

Chapter 18

Dispatching with Inheritance

Overview

Types have traditionally been used for the two disparate purposes of describing storage and describing semantics, and this dual use has led to compromises in the power of the language to express type relationships. One approach to the conflict generated by the two needs is to eliminate or subordinate one use. If type checking for function applicability is not done, the language never stands in the way of the programmer, never prevents programmers from doing things they know are sensible or exploiting relationships they know exist. This approach is taken in C. Types are used at the bottom level to allocate, access, and encode but are not used to describe the semantics of classes of objects.

On the other hand, we have languages such as Smalltalk where all objects are represented as pointers. In Smalltalk the type (class) is used primarily to control function application, and data representation is simplified into near uniformity.

Future languages will separate the storage mapping of an object and its domain membership, permitting them to be defined separately. Doing so will enable us to retain the safety of type checking within a language that can exploit domain relationships. A mode graph gives us a systematic way of representing domain relationships and type conversions.

The goal is a language in which the programmer can manipulate abstractions of both objects and functions. This requires that we be able to describe objects and related classes of objects, actions, and variants of those actions. Then the translator must use that information to dispatch functions and coerce arguments.

Implementation of these mechanisms opens up new problems and leads to the need to

define parameter domains using type expressions with constraints, rather than simple type constants. The semantic basis of a generic language must include a general type-deduction or type-calculation system.

18.1 Representing Domain Relationships

In the familiar strongly typed languages, each type declaration creates a new domain, and it is not possible to define two structurally dissimilar types to represent related external domains, then further declare and use the relationship between them. In these languages, types are used both to define the physical representation of an external domain and to define the semantic properties (function applicability) of program objects. Some domain relationships are built into programming languages. For example, most languages have a few conversion routines that can be used by the compiler, when needed, to make sense of the code. However, most languages do not provide a way to express relationships among user-defined domains.

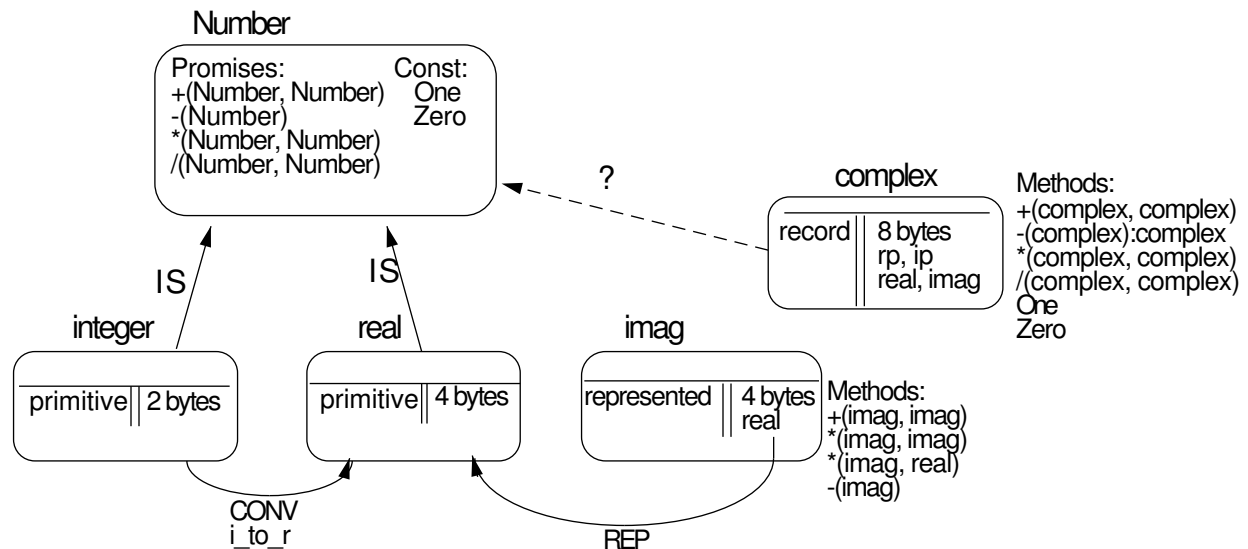
Sometimes a programmer's newly defined domains are unrelated to other domains used in a program. In this case it is not a burden to define a new set of functions for each newly defined domain. However, a programmer's domains are often related and share common properties with each other. Some functions might operate correctly on objects from a variety of related external domains. In the traditional strongly typed language, the programmer must represent all related external domains as variants of one type if they are to be processed by the same set of functions. Generic functions and domains have been developed to solve this dilemma: they permit each variation of a domain to be defined as its own subdomain, and then the relationships among the subdomains are separately declared and used by the translator.

The class hierarchies in the common object-oriented languages (Smalltalk, C++) are tree-structured. The only class relationship supported is the one created by deriving one class from another, and each class can only be derived from one parent. Generalized graphs of classes are not possible in these languages because the objects of a derived class are represented by adding fields to the record type used to represent the parent class. But it is possible to use virtual functions, not class derivation, as the basis for declaring class relationships, and once we do this, we can build a generalized graph of classes.

18.1.1 The Mode Graph and the Dispatcher

In this discussion, we will use a structure called a *mode graph* to discuss how both domain relationships and dispatching can be generalized. We call one node in the graph a *mode*, and diagram it as a round-cornered box, like the box labeled "Number" in Exhibit 18.1. The contents of a mode box vary according to the type of the mode: generic mode boxes contain a list of virtual functions, specific mode boxes contain a type object. Methods defined for a mode are sometimes listed near

Exhibit 18.1. Mode graph for the generic domain “Number”.



it. We will consider typed modes, parameterized modes, subdomain modes (or submodes), generic modes, and representation modes. As we discuss modes, we will point out the aspects of this mode graph that have been incorporated into modern languages.

The *dispatcher* is a process built into a generic programming language that uses the information in the mode graph to dispatch methods for function calls. You may visualize it as a mouse that crawls along a maze of tunnels (the mode graph) looking for a method definition that will satisfy the call. Depending on the language, the dispatcher has more or less information available to it, and has more or less freedom to move from one domain in the graph to another, related domain.

The job of dispatching potentially has two components: work that can be done at compile time because all relevant information is available then, and work that must be deferred until run time. We will distinguish these two job components by talking about the *compile-time dispatcher* and the *run-time dispatcher*. Traditional languages have only a primitive compile-time dispatcher. Modern languages, including object-oriented languages and functional languages, have an extensible compile-time dispatcher and a run-time dispatcher.

Typed modes. A typed mode is used in the mode graph to represent nonpolymorphic types, such as `integer` or a record type. Every typed mode has an associated type object.¹ Actual objects all belong to some typed mode, and all function translation and dispatching must start with the typed mode of the argument.²

¹Review Chapter 14 if necessary.

²Just as object-oriented dispatching starts with the class of the implied argument.

Generic Modes and Promises. A generic mode represents an abstract domain. Its meaning is defined by a set of virtual functions, rather than by a representation. A generic mode is similar to a C++ base class but differs in two important respects. First, a C++ class has an associated representation, from which other representations may be derived; further, derivation is the only way to create an additional representation of a class. In contrast, a generic mode has no representation when it is first created, but typed modes and other generic modes may be attached to it later. The typed modes are representations of the generic. They might be structurally related, like the instances of a parameterized domain, or might have an arbitrary, ad hoc relationship to each other, like `integer` has to `real`.

In an object-oriented class hierarchy, both the fields of the representation record and the functions that operate on the record are inherited. It is this inherited relationship that guarantees that the code will be semantically meaningful when it is executed. In a mode graph, the fields of one mode are not inherited by the related modes, and other means must be provided to assure that the operation of the program will be semantically meaningful. Virtual functions can provide this assurance. The box for each generic mode will contain a set of virtual function declarations which we will call *promises*. The promises of a mode will be used, like virtual functions in C++, to describe the implementation-dependent processes that characterize the semantics of the mode. Any functions defined for the generic mode itself must be written only in terms of its promises. Exhibit 18.1 shows a diagram for a generic mode `Number`. In it, promises state that numbers must have four defined arithmetic operations and also have constants defined for the arithmetic identity values, one and zero.

The generic facility in `Ada` has six predefined modes which can be used to declare the types of arguments to generic packages. For example, the mode “`private`” promises only that assignment and test for equality will be defined. Some other `Ada` modes are “`range <>`”, meaning some integer type, and “`digits <>`”, meaning some floating-point type. In our terminology, short integers, unsigned integers, long integers, and subranges of integers are all submodes of the predefined `Ada` mode “`range <>`”. In turn, “`range <>`” is a submode of the mode “`(<>)`” (which includes any discrete type).

`Ada` would be a better language if it provided a way for a programmer to declare new modes with different promises. Having only six modes available, as in `Ada`, is severely limiting. If the properties of a programmer’s domain do not happen to match one of the predefined modes, the programmer must use some other mode with fewer promises, or `limited private` which has no promises at all. If a function is needed which should be defined for the desired abstract mode, that function must be passed as an argument in the call that instantiates the package. This is a nuisance, at best, and half defeats the purpose of having generics.

Submodes. A mode graph can be used to represent a variety of domain relationships by introducing varied kinds of links from one mode to another. The links corresponding to `Ada` modes and to the “derived from” relationship in C++ are called *submode* links and are labeled by the word `IS`. A submode is one representation of its base mode and inherits function methods from it. In

Exhibit 18.1, the modes `integer` and `real` are shown as submodes of `Number`.

An ad hoc generic domain includes two or more specific domains that are related by their meaning rather than by their structure. Physically, the two domains might be represented differently, but logically, they have a common intent and common functionality. An `IS` link expresses the subdomain relationship that exists between a generic domain and its representations.

In some languages, operations are defined only for specific domains, and the `IS` relationship is not exploited. In contrast, consider `APL`, whose designers expected programmers to have infrequent concern about the representation of numbers and frequent use for the semantic relationship among the representations in use. The generic domain “`Number`”, not one of its specific representations, was made primitive. Further, the relationship between the generic domain and its subdomain was made explicit and flexible.³ Many `APL` operations are defined for numbers, rather than for either integers or reals. Others, which require the semantics of a discrete type, are defined only for integers.

Declaring and Using Mode Relationships. The set of promises on a mode provide a semantically sound criterion for being a submode. Thus if modes, promises, and links were all declarable, we could declare M' to be a submode of M as soon as all promised methods were defined for M' . In a generalized graph structure, a mode might have several supermodes, so these methods might be defined directly for M' , or might be inherited by M' down some other submode link. In our example, there is no problem with declaring that `integer` and `real` are submodes of `Number`. But before the submode link for `complex` can be declared, the promised arithmetic functions and constants must be defined. Once this is done, we can declare that `complex IS Number`. Now the submode `complex` will inherit the functions defined for `Number`, just as the predefined submodes `integer` and `real` do.

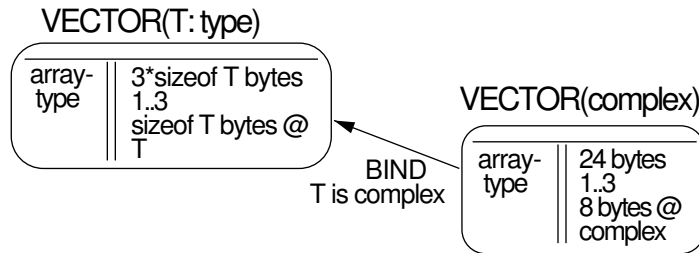
The promises defined for a generic mode guarantee that the promised functions will be defined for every specific instance of the generic mode. This permits us to write representation-independent functions whose domain is the generic mode. When one of those functions must access a representation, it calls a promised function.

When we declare that a specific mode `IS` a submode of a generic, all functions defined on the generic mode are inherited by the submode. That means that a programmer can call a function `F`, defined for the generic mode `G`, on an argument from the submode, `S`. If `F` calls one of the promised functions, the dispatcher must find the appropriate method to carry out the promise. To do this, it must look for and dispatch the method that is applicable to the submode `S`. Often, this dispatching can be done at compile time, but in the most general case, the mode `S` is not known until run time, and dispatching must be done at the last minute.

Instantiation Modes. Parameterized generic domains were discussed in Chapter 17, Section 17.3. The `C` and `Ada` implementations discussed there both require instantiation of the generic

³In `APL`, a number is an integer if and only if it differs from an integral value by no more than the currently defined comparison tolerance.

Exhibit 18.2. Diagram of an instantiated mode.



mode and its dependent functions at compile time. The result is that a set of specific types and functions are generated and compiled, and the fact that these were derived from a generic is forgotten. The mode graph can be used to represent the relationship between a parameterized generic domain and an instantiation of that domain. We will diagram this relationship as a BIND link, labeled by the bindings for the generic parameters that were specified in the instantiation call. A BIND link is illustrated in Exhibit 18.2. We will explore the question of delaying binding until run time and consider the advantages and problems caused by that delay.

Conversions and Coercions. Not every one-argument function is a conversion or a cast. Casts arise only from mapped domains and are simply part of the strong typing system of the language. Conversions are defined ad hoc, and only the programmer can know which of his or her functions are legitimate conversions. The important aspect of a conversion is that it must not change the meaning of an object. Defining a conversion can be done in any reasonable language, but most languages do not provide a way to tell the compiler that the function is, indeed, a conversion. C++ permits the programmer to define and declare conversion functions (called constructors), and languages that are now at the stage of research will incorporate more general provisions for declaring conversion functions. Once the compiler knows about the conversion, its dispatcher can use it to coerce arguments, just as the built-in conversions are used.

Without some provision for coercion, domains with multiple representations are unwieldy and almost impractical to use. We can demonstrate the problem in *Ada*. *Ada* permits the programmer to define new domains, then define new methods for the existing generic operators that operate on the new domains. The definitions of “+” and “*” for the new domain `imag` are an example [Exhibit 17.9]. But even though we can extend the intrinsic operators, it is a tedious job to achieve the ease of use of mixed-type arithmetic that is built into *Pascal*, *FORTRAN*, and *C*, because *Ada* lacks any provision for type coercion. To implement mixed-type arithmetic, the programmer must include definitions for every combination of operator and operand types desired [Exhibit 18.3]. Thus if F binary functions are to be defined over a generic domain with R representations, the number of method definitions needed would be $F \cdot R^2$!

FORTRAN actually supports these four numeric types and supports mixed-type arithmetic on

Exhibit 18.3. Exponential explosion of method definitions.

Assume that a programmer is using four representations of the domain Number: integer, float, double, and complex. To implement full mixed-type arithmetic for these four types, without type coercion, 16 method definitions must exist for each operator. Thus for “+” we would need:

+(int, int)	+(float, int)	+(double, int)	+(complex, int)
+(int, float)	+(float, float)	+(double, float)	+(complex, float)
+(int, double)	+(float, double)	+(double, double)	+(complex, double)
+(int, complex)	+(float, complex)	+(double, complex)	+(complex, complex)

For four arithmetic operators, we would need 64 methods!

all combinations of operands. However, FORTRAN does not have 64 method definitions for these functions. It has only 16 methods (four each for four operators), plus the conversion functions listed in Exhibit 18.4.

Type coercion between alternate representations of the same external domain reduces the number of function method definitions necessary. That is why PL/1 provided coercions from any basic type to any other, directly or through a series of changes. This coercion facility was completely general. Unfortunately, it masked real type errors by changing the type of an erroneous argument to whatever would seem to work. Sometimes this was beneficial, but sometimes it made semantic hash out of the data. It is obviously impossible to create a fixed set of conversions that will handle all mode relationships for all time in a semantically valid manner. When PL/1 tried to anticipate all future conversion needs, and used these conversions to coerce arguments, it resulted, all too often, in converting an object to something else with an unrelated meaning. Worse, coercion errors were especially difficult to track down because the semantically erroneous action was not caused by anything explicitly written in the program, but by some subtle and poorly understood type conversion built into the system.

The only adequate solution is to permit programmers to define their own conversion functions and to declare them in such a way that the compiler can use them for coercion. It would then be the programmer’s responsibility to define only semantically valid conversions. We will diagram a conversion function, F , that converts from mode $M1$ to mode $M2$, as a link from $M1$ to $M2$, labeled with “CONV” and the function name. In Exhibit 18.1, a conversion link is shown from integer

Exhibit 18.4. Coercions built into FORTRAN.

Result Type	Functions Defined
integer	INT(REAL), NINT(REAL), INT(DOUBLE)
real	REAL(INT), REAL(DOUBLE), ABS(CMPLX)
double	DBLE(INT), DBLE(REAL)

to real, for a function named “`i_to_r`”. This means that `i_to_r` can be used explicitly or through coercion to convert an integer to a real.

Represented Modes, Casts, and Coercions. Both conversion functions and casting functions take a single argument from one domain and return a value from a different domain. When the argument and result are both representations of the same object in the same external domain, the function is a conversion.⁴ When the argument type was defined by mapping it onto the result type, or vice versa, the function is a cast.

A cast represents the relationships between two domains. Casts are used, in some languages, by function dispatchers. Thus we want to represent them in our mode graph. A cast relationship is created when the programmer specifies that one mode is to be represented by another. We show it in the mode graph by noting that the type of a mode is a represented type. For example, there is a `REP` link from `imag` to `real` in Exhibit 18.1, because real numbers are being used to represent type imaginary.

Although definition and use of a conversion is often similar in syntax to a cast, the difference is very important: a conversion preserves semantics, a cast changes them. Therefore, a conversion is “safe” for coercion, a cast is not. A cast should never be applied unless the programmer explicitly directs it, and even then, the uses should be restricted to the private parts of classes or their equivalent.

Dispatching

We can view a function as a collection of methods, and dispatching as the problem of choosing the best method for each call. Simply put, the best method is the one whose parameter domains make the best match for the types of the actual arguments. In older languages, only a few predefined operators had more than one defining method, and the “best match” was easy to define; a method is best if:

- Its parameter types match the argument types exactly, or,
- No method matches the argument types exactly but for some method, every nonmatching argument can be coerced, by a built-in conversion function, to match exactly.

The dispatching problem is much more complex in a modern language. First, user-defined functions, as well as predefined operators, have multiple defining methods, so the dispatcher must be written in a more general way. Second, user domains have declarable relationships to other user domains; where a subdomain or an instantiation relationship exists, it should be used by the dispatcher to find methods that can be inherited. Third, user-defined conversion functions must be considered and integrated, as coercions, into the dispatching process.

A mode graph gives us a systematic way to represent domain relationships and type conversions and, thus, forms a framework in which we can look at the dispatching problem. We will use the

⁴Review Chapter 15, Section 15.5, if necessary.

Exhibit 18.5. Defining a subdomain in Pascal.

```

TYPE fingers = 1..10;      { A subrange type declaration.}
VAR f1, f2:  fingers;
...
readln(f1);               { Integer functions are defined for fingers. }
f2 := f1 + 5;             { A legal computation on subdomain fingers. }

```

The type declaration for `finger` defines a subdomain of the integers; integer operations such as “+” and `readln` are defined over this subdomain.

concept of a mode graph to discuss dispatching in C++, Miranda, and Aleph, which is a newly developed language that provides full support for generics.

18.2 Subdomains and Class Hierarchies.

18.2.1 Subrange Types

A domain D' is a *subdomain* of a *superdomain* D if all the elements of D' also belong to the domain D . Many programming languages allow the programmer to declare a new domain that is a subdomain. In Pascal and Ada, the domain corresponding to the subrange type is a subdomain of the domain formed by the base type [Exhibit 18.5].

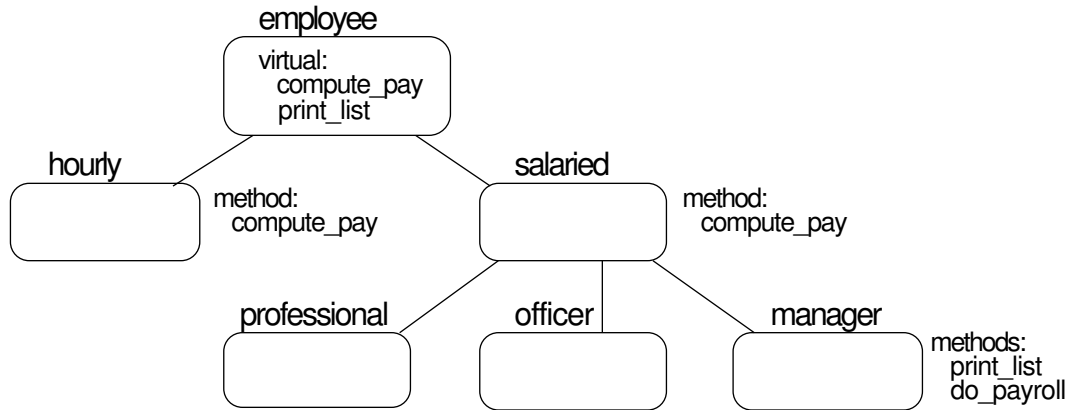
The domain formed by a subrange type is the easiest kind of subdomain to implement because elements of the subrange type and the base type have the same representation. Values from the subrange type may be freely stored in base-type variables; base-type values may be stored in subrange variables if they are within the defined limits of the subrange. Subrange and base-type values may be compared directly for equality.

Functions defined for the superdomain are *inherited* by the subdomain; that is, any function defined for D is also applicable to objects of subdomain D' . In mathematics, the integers are a subdomain of the reals which are, in turn, a subdomain of the complex numbers. Thus complex operations are applicable to reals and real operations are applicable to integers. With Ada and Pascal subrange types, functions defined for the base type are inherited by (applicable to) objects in the subdomain, but not vice versa.

Any function defined for the base type could accept and process arguments from the subrange type. We say that the subrange type can *inherit* these functions from the base type. No special checking or conversion is needed to make a subrange-type argument appropriate for the base-type function. Conversely, functions defined for a subrange type can process any values from the base type that are within meaningful limits. Thus the base type can inherit functions from the subdomain, and those functions will be meaningful part of the time.

Exhibit 18.6. Diagram of a domain hierarchy.

Domains are diagrammed above their subdomains. As we travel down the links, more and more specific information can be known about each subdomain.



To make subdomains and these simple inheritance relationships work, a translator must include run-time range checks to ensure that every program object stored in a subrange variable or processed by a subrange function does actually belong to the subdomain.

18.2.2 Class Hierarchies

Smalltalk and the other object-oriented languages provide much broader support for subdomains. They allow the programmer to define a new subdomain whose representation is semantically related to the superdomain but is not structurally identical to it. These languages exploit the hierarchically related nature of many external domains. Start with a classification system, such as that shown in Exhibit 18.6. Lines connect each domain (above) to its subdomains (below). Less detail is known about or relevant to the broad class at the top of the hierarchy, but as we move down the subdomain links, more and more details can be specified. Each category in the diagram has its own specific characteristics and inherits characteristics from the categories above it. These domains and their relationships are described in Exhibit 18.7.

Altogether, we have defined five kinds of data records, where each one must contain all the information of the domain above it, plus more. For example, a “manager” record must contain fields for an annual salary, name, social security #, two codes, and a link, because a manager *is* a salaried employee and *is* an employee. However, a manager’s record must also contain a pointer to the list of employees managed. Thus although the record types that might implement these domains are structurally different, they have semantic commonalities. A function that processes employee data could also process manager data correctly *if the data fields of the two types were represented in a compatible order*.

Exhibit 18.7. A group of related domains.

Here are descriptions of the fields in the records for two kinds of employees. (These records have a link field so that they can be organized into lists.) In a representation of these domains, the record describing the superdomain (employee) is a subrecord of that describing the subdomain.

Domain	Superdomain	Information to be recorded	Functions defined
Employee		Name, social security #, dept code, job code, pointer to next employee	Print employee list Print paycheck Enter new employee
Hourly_emp	Employee	Pay rate, hours worked, overtime, vacation days, group number	Compute pay Enter time card Change data
Salaried_emp	Employee	Annual salary, vacation days used	Compute pay Enter vacation
Manager	Salaried_emp	List of employees managed	Print list of people managed Add new employee to list Do payroll
Professional Officer	Salaried_emp Salaried_emp	Group number List of managers managed, permitted to sign checks?	Permission predicate

An object-oriented language, such as Smalltalk or C++, permits the programmer to declare and use this kind of domain/subdomain relationship to define a hierarchy of domains. C++ classes were introduced in Chapter 16 as an implementation of abstract data types. Classes are also used to implement generic domains with subdomains and function inheritance.

We will use C++ syntax and rules to explain how these class hierarchies work. Given a class, G , we can derive a subclass G' from G , such that G' has all the members (data and/or functions) of G , plus more. The C++ syntax for declaring a derived class is :

```

class    <new-class-name> : [public] <base-class-name> { (a)
          <new-private-members>                               (b)
public:
          <new-public-members>                               (c)
};

```

When we use a derived-class declaration, a new class, G' , will be constructed that is a subclass of the specified base class, G . If the keyword **public** is used, (line a) all the public parts of G will also be public parts in G' ; if omitted, the public parts of G will become private parts of G' . If G has private members, an instance of G' will contain fields for these parts, but they are not visible to the functions defined for G' and can only be accessed by functions defined for G (unless

a “friend” declaration is included in G). This visibility rule supports modular data-hiding, which is so important for achieving reliable systems.

In section (b), the private members of G' are listed. Included here are data fields that exist for the subclass G' but not for the superclass G , and functions (if any) that will be used only locally, by other functions defined for G' . Finally, additional public members of G' are listed in section (c). These must include all basic functions needed to operate on the subclass. Additional classes may be derived, similarly, from either G or G' .

Exhibit 18.8 shows the C++ declarations that would implement part of the domain hierarchy from Exhibit 18.7. To finish implementation of this hierarchy, we need to supply definitions for the remaining class, `officer`, and definitions for all the functions declared in all the classes. Definitions for these functions can be placed anywhere in the program, but good style dictates that they should be placed just after the class declaration. The functions that must be defined are:

```
employee::employee      employee::print_paycheck    employee::print_list
employee::compute_pay  salaried::compute_pay      salaried::take_vacation
hourly::hourly         hourly::compute_pay        hourly::record_time_card
manager::manager       manager::add_employee       manager::print_list
                        manager::do_payroll
```

Although these method *definitions* are outside the class declaration, the function methods themselves were *declared within the class*, and are class members with full access privileges. The placement of the function definition (before or after the end of the class declaration) makes no difference in the semantics—the only difference is practical; definitions inside the class are expanded in-line, like macros, those outside are compiled as subroutines. C++ functions can be overloaded, that is, a function name can have several defining methods, belonging to several classes. For this reason, the full name of a function method must be used when we define it outside its class. Thus we must write `hourly::compute_pay()` and `salaried::compute_pay`, not simply `compute_pay`.

Representation and Visibility. A program object is created by instantiating a class, and this action is triggered either by a declaration or a “new” command. The result is a record with one field for each of the variables (both public and private) within the class. A derived class will be represented by the same type of record with fields added on the end. The names of the class variables are like the names of the fields of a record, except that more complexity is involved because of information-hiding.

In a Pascal record, all the fields have the same *visibility*; that is, if one part is accessible in a given context, then all parts are accessible. But in a C++ class, the fields corresponding to private members have visibility restricted to the class functions, while public members have broader visibility. When we make a derived class, fields for private members of the parent class will be part of the record and take up space, but the names of those fields will not be known within the derived class, and those fields will be inaccessible to new functions. However, the public functions of the base class become members of the derived class and can be used to manipulate these fields.

Exhibit 18.8. Deriving classes in C++.

Below are declarations for some of the classes described in Exhibit 18.7. Exhibit 18.6 is a diagram of the relationships among these classes. This code is discussed throughout the rest of this chapter. We presume that `card` and `pay_data` are previously defined classes.

```

class employee {
    char name[], soc_sec[13], *dept_code, *job_code;
public:
    employee * link;
    employee(int)          // The constructor; argument is name length.
    void print_paycheck;
    virtual pay_data compute_pay;
    virtual void print_list;

    void employee::print_empl()
    {   cout << "Name:  " << name << "\n\t" << soc_sec
        << "\tDept:  " << dept_code << "\tJob:  " << job_code
        << "\n";
    }
};          // End of class employee.

class salaried : public employee {
    int annual_salary, vacation_used;
public:
    pay_data compute_pay();
    void take_vacation();
};          // End of class salaried.

class manager : salaried {
    employee* staff;
public:
    void add_employee();
    void print_list();
    void do_payroll();
    manager();          //The constructor function.
};          // End of class manager,

class hourly : public employee {
    float pay_rate, hours_worked, overtime;
    int vacation_used;
public:
    pay_data compute_pay();
    void record_time_card(card*);
    hourly();
};          // End of class hourly.

```

Limiting Visibility. It is possible to use a derived class to modify the visibility of members of an existing class. Let us say that class G has some private members and some public members. There is no way that deriving a class from G can affect the visibility of its private members—they may be accessed only within G . But the public members of G may be made private in a class derived from G . To do this, omit the word “public” from the header of the derived class declaration. Public members of the base class then become private members of the derived class. We say that a derived module is *opaque* when it completely hides its base module. A half-and-half, or semi-opaque, situation may also be achieved. When the keyword `public` is omitted, you may list the names of selected members from the base class in the public part of the derived class. This does not create an additional field in the derived class; it simply controls the visibility of the base-class field. The syntax is:

```
class    <new-class-name> : <base-class-name> {
        <new-private-members>

public:
        <base_class_name>::<member_name>;
        <new-public-members>
};
```

Type compatibility. If G' has a public base class G , then an object of type pointer-to- G' can be stored in a pointer variable of type G^* (pointer-to- G) without use of explicit type conversion. For example, the class `employee` has a field named `link` of type `employee*`. Every instance of class `manager` and `hourly` also has this field, because `manager` and `hourly` were derived publicly from `employee`, and `link` is public in `employee`. The `link` field is able to store addresses of type `hourly*` and `manager*` because these classes were derived from `employee`. Thus we may make a linked list of `hourly` employees and store the head in a `manager` record.

This is a crucially important issue. The idea of a class hierarchy is that all variants of a base class are semantically related. Even more, an instance of a derived class *is* an instance of the base class, with some added information. Compatibility of pointer types throughout the levels of a hierarchy is essential to implement this underlying semantic notion.

18.2.3 Virtual Functions in C++.

C++ contains a simple kind of support for virtual functions. A base class may contain a “virtual” declaration, such as the `print_list` function in Exhibit 18.8. The intent is that a virtual function should be used where the method for carrying out some abstract process is implementation-dependent. Let us use the term *virtual class* to mean a class that contains a virtual function or one that is derived from another virtual class. All the classes in the “employee” example are virtual classes because the base class contains one virtual function.

A virtual function *must be defined* for the base class in which it is declared, but it may be redefined for any derived classes that need a different method. In our example [Exhibit 18.9] the method for computing a paycheck is different for salaried and hourly people, and depends on data

Exhibit 18.9. Calling a C++ function with two defining methods.

We define output function methods for the classes in the `employee` hierarchy established in Exhibit 18.6. The method in the derived class calls both of the methods in the base class.

```
void employee::print_list() // Does not print head of list.
{
    employee * scan;
    for (scan=link; scan != NULL; scan=scan->link) print_empl();
}

void manager::print_list()
{
    cout << "\n\nManager: ";
    print_empl();
    cout << "\nEmployees Supervised:\n";
    employee::print_list();
}
```

that is specific to these classes. The function `compute_pay` is, therefore, declared as a virtual function and defined separately for the two classes. Another application of virtual functions is to allow a method to do some actions specific to its local class, then call the general function from a higher class to complete the job. In our example, the `print_list` method for `employee` prints some headings, then calls the base method, using its full name, `employee::print_list`.

If another class is derived from a derived class, the classes form a hierarchy. A virtual function may be defined at several levels of a hierarchy. In this case, several methods for a virtual function may all be applicable to instances of a class at the bottom of the hierarchy. In our “employee” example, two methods are defined for `print_list` (in class `manager` and the base class, `employee`). Both could be applied to an instance of class `manager`, but only the method in the base class could be applied to instances of the classes `salaried` and `employee`.

If a function is called using its full name (`class_name::function_name`) there is no ambiguity, and the call is translated in the ordinary manner. However, if a function is called without using its class name, the call is ambiguous and must be resolved by the dispatcher. The same dispatch rule can be used here as is used in the simple case: start at the bottom of the class hierarchy and look up the hierarchy tree until a method is found for the function. Thus if the programmer wishes to use a method that is not the closest one to its object’s class, the full, qualified name of that method must be written.

It is up to the compiler and the linker to make the necessary connections between the virtual function and its methods, and to ensure that all those methods are consistently typed. Similarly, it is up to the translator to dispatch the correct method for a call on a virtual function. This is complicated because the method being called might not even exist when the call is compiled, and the actual dispatching must be done at link time or at run time.

To understand *why* dispatching must be delayed, consider the `do_payroll` function defined for class `manager` in our “employee” class hierarchy [Exhibit 18.7]. Assume that `do_payroll` calls `print_paycheck` to print out a paycheck for each employee on the manager’s staff list. By inheritance, we can use this function to print a paycheck for any employee, and we can call it from the class `manager`. However, assume that `print_paycheck` calls upon the virtual function `compute_pay` to get the data needed for a check. Now, `compute_pay` is defined differently for hourly and salaried employees, and a manager’s `staff` list will generally contain both kinds of employees. Thus at run time a single function call written in the `print_paycheck` function must sometimes dispatch `hourly::compute_pay` and sometimes `salaried::compute_pay`.

This decision cannot be made until each list element is processed. Thus the type of the argument to `print_paycheck` must be examined at run time. Happily, once this type is known, the simple dispatching rule *still works*. In traditional compilers (such as the typical C compiler) the type information from the declarations is put into the symbol table, used at compile time, then discarded. Little or no type information is carried over to run time. In order to dispatch a virtual function, though, it is necessary to have this information at run time. Some version of the translator’s type objects must exist then, and each object must include a type pointer.

For this reason, objects belonging to virtual classes in C++ are compiled differently. A type field (a pointer to the type object for the object’s class) is made part of every object in a virtual class. The dispatcher examines this type field and selects the right method for an object at run time. Thus we incur space overhead when we use a virtual class, and time overhead when we call a virtual function.⁵ The benefits of virtual functions, however, outweigh these costs. Using function inheritance and virtual functions, we can:

- Avoid proliferation of nearly identical names for implementations of the same external process.
- Avoid writing duplicate copies of code for related classes.
- Write simple code with highly flexible run-time behavior.
- Extend the benefits of strong typing to nonhomogeneous domains.
- Create class hierarchies that are easy to extend when new, related data types must be defined.

18.2.4 Function Inheritance

The real importance of the class hierarchy is that it defines a system of classes with related semantics on which a system of function inheritance can operate. Briefly, functions defined in any class become members of every derived class and may be applied to instances of the derived class and called as if they were local. We say that the derived class *inherits* the function from its base class.

Every function call has an implied argument, which we will call the *object of the call*. To translate a call, the compiler must find and dispatch the correct method for that object. The dispatching rule is this:

⁵This overhead is incurred only for virtual classes. It does not reduce the efficiency of operations on nonvirtual classes.

- If a method is defined in the same class as the object of the call, the translator will dispatch that method.
- Otherwise, move up the class hierarchy, one level at a time, looking for function-members with the correct name. Dispatch the first method for the function that is encountered.
- Unless the function is declared to be “virtual” in the base class, there should be exactly one applicable method in the hierarchy. If no appropriate method is found, there is a type error.

Constructors in Derived Classes. A constructor function is atypical, since we do not ordinarily call it explicitly. Whenever the translator allocates space for a new class instance, it calls the constructor function for that class to initialize the new storage. However, to instantiate a derived class which has a constructor, the translator must execute *two* constructor functions—first, the constructor for the base class, then the constructor for the derived class. Thus the derived class must “inherit” the constructor from its base class.

This leads to a real problem, since many constructor functions have parameters. The header line of each constructor definition specifies what parameters it needs, but how can the constructor for the derived class convey the right arguments to the base constructor? The solution is to expand the syntax of the language to allow the programmer to supply argument lists for a series of constructor functions. Arguments for the constructor of the derived class are given in the instantiation call. Arguments for the constructor of the base class (or classes) are given in the *definition* of the constructor for the derived class. The syntax is:

```

<derived_name>::<derived_name> ( <argument_list_for_derived_class> ) :
    ( <argument_list for base class constructor> )
    <initializers for base class members>
    { <body of derived constructor> }

```

In the definition of the derived constructor, the programmer can use the base-class’s argument list to pass on the derived-class arguments or to supply the base class constructor with constant arguments. If there are several nested derived classes which have constructors, a list of argument lists (separated by commas) is given, with the list for the most basic class first. The constructor for the base class will be executed *before* the constructor for each class derived from it. (If both classes have destructors, the destructor for the *derived* class will be executed before that of the base class.)

Strengths and Limitations. The classes in C++ are immensely powerful and, when used well, can reduce both programming errors and tedious, repetitive coding work. This language is a giant step beyond standard Pascal. Several major semantic ideas are covered here that go beyond the traditional languages. They give C++ much of its power. These are:

- Controllable public/private data and functions.
- Declarable class relationships.

- Functions with multiple defining methods.
- Function inheritance.

By themselves, these semantic mechanisms are somewhat limited. We briefly list these limitations below and consider solutions in Section 18.4.

- These mechanisms do not address the problem of generic domains with representations that are related in an ad hoc manner. The only domain relationship that can be modeled is that between a record type and a longer record type with the same initial parts.
- All hierarchies and all function inheritance are totally tree structured; no class can have two parent-classes. However, some external generic domains naturally form graph-structured relationships in which a domain inherits properties from two different directions.
- During dispatching, the types of the arguments to a call are not considered. Only the class of the object of the call is used to select a method.
- C++ is still a superset of C, and, therefore, you can get around all the visibility rules of C++ by using pointers, which are semantically insecure types in C! The problem is that a pointer to any type can be cast to any other pointer type. This can be used to gain access to private data.

18.2.5 Programmer-Defined Conversions in C++

With some limitations, the C++ programmer is able to define conversions that the dispatcher can use. A class definition normally contains an ordinary constructor function, which is invoked automatically when a new storage object for the class is allocated. In addition, the class may contain one or more one-argument methods with the same name as the constructor. Each one is taken to be a conversion function and must create a value of the type of the class. As with any C++ constructor, the new value is returned by assigning values to some or all of the class members—an explicit return statement is not used in a constructor. The programmer must make sure that any constructor with one argument is a semantically valid conversion function from the type of the argument to the type of the class, because the compiler will use these methods, whenever needed, for coercion.

C++ can use constructor functions to coerce arguments in simple cases. In the class `complex` of Exhibit 18.10, four constructor methods are defined. The first one constructs the complex number zero and will be used to initialize every complex variable for which no other initializer is specified. In the example, the variable `cx` is initialized using this method.

The second method constructs a complex value from real and imaginary components [Exhibit 18.11]. This will be used primarily to define the member functions for the class; it defines the relationship between a complex number and its components. It can also be used to construct initializers for complex objects, as in line (b), and to create complex objects during execution, as in line (d).

Exhibit 18.10. Conversion functions in C++.

```

class complex
{   rp: float;
    ip: imag;
public:
    complex(){ rp=0.0; ip=imag(0.0); }
    complex(float f, imag i){rp=f; ip=i;}
    complex(float f){rp=f; ip=0;}
    complex(imag i){rp=0; ip=i;}

    complex operator-() { return complex(-rp, -ip); }
    friend complex operator+( complex c1, complex c2 );
    friend complex operator*( complex c1, complex c2 );
    friend complex operator-( complex c1 );
}

complex operator+( complex c1, complex c2 )
{   return complex( c1.rp+c2.rp, c1.ip+c2.ip ); }

complex operator*( complex c1, complex c2 )
{   return complex( c1.rp*c2.rp + c1.ip*c2.ip, c1.rp*c2.ip + c1.ip*c2.rp ); }

```

The last two methods are semantically valid conversion functions from types `float` and `imag` to type `complex`. They can be called explicitly, but they will also be used by the translator to coerce arguments in calls on complex operations. Lines (a) and (d) illustrate contexts where an argument will be coerced. In line (a), the float number `-1.6` is used as an initializer for an instance of class `imag` (from Exhibit 16.10). This triggers a call on the conversion from `float` to `imag` that was defined in class `imag`.

In line (d), the operator “+” is called to add a complex number and a float. This situation is somewhat more complex. The dispatcher looks at this call on “+” and must find an appropriate method for it. The dispatching rule in C++ is given in Exhibit 18.12. By this rule, the method `complex operator+(complex, complex)` is finally selected, and the second operand, `f1`, is coerced to type `complex`.

Programmer-Defined Casts in C++. In C++, as in C, no distinction is made between conversions and casts. We have seen that a C++ class can be used to implement a mapped domain with semantic protection against misuse. But casts are required to implement the basic functions for the new mapped domain, and casting is not automatically defined for such a class. In this situation, the facility for programmer-defined conversions can be used to define an upward cast. To specify

Exhibit 18.11. Coercion with a programmer-defined function.

```
main()
{   float f1=2.5;
    imag im=-1.6;           //(a)
    complex cx;            // Initialized to zero.
    complex dx = complex( 1.0, imag(1.0) );   //(b)
    cx = new complex( .1, im );              //(c)
    cx = cx + f1;                       //(d)
}
```

the downward cast, from the class type to a representing type, an **operator** function may be used. This defines the target-type name as a conversion function that can be called explicitly using the normal syntax for casts. Both kinds of programmer-defined casts are illustrated in Exhibit 16.10, where their use is required to define the basic functions for the new mapped class.

It is important that the two casts were defined as private functions. This means that, within the class, the representation of a class object can be manipulated, as it must be, to make the necessary calculations. However, outside the class, the relationship between the two domains is completely unknown.

Exhibit 18.12. Dispatch rules for C++ overloaded functions.

1. Look for a method whose operand types match the argument types exactly.
 2. If none is found, look for a method such that the argument types can be made to match the parameter types by using no more than one predefined conversion function on each operand.
 3. If none is found, and one or more operand belongs to a defined class, look for a method such that the argument types can be made to match the parameter types by using no more than one user-defined conversion function per operand.
 4. If more than one possible way to do user-defined conversions is found, or if none is found, the dispatch fails.
-

18.3 Polymorphic Domains and Functions

18.3.1 Polymorphic Functions

A *polymorphic type* is a single type definition that includes two or more alternative specific type declarations. A *polymorphic object* is the representation of one of these species along with some form of discriminant field that encodes *which* species is present.

It is important to understand the differences between polymorphic functions and the generic arithmetic operators in languages such as Pascal and Ada. The generic nature of the Pascal operator extends only until compile time; the compiler then chooses a specific method to implement the operator. Thereafter Pascal code is fully specific. A polymorphic function does run-time type checking, when necessary, and a language with good support for polymorphism will do this checking automatically. For example, APL is an array-oriented polymorphic language which tags each data object with a type field that describes the number and extent of its dimensions. The programmer may write an APL function, say *Fun*, to operate on a pair of numbers. The programmer may then call it with a simple pair of numbers or with two equal-length arrays of numbers. The shapes of the actual arguments are tested automatically, and the translator executes appropriate code. It will apply *Fun* once if given a simple pair of numbers, but apply it repeatedly to each corresponding pair of elements if given two arrays.

The Pascal variant record is an early attempt to implement limited polymorphic types. It permits the programmer to specify that objects of the type may have a variety of different sizes and structures. However, it is not a very satisfactory implementation of polymorphism for two reasons:

- When creating a polymorphic object, storage is allocated for the largest variant, even when the value stored there is much smaller.
- The theoretical semantics of variant records are not enforced by the translator, and run-time tests of the discriminant field must be coded manually.

More modern languages have extended this idea and addressed the problems.

A *polymorphic function* is a function that accepts arguments of a polymorphic type. It tests the argument at run time to determine which variant is present, then executes code appropriate for that argument. These run-time type tests might be automatically generated by the translator or explicitly written by the programmer.

Polymorphism becomes very important in functional languages because they support higher-order functions. A higher-order function, often called a *functional*, may take functions as parameters and/or produce a function as its result. Some familiar functionals are functional composition and **reduce** [Exhibit 12.15]. But, in a typed language, functionals are almost useless unless they are polymorphic. Thus higher-order functions have been a strong motivating force in the development of polymorphism in functional languages.

Even a language with little or no support for domain checking may be extended to perform polymorphic domain checking manually. To accomplish this the programmer would explicitly attach

Exhibit 18.13. Manual polymorphic dispatching in Ada.

In older languages, we use manual type-testing to write code to process a polymorphic type. The Ada procedure below might be written to print out an item belonging to the polymorphic type `Person` that was defined in Exhibit 14.25. In this type, the field named “Sex” is the discriminant tag. We explicitly test the contents of this field and dispatch one of three specific functions.

```
PROCEDURE Print_Person ( Who : Person ) IS
BEGIN
  CASE Who^.Sex IS
    WHEN Male =>      THEN Print_Male(Who);
    WHEN Female =>   THEN Print_Fem(Who);
    WHEN OTHERS =>   Sex_Error(Who);
  END CASE
END Print_Person;
```

some type information to every data object, then explicitly include code in the function definitions to test this type field.

18.3.2 Manual Domain Representation and Dispatching

Before looking at a modern implementation of polymorphic domains, let us see how polymorphism might be implemented manually in an older language. An *ad hoc polymorphic domain* is the discriminated union of two or more specific types. If an object `OB` belongs to an ad hoc polymorphic domain, `PD`, then its value at run time may belong to any specific type included in `PD`. It is not possible to predict, at compile time, which specific type this will be. However, at run time, the specific type of any object is known.

In some languages, for example, `Pascal`, an ad hoc polymorphic domain may be implemented as a variant record with a tag field. Exhibits 14.25 and 14.24 show a polymorphic type declared in `Ada`. However, neither `Ada` nor `Pascal` supports the run-time type checking that is necessary to ensure valid use of such domains. To process polymorphic objects, we must write code, such as that in Exhibit 18.13, which explicitly tests the tag field and branches appropriately.

Explicit Domain Testing versus Strong Typing. Achieving semantic validity is the purpose of both strong typing and explicitly testing the discriminant tags on a polymorphic type. However, there are some major differences between the results achieved by the two systems.

On the practical side, explicit type-testing can get to be cumbersome. It forces the programmer to write out, in every function definition, the instructions to check the discriminant tag of the argument and produce an error comment if there is a mismatch. A strongly typed language handles this kind of domain checking automatically for simple types; the programmer needs only to declare

a domain name for each object and parameter.

Second, polymorphic type checking both permits and requires domain testing to be postponed until run time. On the one hand, simple strong-type-checking, which happens at compile time, is significantly more efficient if the program is ever used to process a lot of data. Run-time checking is time consuming. On the other hand, run-time domain checking is inherently more powerful and flexible. It permits the domain-checking system (or the programmer) to make finer distinctions between appropriate and inappropriate arguments. A run-time test can depend on the actual data values, not just on the declared type of the argument. For example, in *Pascal*, strong-typing ensures that a list-processing function will always get a list argument. In *Miranda*, automated polymorphic domain testing can distinguish lists from nonlists but can also be used to distinguish lists containing data from null lists.

18.3.3 Automating Ad Hoc Polymorphism

Polymorphic Domains

The modern functional languages have more advanced support for generics than any existing languages in the *ALGOL* family. This support comes at two levels:

- User-defined domains may be ad hoc polymorphic. A domain may have several alternative representations, like a discriminated union type. Functions can be written that test the discriminant automatically and dispatch the appropriate code.
- A generic type may be defined with a type parameter, and functions may be defined with generic parameters of this sort. (These are covered in Section 18.4.)

We use *Miranda* to illustrate these powerful general definition and dispatching methods. A type definition is written using the symbol “:=”, which we read as “is type”. A polymorphic type declaration consists of a series of clauses separated by the “or” symbol, “|”, which correspond to the alternatives of a discriminated union type in *Ada* or *Pascal*. Each clause has a discriminant name followed by a tuple of type specifiers, much like a simple *Pascal* record declaration. The discriminant names can be used either as constructors, for making an object of that type out of appropriate components, or as type predicates, when defining a function for the type.

In Exhibit 18.14 we use *Miranda* to define `tree`, a polymorphic type. The notes for this example follow:

- a. We define the type `tree`. A tree may be a leaf, which is an integer, or a node which is a tuple of two trees. This declaration defines the discriminant tags `Leaf` and `Node`, which we use later as object constructors and as type predicates.
- b. We construct a tree of the first form, using the constructor `Leaf`. Note that the tag name is part of the constructed object—the result of this line is that the name `leaf1` is bound to a two-tuple consisting of the tag `Leaf` and the number 3.

Exhibit 18.14. A tree type in Miranda.

We define an ad hoc polymorphic type named “tree”. This also defines the constructor/predicate names “Leaf” and “Node” for the two variants of type tree. The last two lines construct two trees named leaf1 and tree1. The letters on the right key the code to the notes in the text.

```
tree ::= Leaf integer | Node tree tree      (a)
leaf1 = (Leaf 3)                            (b)
tree1 = (Node leaf1 (Node (Leaf 17) (Leaf 49))) (c)
```

- c. We construct two leaves, as in the line above, and use them immediately to construct one node. That node is, in turn, combined with the previously constructed leaf, leaf1, to make a tree named tree1. The structure of this tree is shown in Exhibit 18.15.

Unlike Pascal, part names for the fields of a tuple are not supplied in the type declaration, and the new type name may be referenced recursively in the declaration. The lack of defined part names requires explanation. In Miranda, the type of an object is not part of the object, as in APL, nor is it part of a name, as in Pascal. Types are deduced from the structure of objects. Field names, in turn, are not part of the type definition. When an object is passed as an argument, a sophisticated type-deduction algorithm⁶ is used to check whether it belongs to the domain of the function. The tuple- or list-structure of the argument is checked, along with the types of its simple components. As part of this checking process, dummy parameter names, defined in the function header, are associated with each field of the argument. These are temporary and local, like parameter names in traditional languages, and are used to access the fields of the argument within that function. Thus the two parts of a pair may be called “left” and “right” within one function body and “head” and “tail” in another.

⁶This algorithm is derived from the work of Hindley and Milner. Cf. Milner [1978].

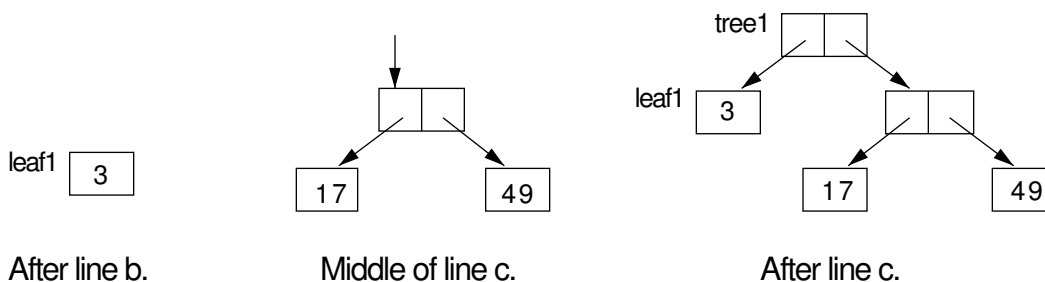
Exhibit 18.15. A tree with polymorphic nodes.

Exhibit 18.16. Polymorphic dispatching in Miranda.

We define and call a polymorphic function named `max-tree` with methods to process both kinds of trees (nodes and leaves). The letters on the right key the example to the explanation in the text.

```

max-tree (Leaf ldata) = ldata                (a)
max-tree (Node n1 n2) = max1, max1>max2     (b)
                      = max2, max2>max1     (e)
                      = max1, otherwise
                      where                  (c)
                      max1 = ( max-tree n1 ) (d)
                      max2 = ( max-tree n2 )

```

Evaluating `(max-tree leaf1)` yields 3, and evaluating `(max-tree tree1)` yields 49.

Miranda permits us to write functions over ad hoc polymorphic domains, like `tree`, that will be domain-checked and dispatched automatically. A function definition is written as a series of methods, each one defined for a different subdomain. The domain predicates are called *patterns* and are written on the left. A pattern can specify an argument type, a discriminant tag (called a *constructor* in Miranda), and/or data values. Following the pattern is an “=” and the appropriate computation method for that particular subdomain. In Exhibit 18.16, we define a simple function that processes the tree type defined in Exhibit 18.14. Following are line-by-line notes for the code in the exhibit:

- a. On this line and the next, the discriminant tags defined by the type declaration are used as domain predicates (left), controlling the choice of computation methods (right). Line (a) defines the function `max-tree` for any argument which matches the type named “Leaf”. Within the body of the method, the single component of a `Leaf` will be called “ldata”. The value of the expression on the right will be returned as the value of the function. This code states that the maximum of a `Leaf` is the number stored in it.
- b. The remaining code defines `max-tree` for arguments that are tuples with two fields. These fields are named “n1” and “n2” within the method body.
- c. The “where” clause defines a local context for this method, containing local names “max1” and “max2”. When the outside-in, lazy evaluation process reaches the first reference to `max1` or `max2`, the expression following the name in the “where” section is evaluated and the result is bound to the local name. This value remains bound to the name throughout evaluation of the block; subsequent references to the name refer to the value computed for the first reference.
- d. The function `max-tree` is called, recursively, with the left field subtree as its argument. The maximum value in the left subtree is bound to the name “max1”.

Exhibit 18.17. Dispatching a Miranda call using data values.

```

sum-list []      = 0
sum-list (a:ls) = a + sum-list ls

```

We recursively sum a list by adding the value of the first list item to the sum of the rest of the list. If `list1 = [3, 4, 17, 9, 5]` and we evaluate `sum-list list1` the result is 38.

- e. This guarded expression compares the two maxima of the left and right subtrees and returns the greater.

This automated run-time dispatching of function methods is implemented by a very general and powerful pattern-matcher. It is not limited to checking for subtypes of a polymorphic domain, but can also perform run-time checks involving the values of data objects. For instance, Exhibit 18.17 has one method defined for null lists and another method defined for lists with at least one component.

Exhibit 18.18 summarizes and gives examples of the kinds of patterns that the *Miranda* programmer may use to define methods. Dispatching a function call is done by a run-time case analysis, examining each pattern in turn, until one is found that matches the structure and/or value of the actual argument. When a match is found, the dummy names used to write the pattern are bound to the parts of the argument, and these bindings are used within the function body. Exhibit 18.19 shows the bindings that would happen for some calls on the functions `sum-list` [Exhibit 18.17], `max-tree` [Exhibit 18.16], and `pow10` [Exhibit 12.20].

The important principle here is that the dispatch is done at run time, distinguishing it from anything that can be done with nonunion types in languages such as *Pascal* and *Ada*. It is true that an *Ada* programmer can emulate run-time dispatching by using a discriminated union type and coding the dispatching process manually, as in Exhibit 18.13. However, manual type checking is never as convenient or as safe as checking that is built into the semantic basis of the language. The *Miranda* function definitions using pattern matching are far clearer and more elegant than corresponding *Ada* code.

Exhibit 18.18. Pattern construction in Miranda.

Pattern type	Example
constant	<code>factorial 0 = 1</code>
number greater than 0	<code>factorial (n+1) = (n+1)*factorial n</code>
null list	<code>sum-list [] = 0</code>
nonnull list	<code>sum-list[a:x] = a + sum-list[x]</code>
tuple	<code>max-tree (Node n1 n2) = ...</code>

Exhibit 18.19. Binding during the pattern match.

Data object definitions:

type	name	value
tuple	leaf1	= (Leaf 3)
tuple	tree1	= (Node leaf1 (Node (Leaf 17) (Leaf 49)))
list	list1	= [1, 3..11]
list	list2	= []
integer	numb1	= 17

The functions below were defined in Exhibits 12.15, 12.20, 18.16, and 18.17.

Call	Bindings
max-tree leaf1	ldata is 3
max-tree tree1	n1 is leaf1, n2 is (Node (Leaf 17) (Leaf 49))
max-tree numb1	Pattern match fails.
max-tree list1	Pattern match fails.
pow10 numb1	n is 16.
sum-list list1	a is 1, ls is [3, 5..11].
sum-list list2	Pattern matches null list, no binding necessary.
reduce '+' list1 0	f is '+', a is 1, x is [3, 5..11], and n is 0.

18.3.4 Parameterized Domains

Miranda is a strongly typed language. Every object has a type, even though it is not declared. The type is deduced from the structure of the object. Functions, also, have types which are deduced from the function code. If a function is called with an argument whose type is inconsistent with the function definition, a compile-time error comment is generated. The programmer may also choose to declare the type of a function, in which case a compile-time error is produced if the declared type and the deduced type do not match.

Names may be defined for types and used as a notational convenience. The names carry no semantic meaning in themselves but are simply a shorthand for the structural description.⁷ To define a type name, you use the “==” sign:

```
<typename> == <type expression>
```

Miranda supports domains with type parameters and provides a notation for talking about a type parameter. Ordinary types are expressed structurally, as shown in Exhibit 18.20. The type of a list is denoted by writing the type of its elements inside list brackets; the type of a tuple is written as a tuple of types. The type of a function is written in curried notation, starting with the type of the first argument and ending with the type of the result.

Type parameters are denoted by strings of asterisks. If a type expression has one type parameter,

⁷See the discussion on type identity in Chapter 15, Section 15.4.

Exhibit 18.20. Miranda type expressions.

Assume T , $T1$, etc. are types. Then we can write the following type expressions:

Expression	Interpretation
$[T]$	A list with elements of type T .
$[[T]]$	A list of lists of elements of type T .
$(T, T1, T2)$	A tuple of three elements of types T , $T1$, and $T2$.
$T \rightarrow T1$	A function with argument type T and result type $T1$.
$* \rightarrow *$	A function whose argument and result have the same type.
$[*] \rightarrow *$	A function that returns an element of the same type as the base type of its list argument.
$* \rightarrow ** \rightarrow **$	A function of two arguments whose result is the same type as the second argument.

we write “*”; for an expression with three distinct types, we write “*”, “**”, and “***”. If a type expression uses the same type parameter symbol twice, it means that both occurrences must be replaced by the same argument. For example, the type of the subscript function is:

`! :: [*] → num → *`

Abstract Data Types. Miranda supports code modules analogous to the generic packages in Ada, except that Miranda’s type parameters are not bound at compile time, so it is capable of true run-time generic behavior. A Miranda script may contain the directive to “%include” the script in some other file, or a directive to “%export” some locally defined symbols. Scripts may have parameters, just as in Ada, and a parameterized script must be instantiated by supplying arguments in the “%include” command.

An ADT, called an “abstype”, is declared by specifying the ADT interface (public symbols) followed by the definitions of those symbols (private part). Exhibit 18.21 declares a stack ADT with a single type parameter for the base type of the stack. The stack itself will be represented by a list. The rules for type compatibility and access to private parts are very similar to those in C++ classes; within the scope of the declaration, the type “stack *” is considered to be just the same as the type “[*]”, and the implementation equations may access it using the ordinary list operations. Outside the `abstype` declaration, though, a `stack` may only be accessed using the declared functions.

18.4 Can We Do More with Generics?

We have looked at a variety of languages that support generic and/or polymorphic functions. It is appropriate, now, to look back and compare these facilities, to analyze strengths and weaknesses, and to ask whether a language could do more.

Exhibit 18.21. The ADT “stack” in Miranda.

```

abstype stack *
with empty::stack *
    isempty::stack * → bool
    push :: * → stack * → stack *
    pop  :: stack * → stack *
    top  :: stack * → *

stack * ==[*]      || A stack is a list of anything.
empty      = []    || A literal stack, to start things off.
isempty s = (s=[]) || A null list represents an empty stack.
push a s   = (a:s) || Append the new item to the head of the list.
pop (a:s)  = s     || Remove list head and discard; return changed stack.
top (a:s)  = a     || Return the list head, don't change the stack.

```

Binding time, Flexibility, and Efficiency

The *binding time* for type variables is the first and biggest difference among the languages we have studied. Ada, C++, and Miranda all support some sort of generic behavior. However, Ada’s generics are much simpler, more efficient, and less flexible. Ada does not need a dispatcher at all, because the programmer binds specific names to instantiations of generics, and uses those specific names when writing the code. The result can be fully compiled, and is easy to compile, but it does not permit any run-time variability at all. For example, variable-length strings are quite clumsy to manage in Ada.

Contrast Ada to Miranda. Variable-length lists are used everywhere in Miranda programs, and polymorphic tuple types exist at run time. The idea of “type” is defined in Miranda so that a single type includes more than one specific representation. The Miranda compile-time dispatcher verifies that a function is defined for the type but does not worry about which representation of the type will be present later. The run-time-dispatcher examines the actual arguments and dispatches the particular method that matches the argument. Many Miranda functions do repetitive processes on data structures built from polymorphic tuple types. The run-time dispatching is essential to support this, since the method selected will differ from one call to the next, depending on the shape of the data structure.

Miranda’s run-time variability makes a language that is flexible and easy to use; strong typing ceases to be a barrier between the programmer and his or her work, and becomes a pure asset. The cost, however, is execution speed. Repeating the pattern match for every repetition of every function is a great deal slower than doing it once at compile time.

The C++ compiler takes a middle course. It makes a distinction between function calls for which complete information is known at compile time and those where some run-time variability

can possibly exist. In the former case, fixed code is compiled, as it would be in C. In the latter case, the run-time dispatcher is called. (Review the discussion of virtual functions in Section 18.2.3.) Using a mixed strategy like this seems to be the best strategy, although it is also the most difficult to implement.

The real problem with a mixed strategy is deciding what can and what cannot be known or deduced at compile time. C++ solves this problem simply by deciding that any computation that involves a virtual function will be dispatched at run time. Ordinary functions will be dispatched at compile time. This works to achieve reasonable efficiency only if the use of virtual functions is relatively unusual.

Defining and Representing a Generic Function

In a traditional language, we know what a function is; it is one body of code, with one type. This situation becomes complicated when we deal with generic functions, and we need to decide what a function is and what can be done with functions.

We need to distinguish between the situation in which a function name is overloaded, that is, used for two unrelated methods, or truly generic. A function is truly generic if the translator knows about more than one method, knows the methods are related, and uses that relationship in the dispatching process.

In comparing generic languages, we see different approaches to the question of “what is a function?” In *Miranda*, a function can have several defining clauses, giving it polymorphic behavior, but everything relevant to one function is defined in one place. *Miranda* functions are first-class objects that can be passed to other functions and created dynamically. When we pass a functional argument, this entire polymorphic unit is passed.

In contrast, C++ virtual functions are defined in bits and pieces, with each method “inside” a different class. A constructor function, on the other hand, may have multiple definitions in the same class. Finally, built-in functions (casts, arithmetic operators) have several definitions and are not included in a class at all. Methods for these functions are defined piece by piece, as needed. It is not necessary to know about or edit prior definitions of a function in order to extend the function to handle a new subtype; the new method is simply included where it is needed.

Both constructors and virtual functions are true generic functions, not simply overloaded names. The constructors in a class all bear a semantic relationship to each other; they all return an object of the type of the class. The methods for a virtual function also are semantically related, even though they are defined at different times and may be written by different people and compiled in different code modules. In *Miranda*, this would not be possible; the methods for a virtual function would all have to be collected and written in one place. Thus the C++ approach for defining a generic function has a definite advantage for writing large systems.

A generic function is a collection of methods, each one defined over a different subdomain of a generic domain. Some languages (such as *Miranda*) require the methods to be lexically grouped on the page, in other languages they are semantically grouped by the translator (like *Ada*’s arithmetic operators), in yet others they are tucked into classes and accessed through the class hierarchy so

that the methods are never connected together or treated as a unit (like virtual functions in an object-oriented language).

Operations on Generic Functions

Let us consider a generic function to be a list (not ordered) of methods. You may picture it as a linked list of code modules. There are three operations that we would like to define for a generic function: passing it as an argument, dispatching it, and executing one of its methods.

When passing a generic as an argument, do we pass the entire unit (including all the methods), as in *Miranda*? Or do we pass a single method? Passing an entire function is only meaningful if we have a run-time dispatcher. In *Ada*, we sometimes pass a function as an argument when a package is instantiated. If we wish to send “+” as an instantiation parameter, *Ada* permits us to name the entire function as the argument; it does not require that we denote a single method for “+” [Exhibit 17.20]. An *Ada* instantiation argument is passed at precompile time and is substituted in the package before *Ada*’s compile-time dispatcher works on the code. This is not the same as passing an entire function at run time. Functional parameters in *C* must be single methods, because no run-time dispatcher exists to handle a whole functions. *Miranda* scripts can pass entire generic units at run time because all dispatching is done at run time.

Then consider dispatching; the dispatcher selects one method for each generic function call, if an appropriate method exists. The compiler must keep the methods for a function in some sort of data structure so that it can search all possibilities during dispatching.

A class hierarchy forms a convenient data structure for organizing the methods of a function. The dispatching algorithm is easy to write if each method is attached to a class and no class has more than one method for the same function. The algorithm searches for a method starting at the class of the first (implied) argument and crawls up the class hierarchy tree until it either finds a method or comes to the base class. In *C++*, the base class is required to have a method for each virtual function, guaranteeing that the dispatch will never fail at run time. (This restriction is not necessary, though.)

Handling methods that are not attached to classes is more difficult, as is handling functions that have multiple definitions within one class. To make sense of this, the dispatcher must examine a list of methods, looking at the type of each method. If the type of each argument matches the declared type of the corresponding parameter, the method may be dispatched. We need to define precisely what “match” means, and what happens if no match is found, or if more than one is found.

18.4.1 Dispatching Using the Mode Graph

Consider a generalized dispatching algorithm in which generic functions are represented as lists of methods, and the dispatcher can move around the entire mode graph, including submodule links, binding links, and conversion links. This kind of dispatching algorithm is used in two current

research languages, Haskell⁸ and Aleph⁹ In the rest of this section, we will refer to this generalized algorithm as the *Dispatcher*, with a capital letter.

Dispatching starts with a particular function (a list of methods) and a particular context for the function call. The context is formed from the types of the actual arguments and (possibly) the type of object that the function must return. The Dispatcher examines each method for the function, in turn, and either eliminates it, selects it, or puts it on a *short list* for future consideration.

In an object-oriented dispatch, the only part of the context that is considered is the type of the implied argument. This is easy to implement, but it is limited and can handle only simple dispatches. Even in the examples we have seen, it is inadequate to handle the problem of two constructors in one class. A more general and symmetric dispatcher would consider the types of all the arguments, as C++ does with constructors, and as the Miranda dispatcher does with all functions. The Dispatcher will consider each argument, in turn, and select a method only if all arguments match or can be coerced to match. Functions such as constructors, which can have different methods for different combinations of arguments, will be dispatched correctly.

Matching. Dispatching one argument reduces to the problem of finding a path through the graph from the mode containing the type of the argument to the declared mode of the parameter. The dispatcher will start at the argument mode, search up IS links and BIND links and across CONV links, trying to reach the target mode. Unlike C++ dispatching, the Dispatcher is not limited to using a single CONV link, and it can use CONV links in combination with the other links.

Using conversions freely brings up the question of semantic validity. What is there to prevent a string of conversions from distorting the meaning of the argument beyond recognition? This issue is the reason that we must distinguish carefully between casts and semantics-preserving conversions. A cast is a semantically invalid operation, done only to access the underlying representation for bootstrapping purposes. Casts should never be used freely and never to coerce arguments. Any chain of conversions that goes through two casts is almost certainly semantically invalid. This puts a burden on the programmer to distinguish which functions represent casts, and which are valid conversions, then leave the casts out of the mode graph. So long as all links in the mode graph are individually valid, the combination of those links should also be valid.

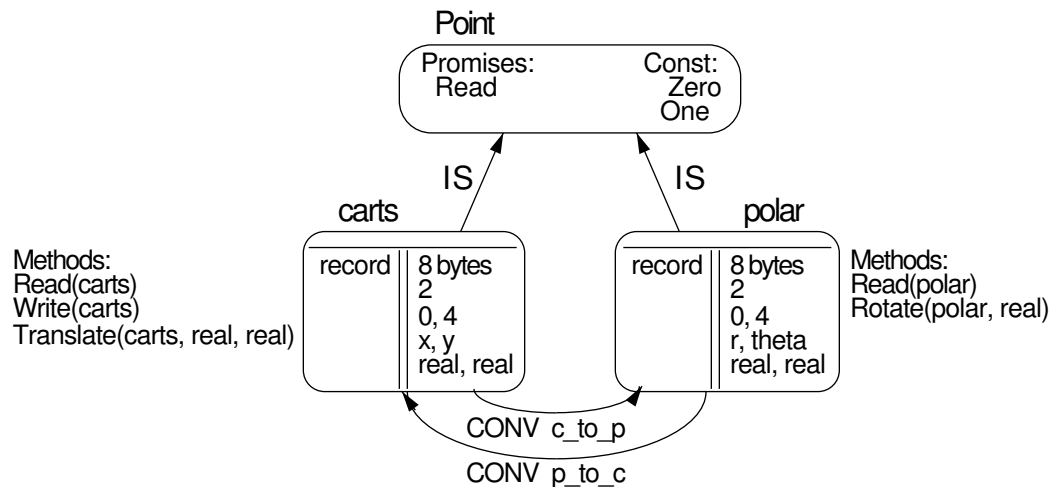
With a class hierarchy, there is only one way to travel from one class to another: up the tree toward the base class. In contrast, a mode graph is an arbitrary graph; one mode can have two or more IS links leading out of it and also have CONV links. We can consider CONV links to be more costly than the other links. The dispatcher must therefore consider more than one path from the starting point to the goal, and dispatching becomes a process of finding the shortest path through a weighted graph, not just traveling up a tree structure.

Where the types of objects are fixed at compile time, all of this work can be done by the compile-time Dispatcher, which then replaces the ordinary type checker. The Dispatcher identifies a type error when the compile-time dispatch fails to find any applicable methods. Where the

⁸Hudak et al. [1992].

⁹A. Fischer and M. Fischer [1973], A. Fischer [1985], and A. Fischer and R. Fischer [1992].

Exhibit 18.22. Mode graph for points on a plane.



types of objects may retain some run-time variability, the Dispatcher can still identify total type mismatches and eliminate many methods as possibilities. Sometimes, though, more than one method is potentially applicable; in this case, the compile-time Dispatcher must return a short list of potential methods, to be further culled by the run-time Dispatcher.

Generics Make Some Problems Easy. Let us define a sample generic mode and show how dispatching would work on it. Assume we are writing a graphics program, in which we are concerned with representing and manipulating points on the plane. There are two good methods for representing points: in Cartesian coordinates, as (x, y) pairs, or in polar coordinates, as (r, θ) pairs. Selector functions `x` and `y` are defined for type `carts`, and `r` and `theta` are defined for type `polar`. Note that selectors are ordinary functions and can be called using ordinary function syntax, even though they are defined as part of the type declaration. This program will need to do many operations on points, including rotation, translation, input, and output. Either representation can be converted to the other using trigonometric functions or square roots.

Translation, that is, moving the point up, down, or sideways, is easy in Cartesian coordinates but difficult in polar. Rotation, that is, moving a point in a circular arc around the origin, is easy in polar but not Cartesian. Output must be in Cartesian, because the terminal screen uses Cartesian coordinates. Some input is in polar, other in Cartesian. The program must use both representations, and the programmer would like to have both available but not worry about representation all the time.

The mode graph for these types is shown in Exhibit 18.22. The generic mode, `point`, promises the function `read` and has two typed submodes, `carts` and `polar`. Any “point” data structures

Exhibit 18.23. Generic calls.

```

VAR p1: polar;
    c1: carts;

Read(c1);           ~~ No problem: Read is defined for type carts,
Read(p1);           ~~ and also for type polar.
If theta(p1)        ~~ The selector theta returns a real.
    > theta(c1)     ~~ The Cartesian argument is coerced to polar.
then Write(p1)      ~~ The polar argument is coerced to Cartesian.
else Write(c1);     ~~ Write is defined for Cartesian.

```

and functions in the surrounding program are declared in terms of the generic mode, `point`.

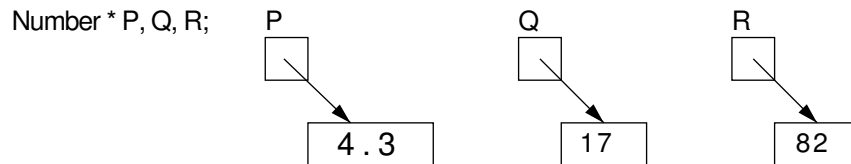
Note that both submodes are represented by pairs of reals, but neither type must ever be cast to the other, because their semantics are totally different. A package for ADT `point` will contain private type declarations for the two point representations and public definitions for the ADT functions. There would also be two data objects (or constant functions) for each type, named `Zero` and `One`.

Exhibit 18.23 illustrates the power of this generic language to simplify the programmer's task.¹⁰ Two variables are declared and data values are read for them. The Dispatcher examines the methods defined for `Read` and chooses the appropriate ones for the two `Read` commands. The `If` statement then applies the selector function `theta` to both points, causing the point `c1` to be coerced before the selection can be done. One of the `Write` statements also coerces its argument.

Everything in this example can be dispatched at compile time. Even though `Read` is a virtual function, it is only used with arguments whose type is fixed and known at compile time, permitting the Dispatcher to identify the correct methods and necessary conversions. Contrast this situation to the problem in Section 18.2.3 which illustrates the need for run-time dispatching. In that application, cells of differing specific types were linked together in one list by pointers with a generic base type. In that case, we could do compile-time dispatching on pointer operations but we still needed run-time dispatching for operations on base-type objects.

This small example is a little artificial, but it illustrates that the programmer using a generic language is, indeed, freed from constant concern about which representation is being used at the moment. In a real program, there would be some reason why a particular point would be represented one way or another, and the time spent performing conversions would be useful. The gain, over traditional languages, is that the programmer can think in terms of the semantics of points, not the semantics of `carts` and `polar`. The resulting code denotes the job of computation more clearly, uncluttered by constant nuts-and-bolts conversion commands.

¹⁰The syntax used here is Pascal-like, so that readers will understand it readily. However, you could certainly not write this code in Pascal!

Exhibit 18.24. Using generic pointers.**18.4.2 Generics Create Some Hard Problems****Dispatching Pointer Arguments**

An interesting question arises when we declare a type that is a pointer to a generic type. For example, let P, Q, and R be of type “pointer to Number” (see Exhibit 18.1) and assume that they have been initialized to point at values as shown in Exhibit 18.24. Now assume we dereference the pointers and add the results by saying $P^{\wedge} + Q^{\wedge}$ or $Q^{\wedge} + R^{\wedge}$. At first glance, it seems that these additions should be straightforward; the promises guarantee that “+” will be defined for all Numbers, and P, Q, and R have the same type.

Looking again, though, we see that the things that P, Q, and R point to may or may not have the same type! Here is a situation in which dispatching must be delayed until run time. The compile-time Dispatcher must make a short list of all the methods for “+” defined on submodes of Number, and the particular method to be used must be selected at the last moment. In our example, the method “+(real,real):real” will be selected for the first addition and “+(integer,integer):integer” will be selected for the second.

A second problem involves possible run-time type incompatibilities. The promises for Number guarantee that every submode will have a definition for “+”. However, “+” is a dyadic function, and these definitions all require that the two arguments to “+” have the same specific type. No methods at all are defined for mismatched operands, like $P^{\wedge} + Q^{\wedge}$ in the example. These are supposed to be handled by the coercion mechanism, and in this case, they would be. The integer operand will be converted to real by the CONV function `i_to_r`. But without the conversion links between the submodes, the computation could not be done, and the dispatch would fail. To avoid run-time errors, the programmer *must* provide enough CONV links to define the relationships among all the submodes of a generic mode.

Using Generic Definitions

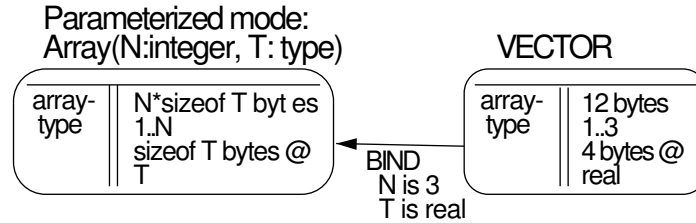
One of the attractions of generic dispatching is that functions defined “high” in the mode graph, for high-level generic modes, can be inherited by all the submodes below them. Inheritance works much as it does in a class hierarchy. But when we use a high-level generic method definition, there is a problem with knowing the type of the result. The method definition can only define the result type in terms of the generic mode to which it is attached. However, when this definition is used,

Exhibit 18.25. What type is returned?

```

k: integer;
r: real;
v: vector;
...
r := v ! k ;

```



it is used with a specific type object, and we would like to know the specific type of the result, not just its generic category.

If we lose track of the specific type that results from an operation, no more compile-time dispatching can be done with that result; all further dispatches for the entire expression must be done at run time if they involve the result of the generic. Unless there is a way to keep track of the specific types involved, generics become too impractical to use extensively, defeating their purpose.

Let us look at one example of this problem. Assume an object belongs to a typed mode, **Vector**, which was derived from a parameterized mode, **Array(N,T)**. **Vector** is connected to **Array** by a **BIND** link that gives the bindings $N=3$, $T=real$ [Exhibit 18.25]. The generic function **Subscript** (written with the symbol “!”) is defined for mode **Array** and has the type:

$$! :: \text{Array}(N, \text{Any}) \rightarrow \text{integer} \rightarrow \text{Any}$$

where mode **Any** is the predefined supermode of all modes.

What type is returned when we call **Subscript** on a **Vector**? The **Subscript** function was defined to return mode **Any**. We know, though, that it will return a real in this case, and it is important for the compile-time dispatcher to be able to use that fact to compile **Vector** expressions.

This is one instance of a very general and difficult problem that arises with parameterized generic modes. We must be able to keep and pass on the specific attributes of the submodes if generic functions defined for supermodes are to be useful. Of course, **Subscript** is predefined in all languages on all array types and is treated as a special case; the programmer does not have to define it. However, a powerful generic language should permit a programmer to redefine **subscript** or to define other functions of this general nature.

To address this specific/generic problem, a language and its translator must have three things:

1. A general type notation that lets the programmer specify type constraints when she or he uses generic modes to declare a function type. For example, the programmer must be able to state that one argument must be the same type as another, or must have a specific structural relationship to another argument's type.
2. Both the dispatcher and the programmer must have access to the information on the **BIND** links and in the type objects of typed submodes. Type operations such as “type of”, “value of binding”, and “dimension of” must be supported.

3. The dispatcher must perform type calculations, using these type operations. The purpose of these calculations is twofold: to guarantee that any specified type constraints are obeyed, and to deduce the most specific information available about the return types.

In the discussion of *Miranda*, in Section 18.3, we saw the use of the symbols “*”, “**”, etc. to denote mode variables, and mode expressions such as “[*]” to denote a list of a generic type. These asterisk symbols and expressions serve purpose (1), above; they permit the programmer to specify constraints (for example, “is the same type as” and “is a list of base type the same as”) in a generic function declaration.

Requirement (2), above, is partially met in *Ada* by providing many predefined type operations which, essentially, let the programmer (or the system) access most of the fields of a type object. Using this information, a programmer-defined function could do things that are built in and non-extensible in traditional languages, such as bounds checks, variations on the subscript function, and the like.

Miranda accomplishes part of goal (2) by permitting the programmer to specify patterns for use by the dispatcher. What seems to be missing is the ability to access the instantiation parameters for parameterized modes. (*Ada* does not have this problem because all instantiation is done before the program is compiled, and a program has full knowledge of the instantiation parameters used.)

Requirement (3) is invisible to the programmer; it does not affect the language syntax or the list of defined operations. However, it affects the semantics in a major way. A dispatcher that is able to do this will permit much more general use of generics and still be able to maintain strong typing.

Mechanisms like these form part of the semantic basis of *Haskell* and *Aleph*, and are sure to play a central role in languages of the future.

Exercises

1. Why can't we use two structurally dissimilar types to represent one external domain in a traditional strongly typed language?
2. How do generic functions solve this problem?
3. What is a generic mode? How is a generic mode different from a C++ base class?
4. How do *Ada*'s predefined modes limit the programmer?
5. What are submodule links? Promises? Supermodes?
6. Why are domains with multiple representations less practical to use than domains with a single representation?
7. What is the danger of providing a general coercion facility?

8. Why should the dispatcher's coercion facility care whether a function is a conversion or a cast?
9. What is a "best match" in dispatching in older languages?
10. Why is dispatching more complex in modern languages?
11. "A subrange type can inherit functions from a base type." Explain.
12. How does a class hierarchy provide broader support for subdomains? Be specific.
13. What is the difference between the handling of a C++ function definition placed inside a class and outside it?
14. When is a derived module considered opaque? How is this accomplished?
15. Why is the compatibility of pointer types throughout the levels of a hierarchy essential?
16. What is a virtual class? Why is it necessary?
17. Why must dispatching of virtual classes be delayed until link or run time? How is this accomplished in C++?
18. What are the benefits of virtual functions? The costs?
19. What is the dispatching rule for function inheritance in C++?
20. Why is the definition of a constructor function for a derived class more complicated than for a base class?
21. What is a polymorphic type? Object? Function?
22. How was polymorphism implemented in older languages?
23. What are the differences between explicit domain testing and strong typing?
24. How do modern functional languages support generics?
25. How are types of objects deduced in Miranda? Explain.
26. In Miranda, what is a pattern? A constructor?