined generic functions create semantic problems. The code that implements any process must be appropriate for the type of its arguments; what works for one type does not for another. The translator, therefore, must be able to handle multiple definitions for a generic function, and it must decide which method to use for each function call. This process is known as dispatching.

An ad-hoc generic domain has subdomains that are related by semantics rather than by

structure. Definitions of a function over two specific domains with an ad-hoc relationship may look different if they depend on representation. However, they perform the same semantic action on objects of the different specific domains.

Many older languages support some predefined generics for forms of nonhomogeneous domains. These include union data types, overloaded names, fixed sets of generic definitions with coercion, extending predefined operators, flexible arrays, parameterized generic domains, domains with type parameters, preprocessors, and generic packages. These are far more limited than support for generic types found in object-oriented languages because final binding of the type parameters happens at precompile time.

17.1 Generics

17.1.1 What Is a Generic?

Chapter 15 discussed domains, type checking, and type conversions in several familiar strongly typed languages. In these languages, types are used to define domains such that each new type constructed, with a few exceptions, defines a distinct domain incompatible with other domains, and each domain contains only one type of object. In this chapter we examine the ways to declare and use generic domains, that is, domains that include more than one type of object. We see how generic domains can be used to express abstractions, to make code reusable, and to support top-down programming.

We say the domain of a function is *specific* if every argument to the function is defined to be a single specific type. The properties of a specific domain are fully defined by the specific type definition used to define the function, and the function makes use of these properties in order to perform meaningful computations. The opposite of "specific" is "generic". A function domain is *generic* if some argument can be of two or more different specific types. There are four ways in which a domain, D, may be *generic*:

- D may be defined by a type expression with components of an unspecified type and/or an array with unspecified array bounds. We call such a type expression a *parameterized type.* Several languages provide some support for this kind of generics. The domain it defines is a *parameterized generic domain.*[1]
- D may have a subdomain, D'. This occurs when D is defined as a type for which subtypes have been declared. For example, the Pascal subrange type "1..10" is a subtype of the type `integer`.

[1]Implementations of generics over parameterized domains are discussed in Section 17.3.

Exhibit 17.1. Generic domains and functions on them.

Domain	Generic Functions on the Domain
number	`+, -, *, /`
matrix of numbers	inverse, transpose, `+`
character string	concatenation, substring
set of ?	union, intersection
stack of ?	push, pop, top

- D may be defined by a polymorphic type. A *polymorphic* type is a single type with internal variability, like a discriminated union.
- D is an *ad hoc* generic domain if it includes objects of more than one specific type, such that all of the specific types are representations of the same external domain. The semantic relationships among these species are important, but the species are related in an ad hoc manner, not by structure or representation. For example, the generic domain "number" includes at least two specific domains "integer" and "real" in most languages, and the generic domain "tree_node" can be defined to include specific types "leaf_node" and "interior_node". Ad hoc generics are a topic of current research interest.

Consider the domains listed in Exhibit 17.1. The domains "number" and "matrix of numbers" are generic because they include both integers and reals. "Matrix" and "string" are generic because their dimensions are not specified, so these types include matrices of all shapes and strings of all lengths. "Set" is generic because the base type of the set is not specified and, therefore, sets of reals, characters, cells, and any other type. are all included. "Stack" is generic for the same reason, and also because the size and structure of stacks can vary.

17.1.2 Implementations of Generics

There are several ways that generic and polymorphic functions may be supported within a programming language, but all approaches must provide answers for the same syntactic and semantic problems. The issues to be considered include:

- What kind of generic and/or polymorphic domains may we define? Parameterized domains? Structurally related domains? Domains with a small number of variants? Unlimited ad hoc generics?
- How and under what conditions may we define a generic or polymorphic function? May we define functions over generic domains?
- How do we translate calls on functions with more than one specific method? What information is considered by the translator when it chooses a method? Can this choice be deferred until run time or is it always made at compile time?

- If we permit run-time type variability, is it possible to compile code with reasonable efficiency?
- How do generic domains and generic functions interact with type coercion?
- What does it mean to call a function which has a generic formal parameter? [See Exhibit 17.3] What type matching rules should apply? How can we implement such calls?

Support for generics has been achieved to different degrees for domains with subdomains, parameterized domains, and ad hoc generic domains. The solutions provided by various languages vary in their flexibility, their ability to preserve semantic validity, their time of binding, and the inefficiency inherent in the implementation. We will examine several approaches to this problem in some detail.

Overloaded Operators. A few generic arithmetic operators are built into the language. A programmer can define new methods for these operators but cannot define any new operators or functions that are generic in the same way. Ada's operators are an example and are discussed in Section 17.2.

Flexible Arrays. A programmer can use array parameters whose bounds are not defined at compile time. Either the array length must be passed as a separate parameter, or the code for the function must work correctly for all possible values of the array length. Several older languages support flexible arrays, including FORTRAN, C, and Pascal. These are covered in Section 17.2.

Parameterized Domains with Instantiation. The generic module is kept in the form of parameterized source code. A preprocessor is used to instantiate the parameterized code with actual type arguments and generate fully specific, ordinary code which is bound to a unique name and then compiled. The programmer uses the unique name, not the generic name, in the code. This kind of generic module is supported by Ada and can be implemented in C using macros. It is discussed in Section 17.3.

Class Hierarchies with Inheritance. Objects belong to classes, as do function methods. The symbol table is available at run time, and the class of the actual arguments to a function can be examined. One name is given to an abstract function, and many methods can be defined for that name, so long as each method is defined in a different class. A function method has one implied argument, and the translator uses the class of that argument to determine which method to dispatch for each function call. This kind of support for generics is present in Simula, Smalltalk, and C++. Because generic dispatching must sometimes be postponed until runtime, these languages are at least partly interpreted. This is discussed in Chapter 18, Section 18.2.

Polymorphic Types. The type of each symbol can be determined at run time. Functions are polymorphic, and each part of a function definition is controlled by a predicate. These predicates may be used to test the type or the value of the actual argument for each call. The translator will use a powerful pattern-matching facility to determine which predicate is true and select the corresponding action. Miranda and Haskell support this kind of generics. They are discussed in Chapter 18, Section 18.3.

Declarable Domain Relationships. Generic function calls are interpreted, not compiled. The type information for generic objects is kept available at run time. One name is given to an abstract function and is used to refer to all of its methods. A generalized graph of relationships between domains can be created or declared, and is used to dispatch functions. In a generic function call, the arguments might be of any specific type which is a subdomain of the generic domain, according to the graph. A function call with generic parameters translates into code that will look at the argument types at run time and execute the function method that is most appropriate for those types, using coercion if necessary. Examples of this approach to dispatching are drawn from C++ and Miranda and are discussed in Chapter 18, Section 18.4.

The extent to which a language supports generics is an important issue. The greater the extent, the more flexible and adaptable a program can be at run time. The smaller the extent, the more difficult it is to create reusable code and code libraries. Generic domains are supported to a minimal extent in most languages, but the kind of support provided, the restrictions, and whether support extends to programmer-defined domains and functions vary dramatically from one language to another. Traditional typed languages such as Pascal usually have some predefined generic domains with predefined relationships, but they lack any way for programmers to define their own generic domains or domain relationships. For example, a general matrix multiplication function cannot be defined in Pascal or FORTRAN because there is no way in these languages to define the generic domain "matrix of integers or reals".

Programmer-defined subdomains and domains with integer parameters are relatively easy to implement and are supported in many languages. Domains with type parameters are more difficult to implement in a strongly typed, compiled language because ordinary type checking requires that the specific type of each function argument be known (or deduced) at compile time. Ad hoc generic domains are the most difficult to implement because there is no structural relationship among the variants that can be exploited.

17.1.3 Generics, Virtual Functions, and ADTs

A generic domain is an abstraction that represents the shared structural and/or semantic properties of its subdomains. Thus a language that permits programmers to declare their own generic domains provides a powerful tool with which to express data abstractions. A corresponding tool for expressing procedural abstractions is the *abstract function*, or *virtual function*. This is a function name, a function header (or "prototype"), and a description of the intent of the function, with no accompanying code. The header defines the number and types of the function parameters and re-

turn value. The accompanying description defines the intended semantics of each, and any relevant assumptions or restrictions.

An abstract data type can be represented in a programming language by a generic domain together with a collection of virtual function declarations. A virtual function would be included for each essential representation-dependent process to be performed on members of the domain. This set of virtual functions explicitly describes the common behavior of all objects of the generic domain and is, thus, a description of the semantics of the generic domain. The virtual functions would be "guaranteed" to exist for every subdomain and would form the basis for defining all functions on the generic domain. In a few languages, a programmer is able to explicitly declare and name a generic domain, D, then use a single method to define a generic function, GF, over D. The function header for GF would refer to D, and the code for GF could call the virtual functions defined for D and rely only upon properties that are common to all specific domains contained in D.

A *realization* of a generic domain is some specific, fully defined type that implements the semantics of that domain. All realizations of a generic domain must have methods defined for all of the domain's virtual functions. A program could contain several realizations of the same generic domain. For example, in Pascal, `integer` and `real` are both realizations of the domain "number". The translator and linker must guarantee that, by run time, realizations of all promised functions for all subdomains must exist.

17.1.4 Generic Functions

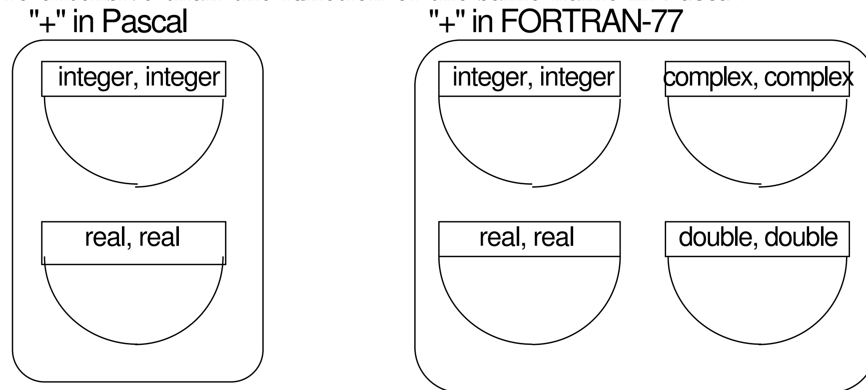
A *generic function* is a single abstract operation that is defined over a generic domain. In most languages, `FETCH`, `STORE`, and comparison for equality are primitive generic functions that are defined for all primitive types. Most languages developed since the early 1970s extend these primitive generics automatically to user-defined types. Let us call these three functions the *universal generic functions*. Other functions that are commonly generic are `READ`, `WRITE`, and the arithmetic operators.

A generic function creates a semantic problem. In order for a process to be meaningful, the code that implements the process must be appropriate for the type of its arguments. In general, code that is meaningful for one type is not meaningful for other types. We must, therefore, ask how a function can meaningfully process data from a generic domain. In the older languages and their translators, this was done in an ad hoc manner. With a limited number of generic functions built into the language, as in Pascal and Ada, the translator can treat these functions as special cases, look at the types of the actual arguments, and generate specific code that is appropriate for them. On the other hand, if the user is permitted to define his or her own generic functions, the translator's type-checking algorithms must be extended to form some general mechanism for handling generic functions. Such mechanisms are the subject of this chapter.

Depending on the language, a user-defined generic function may be defined by one block of code whose parameter is from a generic domain, or by several independent blocks of code which have been declared to share a common name. When a function is defined by more than one body of

Exhibit 17.2. A generic function as a collection of methods.

In this diagram a method is represented by a shield-shaped box, with its domain written across its top. A function is a set of methods with a common name. Note that the function "+" in FORTRAN 77 is much more extensive than the function of the same name in Pascal.



code, each of the separately defined bodies is called a *method* for the function, and each method must be declared with arguments from a different combination of domains. We define the domain of such a function to be the union of the domains of all its methods.

There are three ways that a generic function may be defined by a single block of code: the processing method can be independent of the argument type (as in comparison for equality), the code can be polymorphic (see Chapter 18, Section 18.3), or the code can bypass the problem of processing a generic argument by passing the argument on in a call to some other generic function [Exhibit 17.4]. Functions that operate over ad hoc generic domains are either polymorphic or are defined as a collection of methods [Exhibit 17.2].

Dispatching a Function Method

It is more difficult for a translator to handle function names with multiple methods than names that represent one block of code. The process of deciding which method to use for a call is known as *dispatching the call*, and it is carried out by a part of the translator called the *dispatcher*. The dispatcher uses information about the function arguments to choose an appropriate method. Dispatching could be (and is, in some languages) done on the basis of the number of arguments, the types of one or all arguments, or the actual value of the arguments.

Some languages will try to coerce a function argument if the types of the actual arguments do not match any of the defined methods. Depending on the language, the dispatcher might use a subtype relationship (Pascal or Ada), possible type conversions (Pascal, FORTRAN), or hierarchical relationships among the domains (C++, Smalltalk). For example, if the function `Square` in Exhibit 17.3 were called with an `integer` argument, the hierarchical relationship between the domains `integer` and `number` could be used to dispatch the function call. If none of the defined methods is

Exhibit 17.3. A function with generic formal parameters.

Assume that `number` is a generic type which includes types `integer` and `real`. If we were to define the function `Square` on `numbers` (using a Pascal-like syntax), the function header would be similar to this:

```
Function Square( a:  number):number;
```

Assume that the integer variables `k` and `m` have been declared, and we call the function thus:

```
k := Square(m);
```

We would like this call to be legal. The formal parameter, `a`, is generic, and the actual argument has a specific type which is a subtype of that generic. We would like the subtype `integer` to *inherit* the definition for `Square` that was supplied for its generic supertype.

---

appropriate for the context, and no coercion is permitted that can make the arguments appropriate for some method, the function call contains a *type error*.

In Chapter 18 we consider several very different implementations of programmer-definable generic functions, with very different rules for dispatching. Some of these rules, such as those in Miranda, are quite general and require a sophisticated dispatcher. Others, such as the hierarchical dispatching rule in Simula and Smalltalk are more limited and can be implemented simply.

Functions over Generic Domains

Some languages permit the programmer to define a generic function with an argument from a generic domain and only one defining method [Exhibit 17.4]. When such a function is called, the argument's type will be some specific type that is included in the generic domain. Such a method cannot operate directly on its parameters, but must either test its argument and take an appropriate branch (polymorphic behavior) or perform its process by calling other generic functions defined over the same domain. The lower-level function could, in turn, pass the argument on to another generic function. However, a chain of such generic calls must ultimately end with a call on a bottom-level polymorphic function or a generic function that is defined by a set of specific methods. This bottom-level function will include one body of code for each representation of the generic data type that the programmer is using. In the `prettyprint` example, the chain of calls will end with a call on some body for `print_node` that is defined to format and print out the data from one specific type of node.

Generic Functions in Ad hoc Domains

An ad hoc generic domain has subdomains that are related by their semantics rather than by having some common structure. Definitions of a function over two specific domains with an ad hoc relationship may look quite different if they depend on the representation. The most that can be

Exhibit 17.4. A generic function defined by a single method.

Assume that you are working with binary trees in a hypothetical language that permits definition of generic procedures. `Prettyprint` is a procedure that prints a generic data type called `binary_tree`.

`Prettyprint` calls the explicitly generic functions `print_node`, `go_to_left_son`, and `go_to_right_son`, but it does not do any operations that depend on how the tree is represented.

The programmer must define one method for each of these three functions for every kind of tree node being used. Once this is done, one method suffices to define `prettyprint` for the entire generic domain `binary_tree`, no matter how many ways that domain might be implemented.

said is that they perform the same semantic action on objects of the different specific domains. For example, consider the ADT "Stack", which includes the two generic subdomains "array_stack" and "linked_list_stack". The type declarations that are appropriate for implementing an "array_stack" and a "linked_list_stack" are shown in Exhibits 17.5 and 17.6. These sets of definitions have little in common except the names. Specifically, the selector functions defined in one bear no systematic relationship to the selectors defined for the other.

The code for `push` on an array-represented stack [Exhibit 17.5] bears no easily described rela-

Exhibit 17.5. Pascal **definition of** `push` **on an** `array_stack`**.**

Here are type declarations and the function `push` for a stack of 100 reals:

```
CONST len = 100;
TYPE real_stack= RECORD
        store:  ARRAY [1..len] OF real;
        top:  0..len  END;

VAR rs:  real_stack;
    value:  real;

FUNCTION push (VAR rs:real_stack; r:real):  boolean;
BEGIN
    IF rs.top = len THEN push := false (* error flag *)
    ELSE BEGIN
        push := true;
        rs.top := rs.top + 1;
        rs.store [rs.top] := r
    END (* IF rs.top *)
END; (* push *)
```

Exhibit 17.6. Pascal **definition of** `push` **for a** `list_stack`**.**

Compare the code below for the function `push` on a linked-list implementation of a stack to the code in Exhibits 17.5 and 17.15 for the same generic operation on an array implementation of a stack.

```
TYPE list_stack = ↑stack_cell;
    stack_cell = RECORD
        val :  real;
        next:  list_stack END;

VAR rs:  list_stack;
    value:  real;

FUNCTION push (VAR rs:list_stack; r:real):  boolean;
VAR t:list_stack;
BEGIN
    push := true;                (* stack cannot overflow *)
    t := rs;                     (* save current stack pointer *)
    NEW(rs);                     (* append new cell to top of stack *)
    rs↑.val := r;                (* initialize new cell *)
    rs↑.next := t
END (* push *)
```

tionship to the code for `push` on a linked-list-represented stack [Exhibit 17.6], even though they carry out the same abstract operation. Representation-dependent functions such as `push` and `pop` *must* be defined by separate methods for the species of an ad hoc generic domain. However, the common intent of the subdomains is known to the programmer, and the common functionality can be expressed by providing appropriate definitions for the stack operations. Further functions can then be defined in a representation-independent way for the generic domain `stack`, in terms of these declared abstract functions.

When two species of a generic domain are related by their intent, it is possible to define conversion functions that map the elements of one onto the elements of the other and vice versa. This mapping is done in such a way that corresponding elements are alternative representations of the same external object. When such a mapping is applied, the physical form of the program object is changed but the semantic intent is preserved. These conversion functions are unlike other functions. They do not merely operate on elements of the generic domain; they actually define the semantics of the generic relationship.

17.2 Limited Generic Behavior

Working with domains that have several natural representations becomes difficult in a strongly typed language when the representations must interact, unless the language permits some deviation from the one-type-one-semantics rule. For this reason, many older languages support some predefined generics and/or permit some flexibility in the types of array arguments. In this section we explore ways in which limited forms of nonhomogeneous domains can be and have traditionally been supported.

17.2.1 Union Data Types

Generic domains arise naturally in many algorithms. For example, consider the data structure called a "balanced tree", or "B-tree". This is a branching index structure in which internal nodes point to other nodes and leaf nodes point to potentially large data records. "Node", then, has two varieties, "internal" and "leaf", and the programmer working with these trees must mix the two representations in one structure. What is needed is the ability to define a type "node" that has two independent realizations, or a type "pointer-to-node" that can point at either a data record or another node.

Union data types have been used to implement balanced trees, but they are not a good solution for two reasons. First is the possible violation of the semantic intent unless the union is discriminated, in which case the translator guarantees that the discriminant tag is always correct, and the type checker always checks this tag. With fewer restrictions, a function could be applied to the wrong variant. Second, union data types are not a practical solution when the size of the more frequently used representation is much smaller than the size of the other, because of the excessive amount of storage that would be consumed by the allocated but unused portions of every record. B-trees are an example of a programmer-defined domain for which union data types are not a good implementation.

The C language is popular partly because it permits a direct implementation of this kind of pointer structure. Any C pointer type can be cast to any other pointer type, and such a cast can be used to make a pointer to a leaf node assignment-compatible with a pointer to an interior node. Since types are not checked when pointer assignments are done, there is no problem having pointers in the same tree point at two types of nodes. Programmers then become responsible for doing their own type checking on the nodes. This kind of flexibility is not available, though, in languages that provide the semantic protection of thorough type checking.

17.2.2 Overloaded Names

The several methods of a generic function implement a single external operation over different species of one generic domain. In an ideal language, we would have one name per generic function, whether the function was predefined (like "=") or defined by the programmer. For example, no matter how many instances of the stack ADT we make, we would like to be able to refer to the

Exhibit 17.7. An overloaded function, "+".

Many different abstract processes have been called "+", all of them dyadic, associative, and resembling addition in some way:

- Addition: Compute the arithmetic sum of two numbers.
- Logical OR: Result is "true" if either argument, or both arguments, are "true".
- Set union: Join two sets together and eliminate duplicates.
- String concatenation: Attach one string to the end of another.

"+" is an overloaded generic function because these processes do not have the same semantic intent, and their domains are unrelated. At the same time, ordinary addition is defined over the generic domain "number", which includes integers, reals, etc. This subset of "+" that deals with addition forms a pure generic function.

---

stack functions as "push" and "pop". It is evident that older languages, such as Pascal are far from ideal! If we defined two kinds of stacks in a Pascal program, we would have to define the stack functions twice and give them different names. In modern languages, we are permitted to "overload" a function name by defining multiple methods for one function. An *overloaded name* is a name with two or more bindings in the same scope.

Overloaded names, unfortunately, are not restricted to use with generic functions. If a language supports name overloading, nothing can stop a programmer from overusing it. An overloaded name could be used to denote semantically unrelated processes on unrelated types as easily as to denote a true generic function.

Ada supports the traditional generic arithmetic operators and extensible name overloading.[2] The semantics of these operators can be extended by adding new methods to the existing predefined set. This is called "overloading" because each operator name denotes several specific methods: a "heavy load" for one name. However, no distinction is made in Ada between methods that carry out the semantics of arithmetic and those whose semantics are unrelated. Both kinds of methods may be added to the primitive functions. For example, we might use overloading to extend Ada's "+" function to include methods for "`integer + float`" and "`string concatenate string`", as suggested in Exhibit 17.7. The first method belongs in the same generic family as the primitive methods. The second one is semantically unrelated and does not. The language permits any existing arithmetic operator to be overloaded, but new ones may not be defined.

Using the arithmetic function name "+" to symbolize something unrelated to addition (such as concatenation) is a questionable practice. We want to make a clear distinction between generic

[2] A short clarification of terminology seems necessary here to avoid confusion. Two kinds of generic objects are supported in Ada. In Ada references, the term "generic" refers to parameterized source-code modules, which are covered later in this chapter, and the term "overloaded" refers to operator names which have ambiguous meanings.

functions that implement a single abstract function and those that include methods for a mishmash of abstract functions. Hereafter, we will use the term *generic function* to mean a function name that represents a single abstract function over a generic domain. The term *overloaded function* will refer to a single name used to represent any collection of function methods, possibly having unrelated meanings. Pure generic functions have clean semantics and obey the principle of distinct representation (see Chapter 2.3); overloaded functions do not.

17.2.3 Fixed Set of Generic Definitions, with Coercion

"Number" is the best-known and most-used generic domain. It was recognized early in the history of computing that integers and reals (and in COBOL, BCD strings) are all representations of numbers, and that the programmer must be able to work with all kinds of numbers without worrying about type conversions. Thus the generic domain "number" has been built into most older languages. To say that this generic mode is built-in, we mean that the programmer is permitted to use objects belonging to both primitive types as if they belonged to the same domain. More precisely:

- Two or more representations of the external domain "number" are defined in the language as primitive types.
- Conversion functions are predefined that allow one- or two-way compatibility between those primitive types.
- The language syntax allows objects of the two types to be used together in contexts that were defined only for homogeneous pairs.
- The compiler will coerce an argument of one type to make it appropriate for a context that requires the other.

In FORTRAN 77, C, and Pascal, the functions "+", "-", "*", and "/" are generic functions, defined over the implicit generic domain "number" [Exhibit 17.2]. However, languages designed before the mid-1970s rarely permit the programmer to define new generic functions or even to define new methods for existing generic functions.

When generic functions are built into a language, they often interact with rules for type coercion. In the diagrams of "+" for both Pascal and FORTRAN, note the absence of a method for adding an integer and a real. These languages permit the programmer to write arithmetic expressions that involve a mixture of real and integer operands. However, these computations are not implemented by separately defined methods. Rather, they are implemented by a combination of the method for real addition and a coercion function which converts an integer to a real. The conversion from integer to real is meaningful because both domains can be used to represent the same external objects (whole numbers). These conversion functions can thus be invoked, without a change in the semantics of the argument, when they are needed to carry out the computation.

In contrast, Ada also supports the implicit generic domain "number" but not mixed-type arithmetic. The arithmetic operators are generic, and predefined methods for "+" include:

Exhibit 17.8. Definition of new methods for an Ada operator.

We can implement mixed-type addition by adding new methods to the predefined set. The new method works by explicitly converting one of the operands to be compatible with the type of the other, creating operands of the right types for a previously defined method. The first definition below defines the semantics for "3 + 5.2" but not for "5.2 + 3". The second definition is also required to permit "+" to be called with integer and float operands in either order.

```
function "+" (x:  integer, y:float) return float is
    begin return float(x) + y end "+";

function "+" (x:  float, y:integer) return float is
    begin return x + float(y) end "+";
```

```
function "+" (x,y:  integer) return integer;
function "+" (x,y:  float ) return float ;
```

However, Ada does not support representation coercion of any sort, so addition of an integer and a float is not predefined. It *is* possible for a programmer to define a limited number of mixed-type operations in Ada. This is explained more fully in Section 17.2.4.

A more elaborate built-in generic function is the COBOL assignment command, `MOVE`. Several definitions of `MOVE` exist which allow the large number of numeric and nonnumeric data types to be used together. `MOVE` is also invoked automatically in the process of performing arithmetic, input, and output.

17.2.4 Extending Predefined Operators

As in most languages, the arithmetic operators in Ada are generic, and the intended meaning of each operator is determined by looking at the types of its operands and selecting the method defined for those types. This very limited generic facility was included in Ada, as in most common languages, for convenience, so that the programmer could use more than one representation of numbers without needing to learn unfamiliar unique names for the operator methods.

Unlike many languages, however, Ada does not provide automatic type coercion to make a set of operands conform to the types declared in one of the available methods. Thus mixed-type operations are not predefined. To compensate for this lack, Ada permits more methods to be loaded onto the existing arithmetic operators[3] [Exhibit 17.8]. Each new method must involve a new combination of operand types. As a happy side effect, the overloading mechanism permits us to extend the basic arithmetic operators to work on programmer-defined types [Exhibit 17.9]. A separate function must be explicitly provided for every combination of operator and operand types that the programmer wishes to use. This can lead to quite a lot of definitions if more than one

[3] A similar facility was provided by the MAD compiler, Arden, Galler, and Graham [1963].

Exhibit 17.9. Extensions of Ada operators for new types.

This exhibit further develops the material in Exhibit 17.8. We extend "*" to operate on the mapped domain `float` and extend "+" to add two vectors.

```
type mass is new float;
type vector is array (1..5) of integer;

function "*" (x:  integer; y:mass) return mass is
    begin return mass (float(x) * float(y)) end "*";

function "+" (x,y:  vector) return vector is
    z:  vector;
    begin
        for k in 1..5 loop z(k) := x(k) + y(k) end loop;
        return z;
    end "+";
```

operator or more than two types of operands are involved. To write a large number of these would indeed be tedious!

Examples of new method definitions are given in Exhibit 17.9. The first declaration extends "*" to work on an integer and a new type represented by float. The domain structure built up here begins to be complicated, demonstrating a two-step relationship within a generic domain. To multiply a mass by an integer requires a demotion cast (mass-to-float) and a conversion (integer-to-float). Finally, a promotion cast (float-to-mass) is required to correctly label the type of the result.

The second function defined in Exhibit 17.9 extends the "+" operator to a compound type, vector.[4]

17.2.5 Flexible Arrays

It is not difficult to code a function that can process any length array of a given base type. Processing of "flexible arrays" is supported in some very old languages, for example both FORTRAN [Exhibit 17.10] and C [Exhibit 17.11]. These functions work because the method for summing the elements of a vector involves a process that is repeated for each vector element. The length of the vector is passed as a separate argument and used to stop repetition.

Flexible array parameters are implemented very simply and very similarly in C and FORTRAN. Array arguments are passed by reference in both languages. This avoids the need to allocate storage space for the argument in the function's stack frame, and the need to copy the contents of the array

[4] This is possible because Ada supports coherent passage of compound objects as parameters. An explicit declaration of `k` is not required; the `for` loop implicitly declares the loop variable.

Exhibit 17.10. An implicit parameterized domain in FORTRAN.

The letters at the right refer to the notes in the text; they are not part of the program.

```
      REAL FUNCTION SUMUP (AR, N)   (a)
      REAL AR(N)                    (b)
      SUMUP=0.0
      DO 10 I=1,N                   (c)
      SUMUP = SUMUP + AR(I)         (d)
   10 CONTINUE
      END                           (e)
```

into that stack frame. Neither language supports automatic run-time bounds checks for arrays. The lack of prior knowledge of the size of the array causes no problem for the compiler, since no compile-time decisions are based on it. To ensure correct processing, a flexible array argument must have a recognizable terminating value, or the array length must be passed as a separate argument.

In FORTRAN, the dimensions of an array parameter do not need to be declared, so long as they are also passed as parameters or declared in a `COMMON` statement as global variables. Processing can proceed correctly so long as the array length is known at run time. Exhibit 17.10 shows how a flexible array might be processed in FORTRAN. Following are the program notes:

a. `SUMUP` is defined as a function of two parameters that returns a real.

b. The parameter `AR` is defined as a real array of unknown length.

c. This line defines the scope of a loop (up to statement #10) and directs that it should be executed `N` times, with `I` taking on the values 1 to `N`.

d. This is not a recursive call. FORTRAN does not support recursion. Within a function, the function name serves as a local variable.

e. The final value assigned to `SUMUP` is returned at `END`.

Similarly, in C, arrays are passed by reference and subscript bounds are not checked at all. With multidimensional arrays, all dimensions except the first must be known at compile time in order to translate subscript expressions. However, the first (left-hand) dimension is not used in subscript computations. Thus the first dimension of an array parameter does not need to be declared in the function.

In Exhibit 17.11 we define a C version of the FORTRAN function in Exhibit 17.10. The following notes are keyed to the comments in the code. Line a declares the types of the formal parameters. Note that the length of array `ar` is not specified. Line b declares a local floating-point variable, `sum`, to be used as an accumulator. Lines c and d comprise a `for` loop that will be repeated `n` times, with variable `i` taking on the subscripts 0 to `n`− 1. This loop adds the `i`-th array element to the sum. Line e returns from the function with the final value of `sum`.

Exhibit 17.11. A parameterized domain in C: array of floats.

```
real sumup (float ar[], int n)          /* a */
{   int i;
    float sum;                          /* b */
    for ( sum=0.0, i=0; i<n; i++ )      /* c */
        sum = sum + ar[i];              /* d */
    return sum;                         /* e */
}
```

These are examples of implicit, not explicit, parameterized generic domains. However, they demonstrate that an implementation of integer-parameterized domains can be straightforward. A function with a "flexible array" parameter is, technically, generic, because it can accept arguments of many specific types. However, it does not cause the compilation and interpretation problems inherent in type-parameterized generics. The code for a flexible array argument can be compiled and type checked (but not bounds checked) because all the types involved are determined at compile time.[5]

Arrays with Explicitly Parameterized Bounds

The inclusion (or exclusion) of flexible array parameters in Pascal caused a great deal of dissension among the twenty members of the International Standards Organization committee that developed the standard for Pascal. In the end, array arguments with flexible bounds, called *conformant arrays*, were included in ISO Level 1 Standard Pascal, but not in the ANSI standard. American implementations of Pascal, therefore, generally lack this important facility.

One of the issues the committee argued about was run-time bounds checks, which are normally part of Pascal semantics. Rather than expecting the programmer to pass this information separately or implicitly, the syntax for a conformant array parameter indicates that the bounds are passed *as part of* the argument. These bounds are named in the formal parameter list and can be accessed within the code. Exhibit 17.12 shows how the function to sum a vector would be written in ISO Pascal. The conformant array parameter can be passed as either a value or a `VAR` parameter.

17.3 Parameterized Generic Domains

Varying the bounds or base type of an array or the type of a field in a record creates a collection of closely related types. If we write a type expression with dummy parameters in place of one or more of its fields, the result is a *parameterized type expression.* Consider the domain which is the union

[5] Automatic bounds checks cannot be done in C since the array bounds are not specified at compile time and are not supplied at run time in a form that the run-time system can access.

Exhibit 17.12. A Pascal function with a conformant array parameter.

Compare this ISO Pascal function definition to the C example in Exhibit 17.11. Note that the bounds of the Pascal array are automatically passed as parameters to the function; they are named and can be used in the code.

```
Function SumUp(VAR ar[ Lower..Upper ] of real):real;
Var i:  integer;
    Sum:  real;
Begin
    Sum := 0.0;
    For i:= Lower to Upper do
        Sum := Sum + Ar[i];
    SumUp := Sum
    End;
```

Here are some data declarations and two appropriate calls on the `SumUp` function.

```
Var Scores:  array[1..3] of real;
    Charges:  array[1..1000] of real;

    Answer := SumUp(Scores);
    Answer := SumUp(Charges);
```

Exhibit 17.13. A parameterized type.

We define a type with two parameters, an integer and a type, then instantiate it to produce two specific subtypes. (The syntax used is an extension of Pascal.)

Intent: A buffer for items of type TT

Parameterized type expression:

```
TYPE BUF(m:integer, TT: type) = array [0..m] of TT
```

Specific instances derived by instantiation:

Call	Resulting specific type
`Buf(5, real)`	`array [0..5] of real`
`Buf(10, integer)`	`array [0..10] of integer`

Exhibit 17.14. A parameterized linked list type.

Intent: A linked list cell.

Parameterized type expressions:

```
Type cell_ptr = ↑cell;
Type cell(DD:type) = record data:  DD; next:  cell_ptr end;
```

Specific instances:

Call	Resulting specific type
`cell(integer)`	`record data:integer; next:cell_ptr end`
`cell(mytype)`	`record data:mytype; next:cell_ptr end`

of all the domains associated with a parameterized type, for all possible values of the parameters. We call this a *parameterized domain*. Exhibit 17.13 shows a parameterized type and two domains which have been derived from it by instantiation.

Exhibit 17.14 shows a mutually recursive pair of types which include a parameterized type and a pointer type which depends on it. The two domains formed by these types are both generic. Type `cell` has explicit parameters. Type `cell_ptr` is defined in terms of `cell` and thus "inherits" the generic nature of `cell`.

A parameterized type expression is not compilable code as it stands. Rather, it is a template from which code can be created by instantiation. In an instantiation call, the programmer supplies actual, specific arguments. When this call is processed, the arguments are substituted for the dummy parameters in the type expression, according to the lambda calculus substitution rule. The result is an ordinary type expression. Instantiation happens during an early phase of compilation, before parsing. To use a parameterized generic type, the programmer would create a parameterized source code module containing the parameterized type expression(s) and related function definitions.

With a type-parameterized generic, the processing method is not quite independent of the type of an argument. For example, the code for `pop` and `push` on a stack of reals is almost the same as the code for a stack of integers, except that a different number of bytes of information will need to be fetched, stored, or returned to the calling program. Similarly, matrix multiplication is the same process whether operating on a matrix of reals or integers, except that appropriate methods for the "`*`" and "`+`" functions must be used.

If a function were defined for several species of one type-parameterized domain, the source code for the various definitions would very likely be identical except for the types of the parameters. (Compare the code for `push` in Exhibits 17.5 and 17.15.) The object code, however, is not identical, because the compiler uses the type declarations to compile appropriate methods for the built-in generic functions such as fetch, store, "`=`", "`*`", and "`+`". Thus the types of objects must be known before the code can be fully compiled.

When a programmer writes specific type declarations and code that uses them, we say that the

Exhibit 17.15. Pascal definition of push on a stack of characters.

Here are type declarations and the function push for a stack of 15 characters. Note that the code between BEGIN and END is the same as in Exhibit 17.5; only the type declarations have changed.

```
CONST len = 15;
TYPE char_stack = RECORD
        store:  ARRAY [1..len] OF char;
        top:  0..len  END;

VAR rs:  char_stack;
    value:  char;

FUNCTION push (VAR rs:char_stack; r:char):boolean;
BEGIN
    IF rs.top = len THEN push := false (* error flag *)
    ELSE BEGIN
        push := true;
        rs.top := rs.top + 1;
        rs.store [rs.top] := r
    END (* IF rs.top *)
END; (* push *)
```

data types are *bound at source time*. This is the only programming style supported by languages in the Pascal family, and it does not permit use of user-defined generic domains.

17.3.1 Domains with Type Parameters

We should distinguish between an integer-parameterized domain (a flexible array) and a type-parameterized domain, which has a type parameter like the type cell in Exhibit 17.14. Few existing languages permit component types to be parameterized, while several languages permit functions to be written with implicit or explicit flexible arrays.

An ADT is a collection of types, functions, and objects that express an abstract process on abstract data. When Pascal programmers wish to use an ADT such as “stack” or “linked list”, they start by writing type declarations for their own data and for the ADT; they then write out definitions for the ADT functions, making adjustments to the function headers so that they are compatible with their own types. If two variants on the generic domain are needed, each part of the ADT must be written out twice.

Rewriting and re-debugging the functions for a common ADT is tedious but not difficult. The code for each new variety of stack is so similar to the code for other varieties that code can be copied out of a reference book with minimal modification. This leads us to ask whether it is possible to automate the process of coding up a new version of an ADT. We would like to keep the ADT code

in a library, so that it does not need to be reentered manually each time it is used.

Many ADTs can be expressed as sets of parameterized type definitions and functions with parameters of those types. Definitions of this sort can easily be kept in a library, but we need a way to relate the type parameters in the library modules to the types in a user's program. Also, the function headers for the ADT must be made compatible with the user's own data types.

The easiest way to achieve these goals is by adding a preprocessor to the language translator. The ADT is coded as a parameterized module that contains type and function declarations written in terms of the generic parameters. These symbols will be bound to specific meanings at precompile time by appropriate preprocessor commands. A library of parameterized source code modules could be made available to programmers to include and instantiate, as needed. Before compiling, the preprocessor is used to supply specific meanings for the generic parameters. The source code package is then expanded, like a macro, using these symbol definitions. All parameterized type declarations, function definitions, and object declarations in the package are expanded to form normal nonparameterized instantiations.

17.3.2 Preprocessor Generics in C

Exhibit 17.16 shows how one might use C to write code for a generic sorted-linked-list type. Such a package would contain definitions for several linked-list functions, including `insert_item`, `delete_item`, and several others. For simplicity, only one of these functions is shown here. (The identifiers written in uppercase are symbols that will be defined and instantiated by the preprocessor.)

The generic function `find_item` was written in terms of the generic type names and the generic functions `GREATHAN` and `EQUALS`, which must be defined before expansion. The functions of dereference (`*`), selection (`->`), and assignment (`=`) are also used, but these are defined by the nongeneric part of the type definition. Nothing in the function code depends on the particular type to which `ELEMENT` will be bound; an `ELEMENT` could be a single character or a lengthy record.

To use the generic list package in Exhibit 17.16, the user must include in the program definitions of `ELEMENT` and of the associated type-dependent operations. In all, this is several lines of routine and tedious code. A convenient way to automate the inclusion process is to put the code for several commonly useful types into a header file and use conditional compilation to include only the appropriate set of definitions.

A header file is given in Exhibit 17.17 that would permit the generic linked-list module to be used for linked lists of character strings or of integers. If some other type of data were to be used, a group of similar definitions would have to be written for it and included in another conditional clause. To use this header file, the programmer must type one of the following pairs of lines at the top of the code:

For lists of strings	**For lists of integers**
`#include "lists.h"`	`#include "lists.h"`
`#define ALPHA 1`	`#define INTEGER 1`

Exhibit 17.16. Definition of a generic domain in C.

Below are type declarations for a generic type "sorted linked list" and one function whose domain is this type. Each list cell consists of one data item and a pointer to the next list cell. The generic symbols are written in uppercase letters and will be defined as macros by the programmer and expanded by the C preprocessor before actual compilation. Exhibit 17.18 shows the result of expanding this code with one set of symbol definitions shown in Exhibit 17.17.

```
typedef struct cell {ELEMENT data; struct cell *next;} CELL;
typedef CELL * LIST;

/*-----------------------------------------------------------*/
int find_item(ELEMENT find, LIST head, LIST *scan, LIST *prior)
{
    scan = head;
    prior = NULL;
    while ( GREATHAN( find, (*scan)->data))
    {   *prior = *scan;
        *scan = (*scan)->next;
    }
    return EQUALS( find, (*scan)->data );
}
```

The value of the `#define` symbol is tested by the preprocessor's `#if` command, which triggers inclusion or exclusion of the dependent definitions. During preprocessing, the generic package is converted to a set of ordinary declarations and definitions which define and process a specific type. Exhibit 17.18 shows the ordinary code that results from expanding these generic definitions for type argument `char*`.

Ada Generic Packages

We can implement an ADT in Ada as a *generic package*. A package is a module that contains type, data, and/or function declarations.[6] An Ada *generic* is a parameterized template which can be instantiated to create a subprogram or a package. We will examine a simple generic subprogram first, then tackle the problem of combining a generic with a package.

Parameters to a generic subprogram can be types, array lengths, or function names. The generic parameters are used as follows:

- The integer parameters to the package are used as array bounds in parameterized type expressions within the package.

[6] Ada packages were described in Chapter 16.

Exhibit 17.17. Using a generic type in C.

This is a header file that would permit the generic linked-list module to be used for linked lists of character strings or of integers.

```
/* The function "strcmp" performs alphabetic comparison of two strings, */
/* "strcpy" copies a string value into a string variable.  */
/* \377 is the octal code for the largest 1-byte character.  */

#if ALPHA
    #define ELEMENT char *
    #define LESSTHAN(x, y)  ( (strcmp((x),(y)) <0 )
    #define GREATHAN(x, y)  ( (strcmp((x),(y)) >0 )
    #define EQUALS(x, y)    ( (strcmp((x),(y)) == 0 ))
    #define ASSIGN(x, y)    ( strcpy((x),(y)) )
    #define MAXVAL          '\377'  /* Octal for char code 255.*/

#elif SHORT
    #define ELEMENT short int
    #define LESSTHAN(x, y)  ( (x) <(y) )
    #define GREATHAN(x, y)  ( (x) >(y) )
    #define EQUALS(x, y)    ( (x)==(y) )
    #define ASSIGN(x, y)    ( (x)=(y) )
    #define MAXVAL          0x7fff
#endif
```

- The type parameters are also used in parameterized type expressions. If the data structure within the package uses two interdependent types, such as "cell" and "cell_pointer", both must be passed as parameters.
- A generic must have one functional parameter for each representation-dependent function or operation that is used in the body of the generic.

Exhibit 17.19 gives an Ada generic definition for addition on length-three numeric vectors. Such a definition might be part of a generic package and be included in a library. Let us examine each part of the generic code in Exhibit 17.19 so that we may see how these parts work together.

We begin definition of a generic subprogram by declaring the generic parameters (written here in capital letters). Line (a) declares a dummy type name, `NUMBER`. The predefined generic domain named `private` is used for any type parameter for which assignment and tests for equality are defined. Thus, this generic definition can be instantiated with arrays of `integer`, `float`, or any other type which permits assignment and tests for equality.

Exhibit 17.18. Specific code results from preprocessing.

This function is the result of including the header file in Exhibit 17.17 with the symbol `ALPHA` defined to be 1, and expanding the parameterized code in Exhibit 17.16.

```
typedef struct cell {char * data; struct cell *next;} CELL;
typedef CELL * LIST;

/*----------------------------------------------------------*/
int find_item(char * find, LIST head, LIST *scan, LIST *prior)
{
    scan = head;
    prior = NULL;
    while ( strcmp( (find),((*scan)->data) ) > 0 ) {
        prior = *scan;
        scan = (*scan)->next;
    }
    return (strcmp( (find), ((*scan)->data) ) == 0);
}
```

Exhibit 17.19. A parameterized domain in Ada: vector of numbers.

Step 1 is definition of the generic template:

```
generic
    type NUMBER is private;                                    -- (a)
    type VECTOR is array (1 ..  3) of NUMBER;                  -- (b)
    with function PLUS (x,y:  NUMBER) return NUMBER;           -- (c)
    procedure vector_add (v1, v2:  in VECTOR; v3:  out VECTOR);  -- (d)

procedure vector_add (v1, v2:  in VECTOR; v3:  out VECTOR) is    -- (e)
    begin
        for J in 1..3 loop v3(J) := PLUS( v1(J), v2(J) ) end loop;
end vector_add;
```

Ada provides a very small collection of predefined generic domains. In addition to `private`, generic domains are defined for integer types, floating-point types, fixed-point types, finite types, and pointer types, and for the completely general type with no known properties. Ada programmers are not allowed to define their own generic domains; each generic type parameter must be declared to belong to one of the predefined domains. The programmer, therefore, selects one that comes closest to the properties of his or her own application's domains. We use the type `NUMBER` in this example, which is a super-domain of all the numeric generic domains that are predefined. We, therefore, declare `NUMBER` as `private`.

A second type parameter, `VECTOR`, is declared on line (b). Even though `VECTOR` is defined in terms of `NUMBER`, it must be passed as a separate parameter. This certainly makes the preprocessor easier to implement but is annoying for the programmer.

The code of the generic routine cannot "just use" any operations except assignment and equality-comparison, since these are the only functions that are "guaranteed" for the domain `private`. Often this is no problem, as the code of many functions relies only on these two basic operations. For example, only assignment is needed to define the "push" and "pop" functions for stacks. However, the code for `vector_add` uses both assignment and "`+`" for `NUMBERs`. To do so, "`+`" must be declared as a parameter (line c).

The generic procedure header must be declared before the actual definition is given. Line (d) uses the parameter names to declare the function named `vector_add`. There are three parameters, two vectors to add up, and one to receive the answer. On line (e) we finally begin definition of the actual procedure. The syntax is the same as an ordinary nongeneric procedure, except that it uses the generic type parameter(s).

A generic procedure or package must be instantiated before the code can be compiled. Prior to instantiation, the programmer must write a specific type definition for each type parameter and a specific function method for each of the package's functional parameters. Finally, the programmer must write an instantiation command using these predefined specific objects as arguments. The instantiation call is given as part of a declaration for the name that will be bound to the instantiated procedure or package. This name must be a new, unique name. To instantiate a generic procedure or package we write:

procedure ⟨`procedure name`⟩ **is new** ⟨`generic name`⟩ **(** ⟨`specific argument list`⟩**);**
package ⟨`package name`⟩ **is new** ⟨`generic name`⟩ **(** ⟨`specific argument list`⟩**);**

During the instantiation process, the compiler substitutes the actual arguments supplied by the programmer for the generic parameter names and expands the template into ordinary, fully specific, compilable source code. This is then compiled and bound to a unique function name supplied by the programmer. This name refers to exactly one procedure or package and is not generic or ambiguous at all. If the programmer wishes to instantiate the same package twice, she or he must supply two names for the results.

To instantiate the `vector_add` template we must first define any new specific types and functions needed for the parameters `PLUS`, `NUMBER`, and `VECTOR`. For example, to create a vector of floats, type `float` and floating-point addition are already defined, but we need to declare a new type consisting

Exhibit 17.20. Instantiating a generic template in Ada.

```
type real_vec is array (1 ..  3) of float;                           -- (f)
type int_vec is array (1 ..  3) of integer;

procedure int_vector_add                                             -- (g)
    is new vector_add(NUMBER=>integer, VECTOR=>int_vec, PLUS =>"+");
procedure real_vector_add                                            -- (h)
    is new vector_add(NUMBER=>float, VECTOR=>real_vec, PLUS =>"+");
```

of an array of three floats. Then we write an `is new` declaration containing a call on the template with these actual arguments.

In Exhibit 17.20, we define two length-three numeric array types—`int_vec` and `real_vec`. We use each of these types to instantiate the generic from Exhibit 17.19. In line (g) we instantiate the `vector_add` template. We indicate that the type parameter `NUMBER` is to be replaced by the type `integer`, `VECTOR` is to become `int_vec`, and `PLUS` is the standard “+”. The translator will make these substitutions and generate ordinary, compilable, nongeneric code for a procedure named `int_vector_add`.

We repeat the instantiation process in line (h) with types `float` and `real_vec`, producing the nongeneric procedure named `real_vector_add`. Now we have two ordinary procedures that are alike except for their names and the types of their parameters. The object code will contain translations of both copies. Note that no definition of the function argument was given; when this is done it defaults to the definition (if any) for that symbol in the context surrounding the generic definition.

A Generic Package. We have described both packages and generics in Ada. The normal way to use both generics and packages is by combining them—a package definition is placed inside a generic declaration, producing a generic package. Such a package has parameterized public and private parts that can be instantiated with various component types and array lengths to produce code tailored to an individual application. The elements that must be present in an Ada generic definition are:

1. The generic parameter declarations.
2. The package header, containing declarations of the public symbols
3. The package body, containing definitions of public and private symbols.

As an example, we show in Exhibit 17.21 how the code for the stack package in Exhibit 16.6 can be generalized to a parameterized generic module. First, the entire package is nested within a generic declaration. Then all references to the base type of the stack and the length of the stack are replaced by references to the generic parameters.

Exhibit 17.21. Declaration of a generic stack package in Ada.

```
generic
    MAX: POSITIVE;                -- A positive integer.
    type ELEMENT is private;      -- Type of a generic parameter.
package STACK is                  -- Declare functions in package.
    function PUSH(X: ELEMENT);
    function POP return ELEMENT;
    function TOP return ELEMENT;

package body STACK is             -- Functions, variables for package.
    stk:array(1..MAX) of ELEMENT;
    tos:  INTEGER range 0..MAX;

    function PUSH(X: ELEMENT) is
    begin
        if tos<MAX then
            tos:=tos+1;  stk(tos):= X;
            return true;
        else return false;
        end if
    end PUSH;

    function TOP return ELEMENT is
    begin
        if tos >0 then return stk(tos);
        else raise STK_FULL;      -- Exception error condition.  end if;
    end TOP;

    function POP return ELEMENT is
        ans:  ELEMENT;
    begin
        ans:= TOP;  tos:=tos-1;
        return ans;
    end POP;

begin
    tos:=0;                       -- Initialization for this package.
end STACK;
```

Exhibit 17.22. Instances of the stack package.

In writing a program to do a precedence parse and code generation for arithmetic expressions, one needs to use two stacks: a stack of tokens and a stack of expression trees. Here we instantiate the generic stack package from Exhibit 17.21 twice to create these two stacks. For simplicity here, assume that tokens are represented by single characters, and that the function named `do_error` is the programmer's own error handler and is defined previously.

```
type TOKEN is CHARACTER;
type TREE is access CELL;
type CELL is record
    data:  TOKEN data;
    lson:  TREE;
    rson:  TREE;
end record;

declare
    package tok_s is new STACK(20, TOKEN);
    package tree_s is new STACK(60, TREE);

begin
    if not tok_s.PUSH('#')        -- Push beginning-of-line symbol.
    then do_error("Stack full.") -- Check error return.
    endif;
    ...
```

We can instantiate this package, as shown in Exhibit 17.22, to create multiple stack packages, each with its own storage and its own copy of the code for each function. Within this code, calls on `PUSH`, `POP`, and `TOP` will be compiled with meanings for the fetch and store operations that are appropriate for the actual type of the type-argument. Note that we give a name to each specific package in the instantiation command. When we call the `PUSH` that belongs to a package, the name of the package is used, with "`.PUSH`", to denote the proper method for `PUSH`. The equivalent of the stack package in Exhibit 16.6 could be created by instantiating `STACK` with the parameters `(10, INTEGER)`.

Evaluation

An abstract data type (ADT) is a collection of types, functions, and, possibly, objects that express a generic process on generic data. An ADT can be defined in C by using the preprocessor and in Ada

by using a generic package. Ada's generic preprocessor is very much like the C macro preprocessor.[7] Thus the capabilities of Ada generics are very similar to those in C.

These preprocessor generics are relatively easy to implement and make real progress toward the goal of creating practical, flexible code libraries. However, they are far more limited than the support for generic types provided in various object-oriented languages. The primary limitation is that final binding of the type parameters happens at precompile time. Thus all type flexibility is lost before compilation begins. If a program deals with a domain that has two or more representations, there must be two or more instantiations of the package, and two or more names for those instantiations.

It is much more difficult to support code with run-time generic flexibility. Most languages which do so are translated by interpreters, not compilers. To support runtime generics, the type of each object must be known at run-time, and the translator must include a sophisticated method-dispatcher which examines those types at run time and selects an appropriate method. We will discuss run-time dispatching in Chapter 18.

Exercises

1. Contrast generic domains and specific domains.
2. Describe the four ways in which a domain may be considered generic.
3. What are the problems involved in the implementation of generics?
4. What is a virtual function? What is its role?
5. What are the universal generic functions? Explain.
6. What semantic problems are created by the generic function?
7. What is a method? What is dispatching the call?
8. Why must functions such as "push" and "pop" be defined by separate methods for the species of an ad hoc generic domain?
9. Why is generic behavior considered "limited" in older languages?
10. What is a union data type? What are its limitations?
11. What is an overloaded name? What are the dangers involved in overloading?
12. How do built-in generic functions interact with rules for type coercion?

[7]There is one major difference in name scoping. Unbound names in a C macro will be interpreted in the context of the macro call. In contrast, unbound names in an Ada generic function will be interpreted in the lexical context of the module in which the generic was defined, not the module containing the instantiation request.

13. How do Ada's predefined operators limit mixed-type operations? How does Ada compensate?
14. What is a flexible array? Why is a function with a flexible array parameter considered generic?
15. What is a parameterized domain? A parameterized type expression?
16. What is the difference between an integer-parameterized domain and a type-parameterized domain? Why do some languages support the former but not the latter?
17. What is the role of the preprocessor in relating the types in library definitions to the types in a users's program?
18. What is a generic package in Ada? How are generic packages related to ADTs?