Chapter 8
Structuring Data

Each programming language contains constructs and mechanisms for structuring data. Instead of just the simple sequences of bits in the physical machine, a high-level language provides complex, structured data which more easily lends itself to describing the structure of the problems that are to be solved. These constructs and mechanisms are formed from what is called the *type system* of a language. Far from being an auxiliary aspect, types represent one of the salient characteristics of a programming language and which substantially differentiate one language from another.

In this chapter, we will examine type systems in the general sense, discussing primitive types and the mechanisms used to define new ones. Central to our presentation will be the concept of *type safety*, which will be introduced in Sect. 8.2. We will then tackle the questions of type equivalence and compatibility of types, that is mechanisms which will allow us to use a value of some type in a context requiring another type. We will then discuss polymorphism and overloading. We will conclude the chapter with some questions about storage management (*garbage collection*), which is not, strictly speaking, a topic about data types but which well complements the examination of pointers which we must undertake.

8.1 Data Types

Data types are present in programming languages for at least three different reasons:

1. At the design level, as support for the conceptual organisation;
2. At the program level, as support for correctness;
3. At the translation level, as support for the implementation.

Before entering into detailed discussion of these aspects, which we will do in the coming sections, we give a definition which, as is often the case with programming languages, is not formally precise but suffices to explain the phenomena which we intend studying.

M. Gabbrielli, S. Martini, *Programming Languages: Principles and Paradigms,* Undergraduate Topics in Computer Science,
DOI 10.1007/978-1-84882-914-5_8, © Springer-Verlag London Limited 2010

Definition 8.1 (Data Type) A *data type* is a homogeneous collection of values, effectively presented, equipped with a set of operations which manipulate these values.

We will try to clarify the various terms used in the previous definition. A type is a collection of values, like the integers, or an integral interval. The adjective "homogeneous", for all its informality, suggests that these values must share any structural property which makes them similar to each other. For example, let us take as an example the integers between 5 and 10, inclusive, while we will not consider as a type the collection composed of the integer number 2, the truth value `true` and the rational number $3/4$. Next, such values "come with" the operations which manipulate them. For example, together with the integers, we can consider the usual operations of addition, subtraction, multiplication and division; otherwise, also consider the less common operations such as remainder after integer division and raising to a power. According to our definition, the same set of values, equipped with two distinct sets of operations, forms another data type. The final component of the definition is that it must be "effectively presented", which refers to values. Since we are speaking of languages for describing algorithms, we are interested in values which it is possible to present (write, name) in a finite manner. Real numbers (the "true" ones in mathematics, that is the only complete archimedian ordered field) are *not* effectively presentable because there are real numbers with infinite decimal expansion which cannot be obtained by means of any algorithm. Their approximations in programming languages (`real` or `float`) are only subsets of the rationals.

8.1.1 Types as Support for Conceptual Organisation

The solution to every complex problem has a conceptual structure which often reflects that of the problem. The presence of different types allows the designer to use the type that is most appropriate to each class of concept. For example, a program handling hotel reservations, will handle concepts such as client, date, price, rooms, etc. Each of these concepts can be described as a different type, with its own set of operations. The designer can define new types and associate them with different concepts, even if they are represented by the same values. For example, rooms and prices could be both represented by integers within specified intervals, but their representation as distinct types makes their conceptual difference explicit.

The use of distinct types can be seen both as a design and a documentation tool. When reading a program, it is clear from the declaration of the type that a variable of type "room" has a different role from that of a variable of type "price". In this sense, types are similar to comments, with the important difference that we are dealing with *effectively controllable* comments, as we will see in the next section.

8.1.2 Types for Correctness

Every programming language has its own type-checking rules which regulate the use of types in a program. The most common example is that of an assignment; for a command of the form `x := exp;` to be correct, the majority of languages require that the type of `exp` coincides (or better, is compatible) with the (declared) type of `x`. In a similar fashion, it is forbidden to add integers and records or to call (that is, transfer control to) an object which is not a function or a procedure.

Such constraints are present in languages both to avoid runtime hardware errors (for example, a call to an object that is not a function might cause an address error), and, more likely, to avoid the kinds of logic error frequently hidden beneath type-rule violations. The sum of an integer and a string rarely corresponds to anything sensible.

Programming languages, then, assume that the violation of a type constraint corresponds to a possible semantic error (a design error). The crucial point is that many languages determine that type constraints must all be satisfied before the execution (or to code generation) of a program. This is the role of the *type checker*, a very important component of the static semantic checking phase of a compiler (Sect. 2.3). We have talked about types as if they were effectively controlled comments, because through them, the programmer communicates the legal ways with which the given objects can be used; but (unlike comments), the compiler (or the language's abstract machine) detects and signals every attempted incorrect use of these objects.

It is clear that a program that is correct with respect to the type rules can still be logically incorrect. Types ensure a minimal correctness, which, however, is of considerable help during the development phase of a program. A fairly powerful analogy is that of dimensional control in physics: when a physicist writes a formula, before verifying its correctness using laws, they verify that the dimensions are correct. If the formula is to express velocity, there must be distance over a time; if there is an acceleration, there must be distance over time squared, and so on. If the formula is dimensionally incorrect, no more time should be spent on it because it is certainly incorrect. If, on the other hand, it is dimensionally correct, it *might* be incorrect and must be handled semantically.

Sometimes, however, the type rules can appear too restrictive. A C programmer is used to the free handling of pointers (which are, in C, actual memory locations) and they can find the restrictions on performing arbitrary pointer arithmetic such as those in languages like Java or Pascal unnecessarily restrictive. In this case, the reply by the designers of these two languages is that the benefits of strict control over types considerably outweighs the loss of expressiveness and conciseness.

A more apt example is that of a subprogram that sorts a vector. In many languages, because of the presence of types, it will be necessary to write one routine to order an integer vector, another to order vectors of characters, and still another for vectors of reals, and so on. All these functions are identical as far as algorithm goes and differ only in the declaration of the types of the parameters and variables. The way out, in this case, is to adopt more sophisticated typing rules, which, without renouncing any control, allow one to write a single function which is parameterised

by type. We will see below that languages which allow this kind of *polymorphism* are becoming common.

Let us finally observe that the rules pertaining to types are not always sufficient to guarantee that the constraints they express are satisfied. We give just a single example here. In a language that permits the explicit deallocation of store in a heap, it is possible that references (pointers) are generated that refer to memory that is no longer allocated to the program (*dangling references*). An attempted access using such a reference is an error that can be classified as a type error, but it is not guaranteed that it will be detected and reported by the abstract machine. Let us therefore classify programming languages as existing somewhere between *secure* and *insecure* with respect to types, according to how possible it is that there can be type-constraint violations during program execution that go undetected by the abstract machine.

8.1.3 Types and Implementation

The third principal motivation for the use of types in programming languages is that they are important sources of information for the abstract machine. The first kind of information, clearly, is about the amount of memory to be allocated to various objects. The compiler can allocate one word for an integer, one byte for a boolean value, *n* words for a vector of integers, and so on. When we have types, all of this information is available statically and does not change during execution.

As a consequence of this kind of static allocation during compilation, it is possible to optimize the operations that access an object. In Sect. 5.3, we discussed how access to an allocated variable in an activation record is performed using an offset from the pointer to an activation record, without a runtime search by name. This form of optimisation is possible because the information carried by types allows the static determination of the allocation sizes for almost every object, even for heap-allocated objects. We will soon see that a record is formed from a collection of fields, each of which is characterised by its own name and its own type. For example, using the same notation as in C, the following declaration introduces the `Professor` type, a record with two fields:

```
struct Professor{
   char Name[20];
   int Course_code;
}
```

If now we have a variable, `p`, of type `Professor`, we can access its fields using either the name `p.Name` or the name `p.Course_Code`. However the object `p` is allocated (in the heap or on the stack), access to its fields will always be possible through the use of offsets from the start address of `p` in memory.

8.2 Type Systems

Before going into the detailed treatment, in this section, we introduce a little terminology which we will illustrate in detail in the following sections.

As argued in the previous section, every programming language has its own *type system*, or rather the complex of information and rules which govern the types in that language. More precisely, a type system consists of the following:[1]

1. The set of predefined types of the language.
2. The mechanisms which permit the definition of new types.
3. The mechanisms for the control of types, among which we distinguish the following:
   - Equivalence rules which specify when two formally different types correspond to the same type.
   - Compatibility rules specifying when a value of a one type can be used in a context in which a different type would be required.
   - Rules and techniques for type inference which specify how the language assigns a type to a complex expression based on information about its components.
4. The specification as to whether (or which) constraints are statically or dynamically checked.

A type system (and, by extension, a language) is *type safe*[2] when no program can violate the distinctions between types defined in that language. In other words, a type system is safe when no program, during its execution, can generate an unsignalled error derived from a type violation. Once more, it is not always clear what a type violation is, at least in general. We have already given many examples, such as access to memory that is not allocated to the program, or the call of a non-functional value. We will see below other examples of errors of this kind.

We have defined a type as a pair composed of a set of values and a set of operations. In any particular language, the values of a type can correspond to different syntactic entities (constants, expressions, etc.). Having fixed a programming language, we can classify its types according to how the values of a type can be manipulated and the kinds of syntactic entity that corresponds to these values. Following the classification that we have already seen in the box on page 135, we have values:

- *Denotable*, if they can be associated with a name.
- *Expressible* if they can be the result of a complex expression (that is different from a simple name).
- *Storable* if they can be stored in a variable.

[1]"A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute" [13].

[2]Much of the literature uses the term *strongly typed* in place of type safe.

Let us give some examples. The values of the type of functions from `int` to `int` are denotable in almost all languages because a name can be given using a declaration. For example:

```
int succ (int x){
   return x+1;
}
```

This assigns a name `succ` to the function which computes the successor. Functional values are not in general expressible in common imperative languages because there are no complex expressions returning a function as the result of their evaluation. In the same way, they are not, in general, storable values because it is not possible to assign a function to a variable. The situation is different in languages from other paradigms, for example functional languages (Scheme, ML, Haskell, etc.) in which functional values are both denotable, and expressible or, in some languages, storable. Values of type integer are in general denotable (they can be associated with constants), expressible, and storable.

8.2.1 Static and Dynamic Checking

A language has *static typing* if its checking of type constraints can be conducted on the program text at compile time. Otherwise, it has *dynamic typing* (that is if checking happens at runtime).

Dynamic type checking assumes that every object (value) have a runtime descriptor specifying its type. The abstract machine is responsible for checking that every operation is applied only to operands of the correct type. Often, this can be done by making the compiler generate appropriate checking code that is executed before each operation. It is not difficult to see that dynamic type checking locates type errors but is not efficient, given that operations are intermixed with type checking. In addition, a possible type error is revealed only during execution when the program might be in operation with its end user.

In static type checking, on the other hand, checks are made during compilation. In this scheme, checks are performed and reported to the programmer before the program is sent to the user. When checking is completely static, moreover, the explicit maintenance of type information at execution time is useless because correctness is guaranteed statically *for every execution sequence*. Execution is therefore more efficient, given that checks are not required at runtime. There is, clearly, a price to pay. In the first place, the design of a statically-typed language is more complex than that of a dynamic language, especially if, together with static checking, guaranteed type safety is also desired. In the second place, compilation takes longer and is more complex, a price that one pays willingly, given that compilation takes place only a few times (with respect to the number of times that the program will be executed) and, above all because type checking shortens the testing and debugging phases.

There is, in the end, a third price to pay. This is less clear than the others but is intimately connected with the nature of static type checking. Static types can decree as erroneous, programs that, in reality, do not cause a runtime type error. By way of a simple example, let us consider the following fragment in our pseudocode:

```
int x;
if (0==1) x = "pippo";
else x = 3+4;
```

The first branch of the conditional assigns the integer variable, `x`, to a value that is incompatible with its type but the execution of the fragment causes no error because the condition is never satisfied. However, every static type checker will signal that this fragment is incorrect because the types of the two conditional branches are not the same. Static checking is therefore more *conservative* than dynamic checking. The motivation for this statement is found in the considerations of Chap. 3 on the existence of undecidable problems. In addition to the halting problem, the problem of determining whether a program causes a type error at execution time is undecidable (see the box on p. 204). It follows from this that there exists no static check that can determine all and only those programs that produce errors at runtime. Static typing therefore adopts a prudential position: that of excluding more programs than strictly necessary with the justification that it can therefore guarantee correctness.

As we have already seen more than once in other contexts, static and dynamic type checking represent the two extremes of a spectrum of solutions in which the two methods coexist. Almost every high-level language combines static and dynamic type checks. We promise to return to this topic in due course; meanwhile let us just give a simple example from Pascal, a language traditionally classified as one that uses static type checking. Pascal allows the definition of interval types (see Sect. 8.3.9). For example, `1..10` is the type of the integers between 1 and 10 (inclusive). An expression of an interval type must be checked dynamically to guarantee that its value is properly contained within the interval. More generally, if a language with arrays want to check that the index of an array lies between the bounds of that array, must perform checks at runtime.

8.3 Scalar Types

Scalar (or simple) types are those types whose values are not composed of aggregations of other values. In this section, we will undertake a quick review of the main scalar types found in most common programming languages, while, in the next section, we will be concerned with the types that result from aggregating different values. The details (which we will not give) clearly depend on specific languages. In order to fix our notation, let us assume that we have in our pseudo-language the following way of defining (or declaring) new types:

```
type newtype = expression;
```

Type errors are undecidable

It is not difficult to prove that the problem of determining whether a program will cause a type error during its execution is undecidable. Let us, in fact, make use of what we already know, the undecidability of the halting problem.

Let us consider the following fragment, where `P` is an arbitrary program:

```
int x;
P;
x = "pippo";
```

Under what conditions will this fragment produce a type error as it runs? Only if `P` terminates, in which case it will try to execute the assignment:

```
x = "pippo";
```

which clearly violates the type system. If, on the other hand, `P` does not terminate, it will produce no error because control remains always inside `P` without ever reaching the critical assignment. Therefore, the fragment generates a type error if and only if `P` terminates. If there now existed a general method for deciding whether an arbitrary program causes an type error while executing, we could apply this method to our fragment. Such a method, though, would also be a method for deciding the termination of the (arbitrary) program `P`, something we know to be impossible.

This introduces the name of the type, `newtype`, whose structure is given by `expression`. In C, we would write, for the same meaning:

```
typedef expression newtype;
```

8.3.1 Booleans

The type of logical values, or booleans, is composed of:

- *Values*: The two truth values, *true* and *false*.
- *Operations:* an appropriate selection from the main logical operations: conjunction (and), disjunction (or) and negation (not), equality, exclusive or, etc.

If present (C for example has no type constructed in this fashion), its values can be stored, expressed and denoted. For reasons of addressing, the memory representation does not consist of a single bit but a byte (or possibly more if alignment is required).

8.3.2 Characters

The character type is composed of:

- *Values*: a set of character codes, fixed when the language is defined; the most common of these are ASCII and UNICODE.
- *Operations*: strongly dependent upon the language; we always find equality, comparison and some way to move from a character to its successor (according to the fixed encoding) and/or to its predecessor.

Values can be stored, expressed and denoted. The representation in store will consist of a single byte (ASCII) or of two bytes (UNICODE).

8.3.3 Integers

The type of integer numbers is composed of:

- *Values*: A finite subset of the integers, usually fixed when the language is defined (but there are cases in which it is determined by the abstract machine which can be the cause of some portability problems). Due to representation issues, the interval $[-2^t, 2^t - 1]$ is commonly used.
- *Operations*: the comparisons and an appropriate selection of the main arithmetic operators (addition, subtraction,, multiplication, integer division, remainder after division, exponentiation, etc.).

The values can be stored, expressed and denoted. The representation in memory consists of an even number of bytes (usually 2, 4, or 8), in two's complement form. (Some languages include support for arbitrary-length integers.)

8.3.4 Reals

The so-called real type (or floating point numbers) is composed of:

- *Values:* an appropriate subset of the rational numbers, usually fixed when the language is defined (but there are case in which it is fixed by the specific abstract machine, a matter that deeply affects portability); the structure (size, granularity, etc.) of such a subset depends on the representation adopted.
- *Operations*: comparisons and an appropriate selection of the main numeric operations (addition, subtraction, multiplication, division, exponentiation, square roots, etc.).

The values can be stored, expressed and denoted. The memory representation consists of four, eight and also ten bytes, in floating point format as specified by the IEEE 754 standard (for languages and architectures from 1985).

8.3.5 Fixed Point

The so-called fixed point type for reals is composed of:

Empty or singleton

At first sight, one might be confused by the statement that `void` has one (a single) element rather than none. Let us think a little. We are used to defining a function which "returns nothing" as:

```
void f (...){...}
```

If `void` were the empty set, we could not write a function such as `f`. There exist no functions with an empty codomain, with the unique exception of the function that is everywhere divergent. It is, instead, sensible to assume that in `void`, there is a single element and that this (implicitly) is returned by `f`. Since such an element is unique, we have no (and we must not have any) interest in explicitly saying what it is.

- *Values:* an appropriate subset of the rational numbers, usually fixed when the language is defined; the structure (size, granularity, etc.) of such a subset depends on the representation adopted.
- *Operations:* comparisons and an appropriate selection of the main numeric operations (addition,subtraction, multiplication, division, exponentiation, extraction of square roots, etc.).

The values can be stored, expressed and denoted. The representation in memory consists of four or eight bytes. Values are represented in two's complement, with a fixed number of bits reserved for the decimal part. Reals in fixed point permit compact representation over a wide interval with few precision places.

8.3.6 Complex

The so-called complex type is composed of:

- *Values*: an appropriate subset of the complex numbers, usually fixed by the definition of the language; the structure (size, granularity, etc.) of this subset depends on the adopted representation.
- *Operations*: comparisons and an appropriate selection of the main numerical operations (sum, subtraction, multiplication, division, exponentiation, taking of square roots, etc.).

The values can be stored, expressed and denoted. The representation consists of a pair of floating-point values.

8.3.7 *Void*

In some languages, there exists a primitive type whose semantics is that of having a single value. It is sometimes denoted `void` (even if, semantically, it would be better to call it `unit`, given that it is not the empty set but a singleton):

- *Values*: only one, which can be written as ().
- *Operations*: none.

What is the purpose of a type of this kind? It is used to denote the type of operations that modify the state but return no value. For example, in some languages, (but not in C or Java), assignments have type `void`.

8.3.8 *Enumerations*

In addition to the predefined types, such as those introduced above, we also find in some languages different ways to define new types. Enumerations and intervals are scalar types that are defined by the user.

An enumeration type consists of a fixed set of constants, each characterized by its own name. In our pseudo-language, we could write the following definition

```
type Dwarf = {Bashful, Doc, Dopey, Grumpy, Happy, Sleepy, Sneezy};
```

which introduces a new type with the name `Dwarf` and is a set of seven elements, each one denoted by its own name.

The operations available over an enumeration consist of comparisons and of a mechanism to move from a value to its successor and/or predecessor value (this should be compared with what was said about the character type; in Pascal, the `char` type is, basically, a predefined enumeration).

From a pragmatic viewpoint, enumerations permit the creation of highly legible programs insofar as the names for values constitute a fairly clear form of self documentation of the program. Type checking moreover can be exploited to check that a variable of an enumeration type assumes only the correct values.

Introduced for the first time in Pascal, enumeration types are present in many other languages; the box discusses those in C.

A value of an enumeration type is typically represented by a one-byte integer. The individual values are represented by contiguous values, starting at zero. Some languages (C and Ada for example) allow the programmer to choose the values corresponding to the different elements of an enumeration.

Enumerations in C

In C, our definition of `Dwarf` takes the form:

```
enum Dwarf {Bashful, Doc, Dopey, Grumpy, Happy, Sleepy, Sneezy};
```

Apart from notational variants, the essential point is that in C (but not in C++), such a declaration is substantially equivalent to the following:

```
typedef int Dwarf;
const Dwarf Bashful=0, Doc=1, Dopey=2,
      Grumpy=3, Happy=4, Sleepy=5, Sneezy=6;
```

The type equivalence rules for the language, in other words, allow an integer to be used in place of a `Dwarf` and vice versa. Type checking does not distinguish between the two types and, therefore, better documentation is all that is obtained by the use of an enumeration; stronger type checking is not obtained. In languages derived from Pascal, on the other hand, enumerations and integers are completely different types.

8.3.9 *Intervals*

The values of an interval type form a contiguous subset of the values of another scalar type (the *base type* of the interval). Two examples in Pascal (which was the first language to introduce intervals as well as enumerations) are:

```
type Bingo = 1..90;
     SomeDwarves = Grumpy..Sleepy;
```

In the first case, the `Bingo` type is an interval of 90 elements whose base type is the integer type. The interval `SomeDwarves` is formed from the values `Grumpy`, `Happy` and `Sleepy` and has `Dwarf` as its base type.

As in the case of enumerations, the advantage of using an interval type rather than the corresponding base type is both that it is better for documentation and because it provides a stronger type check. It can be seen that verifying that a value of a certain expression really belongs to the interval must necessarily be made dynamically, even in those languages whose type system is designed for the static checking of type constraints.

As far as representation goes, a compiler can represent a value of an interval type as a one- or two-byte integer according to the number of elements in the interval. In reality, usually an abstract machine will represent the values of an interval in the same way (and in the same number of bytes) in which the base type is represented.

8.3.10 Ordered Types

The boolean, character, integer, enumeration and interval types are examples of *ordered types* (or *discrete* types). They are equipped with a well-defined concept of total order and, above all, possess a concept of predecessor and successor for every element, except for the extreme values. Ordered types are a natural choice for vector indices and for control variables of definite iterations (see Sect. 6.3.3).

8.4 Composite Types

Non-scalar types are said to be *composite* since they are obtained by combining other types using appropriate *constructors*. The most important and common composite types are:

- *Record* (or structure), an collection of values in general of different type.
- *Array* (or vector), a collection of values of the same type.
- *Set*: subsets of a base type, generally ordinal types.
- *Pointer*: l-values which permit access to data of another type.
- *Recursive types*: types defined by recursion, using constants and constructors; particular cases of recursive types are lists, trees, etc.

In the following subsections, we will analyse these types in the same order as in the above list.

8.4.1 Records

A record is a collection formed from a finite number of (in general ordered) elements called *fields*, which are distinguished by name. Each field can be of a type that is different from the others (records are a *heterogeneous* data structure). In the majority of imperative languages, each field behaves like a variable of the same type.[3] The terminology is not, as usual, universal. In Pascal, one talks about records, in C (C++, Algol68, etc.), one talks about *structures* (`struct`); in Java, they do not exist because they are subsumed by the concept of class.

A simple example in our pseudo-language (which is inspired by C's notation) is the following:

```
type Student = struct{
   int year;
   float height;
};
```

[3]In many non-imperative languages, a record is, instead, a finite set of values each of which is accessed by name. Ignoring the question of name-based access, a record type, in such a case, is the cartesian product of the types of its fields.

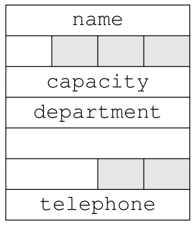


Fig. 8.1 Possible storage arrangement for a record of type `Hall`

Each record of type `Student` is composed of a pair whose first component is an integer, while the second is a real (float).

The only operation usually permitted on a value of a record type is the selection of a component, which can be obtained using the name of the fields. For example, if `s` is a variable that refers to a `Student`, we can assign values to its fields as follows:

```
s.year = 12345;
s.height = 1.85;
```

In many languages, records can be nested, that is the field of a record is permitted to be of a record type.

```
type Hall = struct{
               char name[5];
               int capacity;
               struct{
                  char department[10];
                  int telephone;
               } manager;
};
```

The third field (`manager`) of a record of type `Hall` is a record with two fields (whose type, anonymous, is therefore also a `struct`; the convention is observed that the type occurs before the name being declared). The selection of fields is performed by an obvious extension of the dot notation: if `h` refers to a `Hall`, with `h.manager.telephone`, we can denote the second field of the third field (`manager`) of `h`.

Equality of records is not always defined (Ada permits it but Pascal, C, C++ do not). Similarly, assignment of entire records is not always permitted. In the case in which these operations are not permitted, the language user must explicitly program them, for example by comparing (or assigning) one field at a time.

The order of fields is, in general, significant and is reflected in the way in which records are stored. The fields of a record are stored in contiguous locations, even if reasons of alignment can insert a hole between one field and another. For example, in a 32-bit architecture a record of type `Hall` could be represented by as shown in Fig. 8.1. The `name` field is represented by 5 bytes. The following field, however, being an integer, must be aligned on a word boundary and therefore between `name`

Fig. 8.2 A variant record in C using `union`

```
struct Stud{
   char name[6];
   int reg_no;
   int graduated;
   union{
      int lastyear;
      struct{
         int major;
         int year;
      } major_student;
   } variantfields;
};
```

and `capacity`, 3 empty bytes must be left. A similar thing happens with fields `department` and `telephone`. A record of type `Hall` is therefore represented by 7 words each of 4 bytes, even though only 23 bytes are significant.[4] To avoid these problems, some languages do not ensure that the order of fields will be maintained by the abstract machine, thus allowing the compiler to re-organise fields in order to minimise the number of holes required to enforce alignment. Our record of type `Hall` could be represented in 6 words, with the waste of a single byte. This is a good implementation if the language guarantees a significant level of abstraction between the user and the implementation. In some applications, on the other hand, there is the need to allow users to manipulate the implementation of types. Languages such as C allow designers to allow this kind of operation as well. In such cases, the reorganisation of fields would not be a good design decision.

8.4.2 Variant Records and Unions

A particular form of record is that in which some fields are mutually exclusive. We talk of *variant record* in this case. It is a feature that, with different names and with different syntax and constraints, is provided by many languages (as we will see very shortly, this is also the cause of much complication). In a way different from our usual method of working (which uses one, neutral pseudo-language), we will start with an example in Pascal, one of the languages in which the concept of variant record is present in its clearest form (but is also accompanied by all of the associated problems). In Pascal, we can declare a type `Stud` as follows (the corresponding definition in C, which we will discuss next, is shown in Fig. 8.2):

```
type Stud = record
   name : array [1..6] of char;
   reg_no : integer;
```

[4]The presence of holes is often the reason for which some languages do not permit equality between entire records. A bit-by-bit comparison of two records would distinguish two records which are in reality equal apart from the irrelevant information present in the holes.

```
   case graduated : boolean of
      true: (lastyear : 2000..maxint);
      false:(major : boolean;
             year : (first,second,third)
             )
   end;
```

The first two fields of the record are a vector of 6 characters (`name`) and an integer `reg_no`. The third field, `major` is preceded by the reserved work `case`, which is called the *tag* or *discriminant* of the variant record. A tag `true` indicates that the record has another field: `lastyear` of interval type. A tag `false` indicates, on the other hand, that the record has two more fields: `major` of type boolean and `year` of an enumeration type. The two possibilities which appear between round brackets are the *variants* of the record. In general, the tag can be of any ordinal type and can, therefore, be followed by a number of variants equal in number to the cardinality of this type.

From a semantic viewpoint, in a variant record, only one of the two variants is significant, never both. Which of the two variants is significant is indicated by the value of the tag. The tag and variants can be accessed as if they were any other field. For example, if `s` is one `Stud`,

```
s.graduated := true
```

assigns a value to the tag, while

```
s.lastyear := s.lastyear+1
```

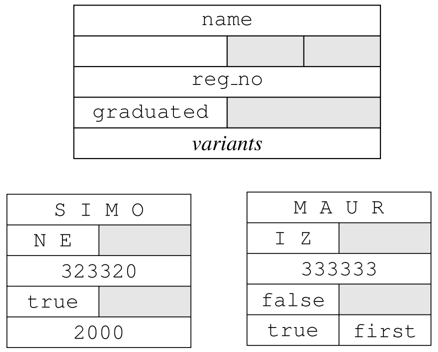
increments by one the first variant.

As far as representation in memory is concerned, all variants share the same region of memory, given that they cannot be active at the same time. In effect, memory saving was one of the reasons for introduction of this data type, even if today this is not so important. Figure 8.3 shows the way in which a `Stud` is allocated.

Unions in C The primitive concept of variant record is not present in C, where we find instead the concept of *union*. A union is analogous to a record (`struct`) in definition and in selection of its fields but with the fundamental difference that only one of the fields of a union can be active at any particular time, given that the fields (which can be of different types) share the same area of storage. It can thus be immediately seen that Fig. 8.2 corresponds exactly to the variant record in Pascal that we just discussed.[5] The single difference (which, from the pragmatic viewpoint, is considerable) is in the necessity in C to add an extra level of naming: the fourth field

[5]Let us observe that, from a notational viewpoint, the outermost `struct` (the one immediately followed by the name `Stud`) is a definition of *type* (precisely the `Stud` type), while the innermost `struct` (which has no name before the braces) is a type expression which determines the type of the field `major_student`.

Fig. 8.3 Possible storage allocations for a variant record



of the structure, named `variantfields`, is of union type and serves to enable access to the real variants. Notice, moreover, that the second variant (which is a pair) must be explicitly declared as a structure with its own name (`major_student`). To access the `major` field, if `s` refers to a `Stud`, it is necessary to write

```
s.variantfields.major_student.major
```

which is certainly much more demanding than Pascal's simple `s.major`.

From the linguistic viewpoint, however, the most revealing difference between Pascal's variant records and their simulation in C using unions is the fact that in C the tag is completely unrelated to the union. As can be seen in Fig. 8.2, `graduated` is a field like all the others and only the programmer's discipline makes it significant with respect to the fields of the union that follows it. The following fragment, for example, is completely legal:

```
s.graduated = 0; // 0 = false: C has no primitive boolean type
s.variantfields.lastyear = 2001;
if (s.graduated) printf("%d", s.variantfields.lastyear);
else print("%d", s.variantfields.major_student.year);
```

Despite being syntactically legal, it is clear that this can pose some semantic problems. The "tag" is false (to indicate that the second union component is active), but a value is assigned to the union's first field, yet then the second component is accessed (for printing).

Variants and Security The above discussion of C unions might seem to suggest that Pascal variant records are to be preferred to the "struct + union" solution adopted for C. There is no doubt that Pascal is more elegant. By merging the two concepts of record and union into the single concept of variant record, the language defines a clearer mechanism, which is more compact and which, one would like to say, is more secure. Unfortunately, this last adjective cannot be used in this context, with reference to type safety; here, Pascal behaves exactly like C.

The fact is that, in Pascal too, the language does not succeed in guaranteeing that there is a formal, verified connection between the value of the tag and the meaning of one of the variants which follow it. The crucial point is that the tag is accessible through an ordinary assignment.

One might think of requiring that the abstract machine should check that every access to a variant happens only if the tag has the correct value (signalling a dynamic error otherwise). Apart from the fact that the language definition does not require such checks, the problem is that this solution would catch some semantic errors but not all. Let us assume, in fact, that we have the following declaration:

```
type Three = 1..3;
var tmp : record
             case which : Three of
                1 : (a : integer);
                2 : (b : boolean);
                3 : (c : char)
          end
```

A hypothetical abstract machine checking the tag prior to access would be able to signal (dynamically) the error in the following program (the error context is analogous to that one we discussed for C above):

```
tmp.which := 1;
tmp.a := 123;
writeln(tmp.c); (* dynamic error *)
```

But the following fragment, on the other hand, would pass unnoticed by all checks, despite the fact that it contains an access to the field `c` that has not yet been assigned a meaningful value:

```
tmp.which := 1;
tmp.a := 123;
tmp.which := 3;
writeln(tmp.c); (* no error, but c not meaningful! *)
```

The situation is even worse, because the tag is not obligatory. The following variant record definition is completely legal:

```
var tmp : record
             case Three of
                1 : (a : integer);
                2 : (b : boolean);
                3 : (c : char)
          end
```

The variants are discriminated with respect to the type `Three`, but in this record, no space is reserved to store the tag. In this case, even the limited form of access check that we have just discussed turns out to be impossible.

Secure union: Algol68

Is it possible to design secure variant records such that they do not threaten the very roots of the language's type system? The reply is certainly yes, provided we are willing to pay the price both linguistically and in terms of efficiency. Let us discuss here in summary form the Algol-68 solution which antedates Pascal's design by some years (Ada is another language with secure variants). Algol-68 allows the definition of union types and requires the abstract machine to follow the evolution of the type of any variable of union type. It does not require an explicit tag:

```
union (int, bool, char) tmp; # tmp of union type #
...
tmp := true; # now tmp is of type boolean #
...
tmp := 123; # now tmp is an integer #
```

For every variable of union type, the abstract machine maintains a hidden type tag which is implicitly set when an assignment occurs. The crucial point is that a union can be used only through a "conformity clause" (a case) which specifies what to do with this variable in all cases. For example:

```
case tmp in
   (int a)  : a := a+1,
   (bool b) : b := not b,
   (char c) : print(c)
esac
```

A conformity clause is the only construct in the language that generate type checks for dynamic types. Beyond the efficiency penalty, it is clear what the linguistic burden of a similar solution would be.

The consequence of all this is that the presence of variant records destroys the security of Pascal's type system. One could then ask if it is worth the effort including such a construct that is so expensive in terms of security in a language which considers typing as one of its primary goals. The reply, given today's climate, is negative: the saving in memory (as well as in conceptual elegance) that variant records provide is not justified by the problems that they cause to the type system.[6] Variant records (or unions) are not present in Modula-3 or in Java.

[6]But it should be noted that Pascal is a language designed at the end of the sixties when the problem of runtime central-memory occupancy was a central concern.

8.4.3 Arrays

An array (or vector) is a finite collection of *elements* of the same type, indexed by an interval of ordinal type.[7] Each element behaves as if it were a variable of the same type. The ordinal interval of the indices is the index type, or the type of the array indices. The type of the elements is the *component type*, or even, with a little imprecision, the *array type*. Since all elements are of the same type, arrays are homogeneous data types.

Arrays are doubtless the most common composite type found in programming languages; they first appeared in FORTRAN, the progenitor of all high-level languages. The syntax and various characteristics of arrays, however, vary considerably from language to language. The fundamental ingredients of array declarations are its name, its index type and the type of its elements. Let us begin with one of the simplest examples we can produce in C:

```
int V[10];
```

The square brackets after the name of the variable indicate that we are dealing with an array formed from 10 elements, each of which is a variable of type integer (from now on, we will simply say: an array of 10 integers). The element type is therefore `int`, while the index type is an interval of 10 elements; using the convention adopted in many languages (C, C++, Java, etc.), this interval starts at 0. In this case, therefore, the index type is the interval 0 to 9. In general, we can assume that a language allows the declaration of the index type to be an arbitrary interval of an ordinal type, for example:

```
int W[21..30];
type Dwarf = {Bashful, Doc, Dopey, Grumpy, Happy, Sleepy,
              Sneezy};
float Z[Dopey..Sneezy];
```

In the first declaration, `W` is an array of 10 integers, with integer indices from 21 to 30. The last line declares `Z`, an array of 4 reals, with indices taken from the `Dwarf` type that run `Dopey` to `Sneezy`.

All languages permit the definition of *multidimensional* arrays, that is arrays indexed by two or more indices:

```
int V[1..10,1..10];
char C[Dopey..Sleepy,0..10,1..10];
```

The array `V` is a 10×10 square integer matrix, with row and column indices running from 1 to 10; `C` is a matrix whose elements are characters and whose bounds are $4 \times 11 \times 10$.

[7]From a semantic viewpoint, an array is a function which has, as domain, an interval for the indices and, as codomain, a type for the array elements.

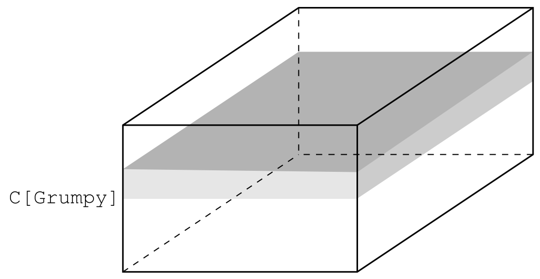


Fig. 8.4 A slice through an array

In some languages, a multidimensional array can be obtained by declaring that the type of the array elements is, at the same time, an array. A possible syntax in our pseudo-language could be the following, so we can declare the above as:

```
int V[1..10][1..10];
char C[Dopey..Sleepy][0..10][1..10];
```

In some languages, the two models (multidimensional array and array of array) are equivalent (one is an abbreviation for the other). This is the case in Pascal. In other languages, only one of the two possibilities is admitted. In C, a multidimensional array is an array of arrays and must be declared as such. Other languages, finally, allow both models, but arrays of arrays support additional operations (*slicing*, as will shortly be seen) that are only defined over multidimensional arrays.

Operations on Arrays The simplest operation permitted on an array is *element selection*. It is performed using an index variable. The most common notation uses square brackets: `W[e]` indicates the element of `W` corresponding to the index denoted by the value of the expression `e`. For multidimensional arrays, it is possible to find `C[Grumpy,1,2]` or `C[Grumpy][1][2]`, depending on whether the emphasis is on multidimensional or arrays of arrays.

Some languages, furthermore, allow operations on arrays in their entirety: assignment, equality, comparisons and also arithmetic operations (performed, in general, elementwise). In other languages, these global operations are just one particular case of operations that allow the selection of interior parts of an array, which is operated on in a global fashion. An *array slice* is a portion of an array composed of contiguous elements. With the declarations above, `V[3]` could indicate the third row of the matrix `V`, and `C[Grumpy]` the plane of the 3-dimensional matrix `C` which is obtained by selecting only its first component (Fig. 8.4). Other languages may allow the selection of more sophisticated slices: diagonals, frames, etc.

Checking The specification of the index type of an array is an integral of its definition. Type checking in the language, therefore, would have to verify that every access of an element of a vector really happens “between the bounds” of the array

and that there is no attempt to access elements that do not exist. With the exception of some special cases, this check can occur only at runtime. A secure language, therefore, will have to ensure that the compiler generates appropriate checks for *every access*. Since this checking will affect the efficiency of the program, some languages, while permitting such checks to be generated, allow their deactivation (for example, Pascal).

Let us observe, by the way, that we are dealing with a of matter of not insignificant importance to the security of a system, understood not just as “type safety”, but as “security” in a real sense. One of the most common attacks, which is also one of the most serious for the security of a system, is called *buffer overflow*. A malicious agent sends messages across the network with the aim of having them read into the buffers of the destination. If the destination does not check that the length of messages does not exceed the capacity of the buffer, the malicious sender can write into an area of memory that is not allocated to the buffers. If the buffer is allocated in an activation record and the sender succeeds in writing into the area the activation record reserves for saving the procedure’s return address, a “return” to any instruction can be made, in particular it can be made to jump to malicious code loaded by the attacker on purpose (for example in the same buffer that caused the overflow). In almost all cases, a buffer overflow attack can be stopped by an abstract machine that checks that every access to an array happens “between the bounds”.

Storage and Calculation of Indices An array is usually stored as a contiguous portion of memory (see Exercise 2 for a non-contiguous allocation technique). For a single-dimensional array, allocation follows the order of the indices. In the case of a multidimensional array, there are two alternative techniques, referred to as storage in *row order* and in *column order*. In row order, two elements are contiguous when they differ in their last index (except in the case of elements at the extremes of a row, or on a plane, etc.). In column order, two elements are contiguous if they differ in the first index (except in the case of extremal elements). Row order is a little more common than column order; the fact that the elements of a row are stored contiguously makes the selection of a row slice easier (this is the slicing operation that is most often provided).

The two storing techniques are not equivalent as far as efficiency is concerned, particularly when there is a cache. Programs that manipulate large multidimensional arrays are often characterised by nested loops that run over such arrays. If the array cannot be stored entirely in a cache, it is important that its elements are accessed “along” the cache, such that the first miss brings into the cache the elements that will be accessed immediately after. If the loop works by row, the cache must also be loaded by row, that is, row order is more convenient; if the loop is columnwise, column order should be preferred.

Once an array is stored in a contiguous fashion, the calculation of the address corresponding to an arbitrary element is not difficult, even if it requires a little arithmetic. Let us consider the generic array of n dimensions, with elements of type `T`.

`T V[`L_1`..`U_1`]...[`L_n`..`U_n`];`

Let S_n be the number of the addressable units (typically a byte) necessary to store a single element of type `T`. Starting from this value, we can successively calculate a series of values which express the quantity of memory required to store increasingly larger slices of `V`. Let us assume we are working in row order, so we have:

$$S_{n-1} = (U_n - L_n + 1)S_n,$$
$$\dots$$
$$S_1 = (U_2 - L_2 + 1)S_2.$$

For example, for $n = 3$, S_2 is the amount of memory required to store a row, while S_1 is the amount of memory required for an entire plane of `V`.

The address of element `V[`$i_1, \dots, i_n$`]` is now obtained by adding to the first address used to store `V` the quantity:

$$(i_1 - L_1)S_1 + \cdots + (i_n - L_n)S_n.$$

In the case in which the number of dimensions (that is, n), and the value of the L_j and U_j are all known at compile time (that is the *form* of the array is static, see next section), it is convenient to reformulate the addressing expression as:

$$i_1 S_1 + \cdots + i_n S_n - (L_1 S_1 + \cdots + L_n S_n). \tag{8.1}$$

Here, the second part (in parentheses), and *a fortiori*, all the S_j are constants that can be determined at compile time. Making use of this formulation, an address is calculated (dynamically) with n multiplications and n additions. The final subtraction disappears whenever all the L_j are zero (which explains why some languages use zero as lower bound for indices).

Form of an Array: Where an Array is Allocated The *shape* of an array is determined by the number of its dimensions and by the interval within which each of them can vary. An important aspect of the definition of a language is the decision about when the form of an array is fixed. Corresponding to the three principal cases, we also have three different models for allocating arrays in memory:

- *Static* shape. Everything is decided at compile time; the array can be stored in the activation record of the block in which its declaration appears (or in the static memory allocated for globals, if used). Let us observe that in this case, the total dimension required for an array is a constant that is known at compile time. The offset from the address of the array's start and the fixed point in the activation record to which the activation record pointer points is also therefore a constant (this was explained in Sect. 5.3.3). Access to an element of the array is differentiated from access to an ordinary (scalar) variable only by the quantity determined by (8.1).
- Shape *fixed at declaration time*. In this case, the shape is not defined at compile time but is recorded and fixed when execution reaches the array declaration (for example, the interval of the index depends on the value of a variable). In this case,

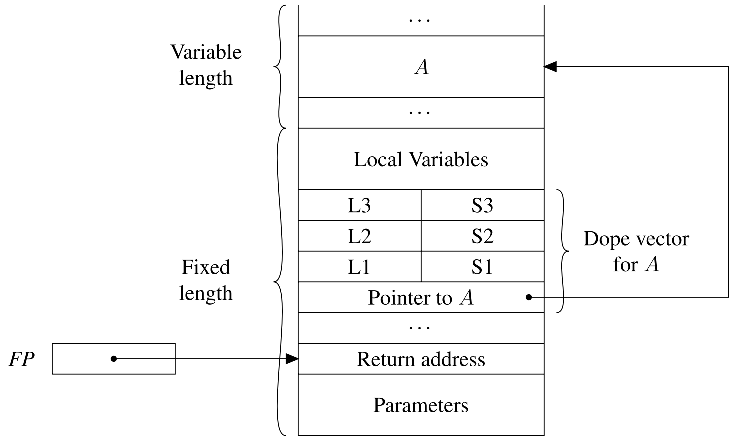


Fig. 8.5 Structure of an activation record with a dope vector

too, the array can be allocated in the activation record of the block in which its declaration appears, but the compiler has no way of knowing what the offset is between the start of the array and the fixed point of the activation record to which the frame pointer refers. This is an unpleasant position because it could have repercussions for other, completely static, data structures. To avoid this problem, an activation record is divided into two parts: one part for fixed-length data, and the other for variable-length data. Access to all the data allocated in the fixed-length part is performed in the usual fashion. On the other hand, when accessing a variable-length data item, an indirection through a variable-length data descriptor is performed. The descriptor is contained in the fixed-length part which is accessed by offset and contains, amongst other things, a pointer to the start of the data structure (in our case, an array). Figure 8.5 shows a scheme for allocation inside an activation record for this. The descriptor for an array is known as a *dope vector*. We will describe dope vectors in a little more detail in a short while.
- *Dynamic* shape. In this case, an array can change shape after its creation as a result of the effects of execution. Stack allocation is no longer possible because the size and structure of an activation record would have to be dynamically modified. Dynamic arrays must therefore be allocated on the heap, while a pointer to the start of the array remains stored in the fixed-length part of the activation record.

To conclude the argument, let us argue that, in addition to the static or dynamic nature of the shape, the decision on allocation of an array must also take its lifetime into account. An array allocated in an activation record has a lifetime limited to that of the block in which it is declared. In languages that allow the creation of arrays with unlimited lifetimes (for example, Java) they must be allocated on the heap.

Dope Vectors The descriptor of an array whose shape is not known statically (and which can either be fixed at execution time or be completely dynamic) is called a

Arrays in Java

In Java, an array is not created at the same time as the declaration of a variable of array type. The declaration `int[] v;` introduces only the name `v`. It is not associated with any data structure (more precisely, this is the same as for any declaration of a variable of class type). The data structure is created on the heap using the predefined `new` operation: `V = new int[10];` creates in the heap a new array of 10 integers and assigns a reference to them to the name `v`.

An array in Java is an object in the strict sense of the term as understood for that language. It has unlimited lifetime.

dope vector. A dope vector, which is usually allocated in the fixed-length part of an activation record contains the following components:

- A pointer to the first location at which the array is stored;
- All the dynamic information required to perform the calculation in expression (8.1): the number of dimensions (also called the *rank* of the array), the value of the lower limit of each dimension (the L_j), the occupation of each dimension (the S_j).

If some of these values are known statically, they are not explicitly stored. The dope vector is initialised when the array is created (using appropriate calculations of the values S_j). To access an element of the array, the dope vector is accessed (by an offset from the frame pointer), expression (8.1) is computed and is added (via an indirect access) to the address of the start of the array.

8.4.4 Sets

Some programming languages (in particular Pascal and its descendants) allow the definition of set types, whose values are composed of subsets of a base type (or universe), usually restricted to an ordinal type. Based on Pascal's syntax but adapting it somewhat, we have:

```
set of char S;
set of Dwarf N;
```

We have a variable `S` which is a subset of the characters and a variable `N` which will contain a subset of the type `Dwarf`, as defined on page 207. Languages allowing sets provide appropriate syntax for assigning a specific subset to a variable, for example:

```
N = (Grumpy, Sleepy);
```

The possible operations on values of set type set are the membership test (the test that an element belongs to a set) and the usual set-theoretic operations of union, intersection and difference; complement is not always provided.

A set is usually represented as a bit vector (the *characteristic* vector of the set) whose length is the same as the cardinality of the base type. For example, a subset of `char`, in an implementation that uses the ASCII 7-bit code, would be represented as 128 bits. If, in the characteristic vector, the jth bit is set, it indicates that the jth element of the base type (in the standard enumeration) is in the set; otherwise, a bit equal to zero indicates that the corresponding element is not in the set. Although this representation permits the highly efficient execution of set-theoretic operations (as bitwise operations on the physical machine), it is clear that it is completely inappropriate for subsets of types whose base type has a cardinality greater than a few hundred. To obviate this problem, languages often limit the types that can be used as base for set types. Or rather, they select other representations (for example, hash tables), which will support the representation of the set in a more compact fashion, even if this reduces the efficiency of the operations performed upon it.

8.4.5 Pointers

Some languages permit the direct manipulation of l-values. The corresponding type is called a pointer type. In general, languages permit the definition of a pointer type for every type: if `T` is a type, it is possible to define the type of "pointers to (variables of type) `T`". It should be noted that, in this context, we will be interested in pointers as far as they are explicitly present in the language, in contrast to the obvious presence of pointers in the abstract machine.

In a language with a reference model for variables, pointer types are not necessary (or required). Every variable is always a reference, or rather it is considered as an l-value, even if this value cannot explicitly be manipulated. In languages with modifiable variables, on the other hand, pointers provide a way of referring to an l-value without automatically dereferencing it. In such languages, one of main uses for pointers is to construct values of recursive type (as linked structures) which are not in general provided as primitives of the language itself.

In our pseudo-language, we use `T*` to indicate a pointer type to objects of type `T` and therefore we declare a pointer as:[8]

```
T* p;
```

[8]The mother tongue C programmer would write the example with a different distribution of spaces: `T *p`, which is read as `*p` is a pointer to `T`. In C, `*` is a modifier of the variable and not of the type (which is apparent when you introduce two pointers in the same declaration: C requires `int *p, *q;`, while we would write `int* p,q;` for uniformity with the other types). Although the C approach seems simpler (particularly when considered with the dereferencing operator), it is semantically more correct (and more uniform with respect to other languages) to see `*` as a type modifier and not a variable modifier.

The values of type `T*` are pointers (from an implementation viewpoint: addresses) to memory locations (that is modifiable variables) which contain values of type `T`. It is not always the case such pointers can refer to arbitrary locations; some languages require that pointers point only to objects allocated in the heap (this is the case with Pascal, in its descendants and in some earlier versions of Ada). Other languages, on the other hand, allow pointers also to point to locations on the system stack or into the global area (this is the case with C++ and later versions of Ada).

There usually exists a canonical value, which is an element of type `T*`, for every `T`, which indicates the null pointer, that is which points to no value. We write `null` for this value. The operations permitted on values of pointer type are usually creation, equality (in particular, equality to `null`), dereferencing (access to the object begin pointed to).

The commonest way of creating a value of pointer type is to use a predefined construct or a library function which allocates an object of an appropriate type on the heap and also returns a reference to the object. This is the purpose of `malloc` in C:

```
int* p;
p = (int *) malloc (sizeof (int));
```

or of `new` in Pascal. For heap management in the presence of explicit allocation, we refer to our discussion in Sect. 5.4.

In some languages, it is possible also to create pointers by applying special operators which return the storage address of an object stored in memory. Again we use C as the model for our examples. C uses the `&` operator to obtain the address of an entity:

```
float r = 3.1415;
float* q;
q = &r;
```

The pointer `q` now points to the location that contains the variable `r`. This is also an example of a pointer pointing to a location on the stack and not in the heap.

Dereferencing a pointer is often indicated by some operator, for example, the `*` in C. Continuing the example above:

```
*p = 33;
r = *q + 1;
```

Here, the first line assigns the value 33 to the object pointed by `p` (which is in the heap). The second line assigns to `r` the value 4.1415 (on the stack). It can be seen that, as a side-effect, the second assignment also modifies the value pointed by `q` (but not `q` itself). It can also be observed that the l-value/r-value distinction for variables in an assignment still remains valid even for a dereferenced pointer. When `*p` is on the left of an assignment, it indicates the l-value that is obtained by dereferencing `p` (that is, the assignment will work on the address contained in the

pointer); when `*p` is on the right of an assignment, it indicates the r-value of the same location.

From the pragmatic viewpoint, pointers play a key role in the definition of so-called *recursive* structures such as lists, trees, etc. Some languages directly allow the definition of recursive types, as will be seen in Sect. 8.4.7. In languages with pointers, recursive data structures can be defined in a natural fashion. For example, the type for the lists of integers[9] can be defined as:

```
type int_list = element*;
type element  = struct {int val;
                        int_list next;};
```

It can be seen that, in order for such a definition to be legal in a hypothetical language, problems related to the use of a name before its definition (which we mentioned in Sect. 4.3.3) must be solved.

Pointer Arithmetic In C and in its descendants, in addition to the operations that we have just discussed, it is possible to perform some arithmetic operations on pointers. It is possible to increment a pointer, subtract one pointer from another (so obtain an offset constant), add an arbitrary value to a pointer. This complex of operations goes under the name of pointer arithmetic. The semantics of these operations, although denoted by the standard arithmetic operators, must be understood with reference to the pointer type on which they operate. By means of an example, let us consider the following fragment:

```
int* p;
int* c;
p = (int *) malloc(sizeof(int));
c = (char *) malloc(sizeof(char));
p = p+1;
c = c+1;
```

If the two `malloc`s return addresses (i for `p` and j for `c`), at the end of the fragment, the value of the two pointers is *not* $i + 1$ and $j + 1$, but, rather, i+`sizeof(int)` and $j +$ `sizeof(char)`. And thus, analogously, for other increment and decrement operations.

It should be clear that pointer arithmetic destroys every hope of type safety in a language. There is no guarantee whatsoever that, at any time during the execution of a program, a variable declared as a pointer to an object of type `T` will still point to an area of store in which a value of type `T` is actually stored.

[9]As we saw in Sect. 4.3.3, a list is a variable-length data structure formed from a possibly empty ordered sequence of elements of some type, in which it is possible to add or remove elements and where only the first element can be directly accessed.

Deallocation We have already seen that one of the most common methods for creating values of a pointer type is to use a function defined as standard in the language to allocate an object on the heap and also return a pointer to it. The deallocation of memory can be explicit or implicit.

In the case of implicit deallocation, the language does not provide the programmer with mechanisms to reclaim memory. The programmer continues to request allocations while there is still available heap. When the heap memory is full, however, the computation need not abort: already allocated values might not be accessible so they can be reused. Consider indeed the following fragment:

```
int* p = (int *) malloc(sizeof(int));
*p = 5;
p = null;
```

The last command, which assigns null to `p`, destroys the only pointer through which this previously allocated area can be accessed. It is possible to equip the abstract machine with a mechanism to recoup such pieces of memory. This technique, which is called garbage collection, is a subject that has been studied a great deal when implementing programming languages. We will examine it in Sect. 8.12.

In the case of explicit deallocation, the language makes available a mechanism with which the programmer can release the memory referred to by a pointer. In C, for example, if `p` points to an object on the heap that was previously allocated by `malloc`, calling the function `free` on `p`, will deallocate the object pointed to by `p` (the store will be put back on the free list if the technique described in Sect. 5.4 is used). It is good practice to assign the value `null` to `p` when this is done. It is a semantic error (with unpredictable result) to invoke `free` on a pointer which does not refer to an object allocated using `malloc` (this can happen if either pointer arithmetic has been used or if the pointer was created using the operator `&`).[10] As we saw in Chap. 4, the most important problem that is raised by deallocation is that it is possible to generate dangling references; that is pointers with a value other than `null` which point to storage that is no longer allocated. The simplest example is probably the following:

```
int* p;
int* q;
p = (int *) malloc(sizeof(int));
*p = 5;
q = p;
free(p);
p = null;
...          //
...          // series of commands not modifying  p or q
```

[10]The point is that `malloc` does not only allocate the space required for the object required, but also a descriptor to the data. The descriptor contains the size of the allocated block and possibly other information. It is by accessing such data that `free` (which takes a single pointer as parameter) can determine what is to be deallocated.

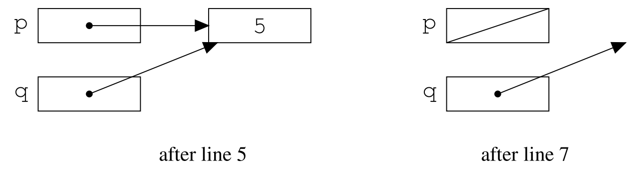


Fig. 8.6 Dangling references

```
...          //
print(*p);  // error that may be trapped
print(*q);  // error that cannot be trapped
```

Figure 8.6 shows the situation in memory as well as the pointers immediately after lines 5 and 7. The pointer `q` has a value that is not `null`. It points to an area of memory that could by now be allocated to other data. We can certainly assume that the abstract machine checks and signals an error every time a pointer dereferences to `null` (as on line 11),[11] while the de-referencing of `q` in line 12 will not be signalled and can be the cause of errors that are as devastating as they are difficult to find.

In languages which allow pointers to refer to objects on the stack, dangling references can also be generated without explicit deallocation. It is sufficient to store the address of a variable local to a function in a nonlocal environment.

```
{int* p;
 void foo(){
    int n;
    p = &n;
 }
 ...
 foo();
 ...    // here p is a dangling reference
}
```

If a language allows dangling references to happen, it is obvious that it cannot be type safe. If a language does not allow pointers to data on the stack, dangling references can be avoided by not having deallocation, both at the level of program design (by using a language with implicit pointer deallocation and garbage collection), or by deallocating nothing even if the programmer requests it: this is the case with some early implementations of Pascal, where the deallocation function `dispose` was implemented as a function with an empty body.

If, though, a language provides explicit deallocation, or pointers to the stack, some techniques are discussed in Sect. 8.11 with which dangling pointers can be rendered harmless, thereby restoring type safety.

[11]This can happen often without affecting efficiency, by making use of mechanisms for protecting addresses on the underlying physical machine.

Arrays and pointers in C

In C, arrays and pointers are considered under certain circumstances to be equivalent. The declaration of an array introduces a name which can be used as a pointer to the whole array. Conversely, if an integer pointer refers to an array, it can also be used as the name of an array:

```
int V[10];
int* W;
W = V;         // W points to the start of array V
V[1] = 5;
W[1] = 5;
*(W+1) = 5;
*(V+1) = 5;
```

The last four assignments in this example are all equivalent. They assign the value 5 to the second element of the array pointed to by both `V` and `W`. (It can be seen how pointer arithmetic is natural here.) In the case of a multidimensional array (stored in row order), pointer arithmetic allows various combinations of operation. In the scope of the declaration `int Z[10][10]`, the following expressions are equivalent: `Z[i][j]`, `(*(Z+1))[j])`, `*(Z[i]+j)` and `*(*(Z+i)+j)`.

The equivalence of arrays and pointers is essential when passing parameters. It will be recalled that C has only call by value. When it passes a vector, it always passes a pointer (by value), never the array itself. The formal parameter can be declared as an array (`int A[]`) or as a pointer (`int *A`). In the case of multidimensional arrays, it is required only to specify the number of elements of the dimensions other than the first in the formal parameter (for example `A[][10]`, or even `(*A)[10]`). This information is necessary for the static generation of the correct code for accessing elements of the vector.

Finally, C permits storage of multidimensional array as a vector of pointers to arrays; for this organisation by row-pointer, see Exercise 2.

8.4.6 Recursive Types

A recursive type is a composite type in which a value of the type can contain a (reference to a) value of the same type. In many languages recursive types can be assimilated to (and in effect are) records in which a field is of the same type as the record itself. The simplest example is perhaps that of a list of integers, which we present in our pseudo-language (adapting Java's notation):

```
type int_list = {int val;
                 int_list next;};
```

To end the recursion, the language can provide a special value, for example `null`, which belongs to any (recursive) type and which we can imagine as having

```
{2,{33,{1,{4,{3,{1,{21,null}}}}}}}
```

Fig. 8.7 A value of type int_list

```
type char_tree = {char val;
                  char_tree left;
                  char_tree right;};
{A,{B,{C,null,null},{D,{E,null,null},null}},{F,null,null}}
```

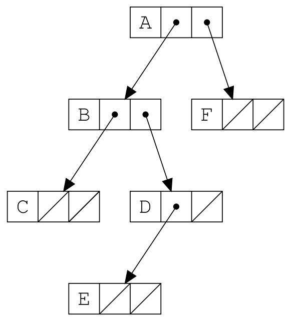


Fig. 8.8 Binary tree over char

the empty value. A value of type `int_list` is therefore a pair: the first element is an integer, the second is a value of type `int_list`, and so on, until one of these `int_list` values is `null` (see the example in Fig. 8.7).

Figure 8.8 shows the definition of a recursive type for binary trees of characters. It also shows a value and the usual graphical representation for trees.

As can be seen from these very simple examples, the possibility of defining recursive types makes the type system of a language flexible and powerful (see Exercise 4). The operations permitted on the values of recursive types are the selection of a component and equality test for `null`.

If the language admits modifiable variables (which is the case in all imperative languages), it is also possible to construct circular values. In purely functional languages, the values of a recursive type are always in the form a tree.

Recursive types are usually represented as structures in the heap. A value of a recursive type corresponds to a linked structure. Every element of the structure is composed of a record in which the recursive reference is implemented as an address to the next record (the one created in the previous recursive step). The representation in the heap of a binary tree shown in Fig. 8.8 corresponds exactly to its graphical representation.

The creation of values of a recursive type (so that they can be associated with a name) is an operation which varies considerably from language to language. In many functional languages, the values of a recursive type are expressible. There exist explicit syntactic constructs for designating their values. An example could be the expression in Fig. 8.7 or in Fig. 8.8 (see, also, the box an ML). In imperative languages, on the other hand, the values of recursive types are constructed by explicit allocation of their components in the heap. To continue our analogy with Java, we have a predefined constructor, `new`, which allows us to allocate instance of the type in the heap:

```
int_list l = new int_list();
```

The part on the right of the assignment symbol creates on the heap a record of the form specified by `int_list` and assigns the name `l` to a reference to this record. At this point, its fields can be initialised. For example:

```
l.val = 2;
l.next = null;
```

It remains to discuss the management of the heap when there are values of recursive type. They are in fact dynamically allocated (explicitly as in Java, or implicitly as in ML) but languages which admit recursive types as primitive do not in general permit explicit deallocation of the values thus created. As in the case of implicit allocation that we discussed in the context of pointers, it is necessary to provide the abstract machine with mechanisms for garbage collection in order to reclaim the chunks of memory that can no longer be accessed.

8.4.7 Functions

Every high-level programming language supports the definition of functions (or procedures). Few traditional languages permit the denotation of function types (that is, give them a name in the language). Let `f` be a function defined as:

```
T f(S s){...}
```

It has the type `S -> T`; more generally, a function with header

```
T f(S1 s1, ..., Sn sn){...}.
```

is of type $S_1 \times \cdots \times S_n$->T.

The values of a functional type are therefore denotable in every language but are rarely expressible (or storable). In addition to definition, the principal operation permitted on a value of functional type is *application*, that is the invocation of a function on some arguments (actual parameters).

In some languages, moreover, the type of functions has the same status of any other type: it is possible to pass functions as arguments to other functions, and to define functions which return functions as results. Such *higher order* languages belong, in particular, to the functional paradigm (which includes, to name but a few: ML, Haskell, Scheme). In these languages, a function is not just a piece of code equipped with its local environment, but may be handled like other data. We will see in Chap. 11, the outline of a possible abstract machine for doing this.

Recursive Types in ML

ML is an important functional language. It is equipped with an extended type system. It is a type-safe language with many interesting properties. The language allows the definition of explicitly recursive types using *constructors*, which are introduced as part of the definition of the type. By way of an example, our definition of list of integers can be written as (ignoring the fact that ML has lists as a predefined type):

```
datatype int_list = Null | CONS of int * int_list;
```

In this definition, `Null` and `CONS` are constructors, that is function symbols to which there corresponds no code, but which serve only to construct, syntactically, terms of the desired type (in this case `int_list`). The vertical line is read as "or". A value of type `int_list` is the constant `Null`, or is a pair (denoted by the constructor `CONS`) composed of an `int` and an `int_list`. Therefore `CONS(2,Null)` is a simple value of type `int_list`. Figure 8.9 shows the ML expression which corresponds to the value in Fig. 8.7.

The language permits the definition of ill-founded recursive types. For example the following definition introduces the type `empty`; it is so-called because it is impossible to construct values of this type:

```
datatype empty =  Void of empty;
```

Indeed, there is no basic constructor which will allow us to construct a value of type `empty` without presupposing another such value.

```
CONS(2,CONS(33,CONS(1,CONS(4,CONS(3,CONS(1,CONS(21,Null)))))))
```

Fig. 8.9 A value of type `int_list` in ML

8.5 Equivalence

Having analysed the principal types that we find in programming languages, the time has come to discuss the rules which govern the correctness of programs with respect to types. The first class of these rules deals with the definition of when two types, which are formally different, are to be considered interchangeable. The rules define an equivalence relation between types. If two types are equivalent, every expression or value of one type is also an expression or value of the other, and vice versa. In the next section, we will discuss compatibility rules which specify when a value of one type can be used in a context in which the value of another type would be expected (but not conversely).

Let us recall that a type is defined in the following form (the details will vary from language to language):

```
type newtype = expression;
```

Relations of preorder and equivalence

A binary relation, $*$, on a set, D, is a subset of the cartesian product of D with itself: $* \subseteq D \times D$. We write $c * d$ for $(c, d) \in *$. A relation is *reflexive* when, for every $d \in D$, it is true that $d * d$. It is symmetric if, for all $c, d \in D$, if $c * d$ holds, then also $d * c$ holds. It is transitive if, for every $c, d, e \in D$, if $c * d$ and $d * e$, then $c * e$. It is antisymmetric if, for every $c, d \in D$, if $c * d$ and $d * c$, then c coincides with d.

If a relation is reflexive and transitive, it is said to be a *preorder*. If a preorder is also symmetric, it is an *equivalence*. If, on the other hand, a preorder is also antisymmetric, it is a *partial order*.

Different languages interpret a definition like this in two different ways which lead to two distinct rules for type equivalence:

- The definition of a type of can be *opaque*, in which case we have equivalence *by name*.
- The definition of the type can be *transparent*, in which case we have *structural* equivalence.

8.5.1 Equivalence by Name

If a language uses opaque type definitions, each new definition introduces a new type which is different from every preceding one. In this case, then, we have equivalence by name, as defined below.

Definition 8.2 (Equivalence by name) Two types are equivalent by name only if they have the same name (so a type is equivalent only to itself).

By way of example, let us consider the following definitions of types:

```
type T1 = 1..10;
type T2 = 1..10;
type T3 = int;
type T4 = int;
```

Four distinct types are introduced. There is no equivalence between them.

However, equivalence by name is too restrictive. Some languages (Pascal for example) adopt the rule of *weak* (or *lax*) equivalence by name. The simple renaming of a type does not generate a new type but only an alias for the same one. In the example above, using a weak equivalence by name, the types `T3` and `T4` are equivalent, but types `T1` and `T2` remain distinct.

Under equivalence by name, each type has a unique definition which occurs at a unique point in the program, a fact which simplifies maintenance. Despite its simplicity, from the programming point of view, equivalence by name is the choice that

Equivalence by Name in Pascal

An aspect of equivalence by name that is not always pleasant is that two expressions of “anonymous” type correspond to two different types (because they do not have names, they cannot have the same name!). In the two following Pascal declarations:

```
var V : array [1..10] of integer;
    W : array [1..10] of integer;
```

`V` and `W` do *not* have the same type. The situation is particularly bad when we declare a formal parameter to a procedure using an expression of anonymous type:

```
procedure f (Z : array [1..10] of integer);
```

In this example, neither `V` nor `W` can be passed as actual parameters to `f` because the formal parameter `Z` is of a (third) type which is distinct from the two previous ones.

most respects the intentions of the designer. If the programmer has introduced two different names for the same type, they will have their reasons which the language respects by maintaining two different types.

Note that equivalence by name is defined with reference to a specific, fixed program and it does not make sense to ask whether two types are equivalent by name “in general”.

8.5.2 Structural Equivalence

A type definition is transparent when the name of the type is just an abbreviation for the expression defining the type. In a language with transparent declarations, two types are equivalent if they have the same structure, that is if, substituting names for the relevant definitions, identical types are obtained. If a language uses transparent type definitions, the equivalence between types that is obtained is said to be *structural equivalence*.

We can give a more precise recursive definition of structural equivalence.

Definition 8.3 (Structural Equivalence) Structural equivalence of types is the (least) equivalence relation satisfying the following properties:

- The name of a type is equivalent to itself;
- If a type `T` is introduced with the definition `type T = expression`, `T` is equivalent to `expression`;
- If two types are constructed by applying the same type constructor to equivalent types, then the two types are equivalent.

By way of example, consider the following definitions:

```
type T1 = int;
type T2 = char;
type T3 = struct{
            T1 a;
            T2 b;
          }
type T4 = struct{
            int a;
            char b;
          }
```

T3 and T4 are structurally equivalent. Some aspects of the definition of equivalence that we have given are deliberately vague or imprecise. For example, it is not clear whether the three following types are equivalent:

```
type S = struct{
           int a;
           int b;
         }
type T = struct{
           int n;
           int m;
         }
type U = struct{
           int m;
           int n;
         }
```

They result from the application of the same constructors, but they have been given names (or field order) that are different. A further subtle question concerns structural equivalence and recursive types:

```
type R1 = struct{
            int a;
            R2  p;
          }
type R2 = struct{
            int a;
            R1  p;
          }
```

Intuitively, we can think that R1 and R2 equivalent, but the type checker will be unable to solve the mutual recursion involved in these definitions (showing the equivalence between R1 and R2 requires a mathematical argument of some sophistication).

Structural equivalence allows us to speak about equivalent types in general, not just those in a specific program. In particular, two equivalent types can always be substituted for each other in any context without altering the meaning of the program in which the substitution occurs (this general property of substitution is often known as *referential transparency*).

It will come as no surprise to the reader to learn that existing languages almost always use some form of combination or variant of the two equivalence rules that we have just defined. We have already seen that Pascal uses (weak) equivalence by name; Java uses equivalence by name, except for arrays, for which it uses structural equivalence; C uses structural equivalence for arrays and for types defined using `typedef`, but not when it involves records (`structs`) and unions, for which it always uses equivalence by name; C++ uses equivalence by name (except when it inherits from C); ML uses structural equivalence, except for types defined using `datatype`; Modula-2 inherits from Pascal its use of equivalence by name, but Modula-3 uses structural equivalence; and so on.

8.6 Compatibility and Conversion

The relation of compatibility between types, which is weaker than equivalence, allows the use of one type in a context in which another has been requested.

Definition 8.4 (Compatibility) We say that type T is compatible with type S, if a value of type T is permitted in any context in which a value of type S would be admissible.

In many languages, it is the rule of compatibility (and not that of equivalence) which controls the correctness of assignment (the type of the right-hand component must be compatible with that of the left-hand one), parameter passing (the type of the actual parameter must be compatible with that of the formal parameter), etc. It is clear that two equivalent types are also compatible with each other but in general the relation of compatibility is not symmetric: the canonical example is the compatibility that exists in many languages (but not in all: ML and Java, for example, do not admit it) between `int` and `float` (but not vice versa). The compatibility relation is therefore a pre-order (that is a reflexive and transitive relation) that is not symmetric, but almost never an ordering (that is it is not symmetric, but not anti-symmetric either). In fact, in a language which admits structural equivalence, two types that are structurally equivalent are compatible but not equal.

The compatibility relation, more than that of equivalence, varies enormously between languages. Without pretending to be complete, let us list some of the possible concepts that we can encounter in order of generality (that is, of the "generosity" of the compatibility relation). A type T can be compatible with S when:

1. Types T and S are equivalent; it is the most restrictive version of compatibility.
2. The values of T form a subset of the values of S: this is a case of an interval type, contained in its own base type (and therefore compatible with it).
3. All the operations on the values of S are also permitted on values of T. The simplest example (which however the does not correspond to compatibility in any of the principal commercial languages) is that of two record types. We assume that we have declared

```
type S = struct{
            int a;
         }
type T = struct{
            int a;
            char b;
         }
```

The only possible operation on values of type `S` is the selection of a field with name `a`; this also makes sense for values of type `T`. There is no inclusion relation between values of `T` and those of `S`, but there is a canonical way to obtain a value of `S` from a value of `T`: take the first component and forget the second. This interpretation of compatibility leads to a particular notion of compatibility based on the concept of *subtype* in object-oriented languages (we will discuss this below in Sect. 10.2.4).

4. The values of `T` correspond *in a canonical fashion* to values of `S`. In addition to the case outlined in the previous point, this is the case that we have already mentioned of `int` being compatible with `float`. (This is not an instance of point (2), above, because, as far as the implementation is concerned, the values of the two types are distinct).
5. Values of `T` can be made to correspond with some values of `S`. Once the requirement of the canonicality of correspondence has been dropped, every type can be made to be compatible with another defining a (conventional) way to transform one value of `T` into one of `S`. Using this very broad notion of compatibility, `float` can be made compatible with `int` by arbitrarily defining the conversion procedure (for example, rounding, truncation, etc.).

In order to manage this collection of different interpretations as a unified whole, we introduce the notion of *type conversion*. This notion will be stated in two distinct ways:

- *Implicit conversion* (also called *coercion* or forced conversion). This is the case in which the abstract machine inserts a conversion when there is no such indication in the high-level language;
- *Explicit conversion* (or *cast*). This is when the conversion is indicated in the text of the program.

Coercions In the presence of compatibility between the types `T` and `S`, the language allows a value of type `T` to occur where a value of type `S` is expected. When this happens, the compiler and/or abstract machines inserts a type conversion between `T` and `S`. We will call this a *type coercion*.

From a syntactic viewpoint, coercion has no significance other than annotating a compatibility. From the implementation viewpoint, however, a coercion can correspond to different things depending upon the notion of compatibility adopted:

1. Type `T` is compatible with `S` and have the same storage representation (at least for values of `T`). In such a case, coercion remains a syntactic matter and causes no code to be generated.

Casts which do not modify representation

In some situations, especially in systems programming, it is useful to be able to change the type of an object without changing its representation in memory. This kind of conversion is obviously forbidden in type-safe languages, but is, though, permitted in many languages (these are often said to be "unchecked" (ADA) or "non-converting type casts").

An example in C using the addressing (`&`) and de-reference (`*`) operators is:

```
int a = 233;
float b = *((float*) &a);
```

First of all, the address of the integer variable `a` is taken, then it is converted explicitly into a pointer to a `float` using the cast `(float *)`; finally, it is dereferenced and assigned to the real value `b`. The entire example is merely annotation for the compiler which does not result in any action by the machine. The bits, which in twos complement represents 233, are now interpreted as an the IEEE 754 representation for floating point numbers (provided that integer and `float` are represented by the same number of bytes).

It is clear that a conversion which does not modify the stored representation is always an extremely delicate operation, as well as being a potential source of hard-to-fix errors.

2. Type `T` is compatible with `S` but there exists a canonical way to transform values of `T` into values of `S`. In this case, the coercion is executed by the abstract machine, which applies precisely the canonical conversion. For example, in the case of `int` and `float`, the abstract machine inserts code to transform (at runtime) the representation in twos complement that is required for integers to that for floating point.
3. The compatibility of `T` with `S` is based on an arbitrary correspondence between values of `T` with those of `S`. In this case, too, the abstract machine inserts code that performs the transformation. This is the case, also, of all those situations where `T` and `S` have the same representation , but `T` is a superset of `S`. For example, `T` is `int` and `S` is an integer interval[12] (note that we are in the symmetric case to the "canonic" case, considered in point 1, in which $T \subseteq S$). Coercion in this case does not transform the representation but dynamically checks (at least if the language is supposed to be type safe) that the value of `T` belong to `S`.

Languages with strong type checking tend to have few coercions (that is few compatibilities) of the last kind. On the other hand, in a language like C, the type system is designed to be by-passed and so permits numerous coercions (from char-

[12] Almost all languages with interval types allow this kind of compatibility for the obvious reason of programming flexibility.

acters to integers, from long reals to short, from long integers to short reals, and so on).

Explicit Conversions Explicit conversions (or *casts*, using the name by which they are known in C and in other languages) are annotations *in the language* which specify that a value of one type must be converted to that of another type. In this case, as well, such a conversion can be either just a syntactic marking or can correspond to code that is executed by the abstract machine, using the scheme for coercions that we have already discussed. In our pseudo-language, we will follow the conventions of the C language family and will denote a cast using brackets:

```
S s = (S) t;
```

Here we are assigning a value `t`, having converted it into type `S`, to a variable of type `S`. Not every explicit conversion is permitted, only those for which the language knows how to implement the conversion. It is clear that we can insert a cast where a compatibility exists. In this case explicit conversion is syntactically useless but must be advised for documentation purposes. Languages with few compatibilities make available, in general, many explicit conversions, which the programmer can use to annotate the program where a change of type is necessary.

Generally, modern languages tend to prefer casts over coercions. They are more useful as far as documentation is concerned; they do not depend upon the syntactic context in which they appear, and, most importantly, they behave better in the presence of overloading and polymorphism (which we will consider shortly).

8.7 Polymorphism

A type system in which any language object (value, function, etc.) has a unique type is said to be *monomorphic*.[13] Here we will be interested in the more general concept given in the next definition.

Definition 8.5 A type system in which the same object can have more than one type is said to be polymorphic.[14] By analogy, we will say that the object is polymorphic when the type system assigns more than one type to it.

Even the more conventional languages contain limited forms of polymorphism. The name `+` in many languages is of type `int`×`int`→`int`, as well as of type `float`×`float`→`float`. The value `null` has the type `T*` for every type `T`. The function `length` (which returns the number of elements of an array) is of type `T[]`→`int` for every type `T`. The list could be continued. In conventional languages,

[13]This is a word derived from the Greek and means "has only a single (*mono*) form (*morphos*)"

[14]"Which has many (*poly*) forms."

however, it is not in general permitted for the user to define polymorphic objects. Let us take as an example, a language with a type system that is fairly inflexible (for example, Pascal or Java but to some extent C also) and let us assume that we want to write a function that sorts a vector of integers. The resulting function could have the signature:

```
void int_sort(int A[])
```

If we now need to sort a vector of characters, we need to define *another* function

```
void char_sort (char C[])
```

which is wholly identical to the previous example except in the type annotations. A polymorphic language would allow the definition of a single function

```
void sort(<T> A[])
```

where `<T>` denotes a generic type which will be specified at a later stage.

In this section, we will analyse the phenomenon of polymorphism. We will start by distinguishing three different forms:

- *Ad hoc polymorphism*, also called *overloading*.
- *Universal polymorphism*, which can be further divided into:
  - Parametric polymorphism, and
  - Subtyping or inclusion polymorphism.

8.7.1 Overloading

Overloading, as its other name (*ad hoc* polymorphism) suggests, is really polymorphism in appearance only. A name is *overloaded* when it corresponds to more than one object and context information must be used to determine which object is denoted by a specific instance of that name. The most common examples of this are:

- The use of the name `+` to indicate either integer or real addition (and sometimes it also denotes concatenation of character sequences).
- The ability to define more than one function (or constructor) with the same name but whose instances differ in the number or type of their parameters.

In the case of overloading, therefore, a single name corresponds to more than one object (in fact, to a finite number of objects). If it is the name of a function, it will be associated with different pieces of code. The ambiguity of the situation is solved statically using type information that is present in the context. From the conceptual viewpoint, we can imagine a kind of pre-analysis of the program which solves cases of overloading by substituting a unambiguous name which uniquely denotes a unique object for each symbol that is overloaded. Overloading is therefore

Polymorphic types and universal quantification

Rather than writing a polymorphic type between angle brackets, a more uniform notation (one which is more suggestive, as well as mathematically more accurate) is that which uses universal quantification. Instead of `<T> -> void`, we write `∀T.T[]->void`.

This is a notation that is well-suited to the description of all the variations on polymorphism which have been proposed; in Sect. 8.7.3 (and again in Sect. 10.4.1) it will be used to analyse subtype polymorphism.

a sort of abbreviation at the syntactic level. It vanishes as soon as we introduce additional information.

Overloading should not be confused with coercion. They are two different mechanisms which solve different problems. On the other hand, in the presence of coercions, it might be not at all clear how a case of overloading should be solved. Consider for example the following expressions:

```
1 + 2
1.0 + 2.0
1 + 2.0
1.0 + 2
```

This quartet of expressions can be interpreted in several ways. First, + is overloaded with four different meanings. Second, + is overloaded with two meanings (integer and real) and coercion is inserted in the final two cases. Next, + merely denotes real addition and in the other three cases, coercions are employed. It is up to the definition of the language to determine which of these interpretations is correct.

8.7.2 Universal Parametric Polymorphism

We begin with a definition that is relatively imprecise but adequate for our needs.

Definition 8.6 A value exhibits universal parametric polymorphism (or parametric polymorphism, more briefly) when it has is an infinite number of different types which can be obtained by instantiating a single schema of general type.

A universal polymorphic function is therefore composed of a single piece of code which operates uniformly on all the instances of its general type (because information about the type is not exploited by the algorithm which the function implements).

Among the examples that we discussed at the start of the section, there are some that fall into this category: the value `null`, which belongs to every type `T*` (that is to every type that can be obtained by substituting an actual type for `T`); the function `void sort(<T> A[])`, which sorts an array of *any type whatsoever*.

Before presenting more examples, we need to extend the notation. Following on from what was suggested in Sect. 8.4.7, we can write the polymorphic type of `sort` as `<T>[]`$\rightarrow$`void`. In this notation, we use angle brackets to indicate that `<T>` is not a usual type but a sort of parameter. By substituting any "concrete" type for the parameter, we obtain a specific type for `sort`. Each way to substitute a type for `<T>` corresponds to one of the infinite number of ways to apply the function: `int[] -> void, char[]-> void`, etc. With this notation, the type of `null` should be more correctly given as `<T>*`.

Let us give another example. A function which exchanges two variables of any type. As with `sort`, if it is written in a language without polymorphism, we would need a `swap` function for every possible variable type. Using universal polymorphism, and assuming that we have at our disposition call by reference, we can write:

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

A polymorphic object can be *instantiated* to a specific type. The instantiation can happen in many different ways, according to the specific mechanism(s) for polymorphism provided by the language. The simplest model is that in which instantiation occurs automatically and is performed by the compiler or the abstract machine:

```
int* k = null;        1
char v,w;
int i,j;              3
...
swap(v,w);            5
swap(i,j);
```

Without requiring additional annotations, on line 1, using information from the context (the assignment is performed on the variable of type `int*`), the type checker instantiates the type of `null` as `int*`. On line 5, `swap` is instantiated to character and, on line 6, to integer.

Parametric polymorphism is general and flexible. It is present in programming languages in two notationally different forms called explicit and implicit parametric polymorphism.

Explicit polymorphism Explicit polymorphism is the kind we have discussed thus far. In the program, there are explicit annotations (our `<T>`) which indicate the types to be considered parameters. This is the kind of polymorphism present in C++ (using the concept of "template") and in Java (from version J2SE 5.0 using the "generic" notion).

Implicit Polymorphism Other languages (in particular functional languages like ML) adopt implicit parametric polymorphism, in which the programmer need not provide any type indications and instead the type checker (or rather, more properly, the module performing type inference—see Sect. 8.9) tries to determine for each object the most general type from which the other types can be obtained by instantiation. The simplest example of a polymorphic function is that of the identity which, using a notation as close as possible to that which we have used this far, can be written as:

```
fun Ide(x){return x;}
```

We will use the word `fun` to indicate that we are defining a function. The rest is totally analogous to the definitions we have already seen, except that there are no type annotations, either for parameters or results. Type inference will assign to the function `Ide`, the type `<T>-><T>`. If the actual parameter is of type `X`, the result will be of type `X`, for any type `X`. The application of a polymorphic function obviously takes place without type indication. `Ide(3)` will be correctly typed as a value of type `int`, while `Ide(true)` be typed as `bool`.

Let us consider here a final example of implicit parametric polymorphism combined with higher order:

```
fun Comp(f,g,x){return f(g(x));}
```

`Comp` applies the composition of its first two parameters (the functions) to the argument `x`. What is the type that can be inferred for `Comp`? Here is the most general possible:

```
(<S>-><T>) x (<R>-><S>) x <R> -> <T>
```

The inferred type is perhaps more general than the reader would have thought.

8.7.3 Subtype Universal Polymorphism

Subtype polymorphism is typically present in object-oriented languages. It is a more limited form of universal polymorphism than parametric polymorphism.

Indeed, here as well, a single object can be assigned an infinite number of different types. They are obtained by instantiating a single most general type. In this case, too, since it is a case of true polymorphism (i.e., universally quantified polymorphism), there is (at least conceptually) a single algorithm which is *uniform in the type*—that is, it does not depend upon the particular structure of the type—that can be instantiated to each of the infinite number of possible types.

However, in the case of subtype polymorphism, not all of the possible instantiations of the schema for the most general type are admissible. Instantiations are limited by a notion of *structural* compatibility between types, that is, using the concept of subtype.

Templates in C++

Parametric polymorphism is obtained in C++ by means of *templates*. These are program schemata which include parameters (of class or of type). The function `swap` could take the following form:

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

The operator, &, attached to the type of a formal parameter indicates that the parameter is passed by reference, a primitive parameter-passing mode in C++ (in which it differs from C). The instantiation of a template is automatic. The call `swap(x,y)` is instantiated to the common type of `x` and `y`.

To be more precise, let us assume that a subtype relation is defined over the language's types. We denote this relation by the symbol `<:`. Therefore, we read `C <: D` as "`C` is a subtype of `D`". For the time being, we can be content with an abstract concept, but in Sect. 10.2.4, we will flesh out this concept in terms of relations between classes in object-oriented languages.

Definition 8.7 A value exhibits subtype (or bounded) polymorphism when there is an infinity of different types which can be obtained by instantiating a general type scheme, substituting for a parameter the subtypes of an assigned type.

To express subtype polymorphism, it is useful to make use of the notation with universal quantifiers that we introduced in the box on page 239. A polymorphic function of type

```
∀T<:D.T-> void
```

can be applied to all the values of any subtype of `D`. Polymorphism therefore is not general but is *limited to subtypes of* `D`.

8.7.4 Remarks on the Implementation

In this book, we cannot possibly explain the implementation of universal polymorphism in detail. We will limit ourselves to two paradigmatic examples of the problems which can be encountered.

Implicit polymorphism in ML

The majority of compilers for ML are interactive. The user enters one expression (or definition) at a time. First of all, the system verifies the correctness of the typing of the input expression and then derives the most general type for it. If the types are correct, the expression is evaluated and the result is displayed (in the case of a definition, its evaluation will extend the environment with the new association). We can define the identity function as:

```
- fun Ide(x) = x;
val Ide = fn : 'a -> 'a
```

In this example, the first line is what is entered by the user ("`-`" is the prompt), while the following line is the system's reply. The name `Ide` is added to the environment and associated with a functional value of type `'a -> 'a`. The identifiers prefixed by '`'`' are *type variables* in ML, that is they can be instantiated. We can request the evaluation of `Ide` as follows:

```
- Ide(4);
val it = 4 : int
- Ide(true);
val it = true : bool
```

Now the functions:

```
- fun Comp(f,g,x) = f(g(x));
val Comp = fn : ('a->'b)*('c->'a)*'c -> 'b
- fun swap(x,y) = let tmp = !x in
                  (x:=!y; y=tmp;);
val swap = fn : ('a ref)*('a ref) -> unit
```

In the case of `swap`, type inference finds that the two arguments cannot be generic values but *variables* (or *references*) of any type whatsoever (`'a ref`). The exclamation mark, "!", is the explicit dereferencing operator. The identifier, `unit`, is the singleton type that is used for expressions with side effects.

```
- val v = ref 0;
val v = ref 0 : int ref
- val w = ref 2;
val w = ref 0 : int ref
- swap(v,w);
val it = () : unit
- !w;
val it = 0 : int
```

In the example, `v` and `w` are initialised, respectively, to a reference to `0` and `2`. Type inference correctly deduces that they must be variables of type `int`. After the swap, the dereferencing of `w` obviously yields `0`.

The first way of handling polymorphism is that of solving it statically at link time[15] (this is the case with C++). When a polymorphic function is called in two different instances, its code is instantiated in two different ways, one for each of the different instances required. Taking again the concrete example of `swap`, there is a local variable, `tmp`, which must be allocated in the activation record when `swap` is called. But the space to allocate for `tmp` depends on its type which is not known when the template for `swap` is originally compiled. At link time, though, polymorphic functions can be identified and their code is modified (instantiated) so as to take into account the actual type(s) with which they are called. The resulting code can then be linked. It can be seen that there can be many copies of the same template (one per instantiation) in the executable code. On the other hand, the execution of a template-using program is as efficient as a program that does not use them, because that templates no longer exist at runtime.

In the case of other polymorphic languages (and ML is one of them), a single version of the code for a polymorphic function is maintained. It is this single piece of code that is executed when one of its instances is required. What is done about the type-dependent information (in the case of `swap`, this is the amount of store to allocate for `tmp`)? It is necessary to change the whole data representation. Instead of directly allocating the data in the activation record, a pointer is maintained that points to the actual data. The data also includes its own descriptors (representing dimensions, structure, etc.). In the case of `swap`, when it is necessary to store a value in `tmp`, a pointer to the variable `x` is accessed and its size (as recorded in the descriptor) is accessed. At this point, the memory necessary for `tmp` is allocated in the heap and a pointer to it is stored in the activation record. The flexibility, uniformity and the conciseness of the code (there are no repeated instances of the same function) are paid for by decreased efficiency—an indirect reference is always required to access a piece of data. We will see more implementation details for subtype polymorphism in Sect. 10.4 when considering object-oriented languages.

8.8 Type Checking and Inference

We have already seen how a language's type checker is responsible for verifying that a program respects the rules (in particular, the compatibility rules) imposed by the type system. In the case of a language with static checking, the type checker is one of the compiler's modules; when type checking is dynamic, the type checker is a module in the runtime system. To perform its task, the type checker must determine the type of the expressions occurring in the program (for example, it must determine whether the type of the expression on the right-hand side of an assignment is compatible with the variable on the left-hand side). To do this, it uses the

[15]Some authors hold that in cases of this kind, we should talk of *generics* rather than polymorphism. We think, on the contrary, that the phenomenon of polymorphism is a syntactic phenomenon of a language, that can be obtained through several different implementation approaches.

type information that the programmer has explicitly placed at critical points in the program (for example, declarations of local names, parameter declarations, explicit conversions, etc.), as well as information that is implicit in the program text (for example, the types of predefined constants, those of numerical constants and so on).

To determine the type of complex expressions, the type checker performs a simple traversal of the program's abstract syntax tree (see the start of Sect. 6.1.3). Starting at the leaves (which represent variables and constants whose types are known), it moves upwards through the tree towards the root, computing the type of the composite expressions on the basis of the information provided by the programmer and the information it obtains from the type system (for example, the type system could establish that + is an operator which, when applied to two expressions of type `int`, permits an expression also of type `int` to be derived, while =, when applied to two arguments of scalar type, gives an expression of type `bool`). In many cases, the information provided by the programmer can turn out to be redundant, for example:

```
int f(int n){return n+1;}
```

Starting with the fact that `n` is of type `int`, it can be easily inferred that the value returned by the function must be of type `int`. The explicit specification given by the programmer of the result type is a redundant specification that the language requires as a last resort so that it can report possible logical errors.

Instead of the simple type checking method we have merely outlined, some languages use a more sophisticated procedure which we might call *type inference*. The progenitor of this family of languages is ML (we have cited this language a number of times above), whose type system is sophisticated and refined. Some ideas presented in work on ML have inspired the design of many different languages, even those of a non-functional kind. In order to introduce the concept of inference in a simplified form, we will again use the definition of function `f`. It is clear that also the specification that the formal parameter, `n`, must be an `int` is redundant. This is because the constant `1` is an integer and + takes two integers and returns an integer, so `n` must be an integer. From a declaration of the form:

```
fun f(n){return n+1;}
```

it is certainly possible automatically to derive that `f` has type `int -> int`. Type inference is exactly this process of the attribution of a type to an expression in which explicit type declarations of its components do not occur. To perform this derivation, the algorithm also works on a syntax tree, again starting at the leaves but, this time, it must take into account the fact that it might not be immediately possible to determine a specific type for any atomic expression (`n`, in this case). In a case like this, the inference algorithm assigns a *type variable* which it might indicate by `'a` (using ML notation). Climbing the tree again and using information present in the context, some constraints for type-variables will be collected. In our case, from the type of `1` and from that of + (which comes from looking up the table of predefined symbols), the constraint `'a = int` is derived.

This form of inference is more general and powerful than the simple type checking algorithm used in languages like Pascal, C or Java. It is capable of determining the most general type of a function, that is it can account for all the polymorphism implicit in an expression. This is done as follows:

1. Assign a type to each node in the syntax tree. For predefined names and constants, use the type stored in the symbol table. For new (i.e., programmer-defined) identifiers, and for every composite expression (which are stored as the internal nodes of the tree), use a type variable (one new variable for every expression or name);
2. Rewalk the syntax tree, generating an (equality) constraint between types at every internal node. For example, if we apply the function symbol, `f`, to which we have previously assigned the type `'a` to the argument `v` of type `'b`, the constraint `'a = 'b -> 'c` will be generated to indicate that `f` must really be a function and that its argument must be of the same type as `v` (`'c` is a new type variable).
3. Resolve the constraints thus gathered using the *unification* algorithm (a powerful, but conceptually simple, instrument for symbolic manipulation which we will discuss in the context of logic languages, in Sect. 12.3).

There are cases in which solving constraint equations does not remove all variables. If we apply the inference algorithm to

```
fun g(n){return n;}
```

we will obtain, as type, `'a -> 'a`. We already know that this is not an error but a positive characteristic: an expression whose most general type contains a type variable is a polymorphic expression.

8.9 Safety: An Evaluation

We started our analysis of types in programming languages with the idea of safety based on types. This same notion has guided us in our examination of the various characteristics of type systems. The time has arrived to take into account what we have understood, classifying programming languages according to their safeness. Let us, then, distinguish between:

1. Non safe languages;
2. Locally safe languages;
3. Safe languages.

In the category of unsafe languages, we find all of those languages whose type system is a more or less just a methodological suggestion to the programmer, in the sense that the language allows the programmer to bypass type checking. Every language that allows access to the representation of a data type belongs to this category, as does every language that allows access to the value of a pointer (pointer arithmetic, that is). C, C++ and all the languages of their family are unsafe.

Locally unsafe languages are those languages whose type system is well regulated and type checked but which contain some, limited, constructs which, when used, allow insecure programs to be written. The languages Algol, Pascal, Ada and many of their descendants belong to this category, provided that the abstract machine really checks types that need dynamic checking, such as checking the limits of interval types (which, it should be recalled, can be used as an index type for arrays). The unsafe part results from the presence of unions (uncontrolled variants) and from the explicit deallocation of memory. Of these two constructs, it is the second that has the greater impact in practice (it is much more common to encounter programs which deallocate memory than those which make significant use of variant records), but it is the former that is the more dangerous. As we will see in the next section, it is possible to equip the abstract machine with appropriate mechanisms to allow it to detect and prevent dangling pointers, even if, for reasons of efficiency, they are almost never used. We have already seen different examples of type-system violations which are possible when using variant records. We have postponed until last the nastiest possibility: that variant records can be used to access and manipulate the value of a pointer. In Pascal, we can write:[16]

```
var v : record
            case bool of
               true  : (i:integer);
               false : (p:^integer)
        end
```

Here, `^integer` is the Pascal notation for the type of pointers to integers. It is now possible to assign a pointer to the variant `p`, manipulate it as an integer using the variant `i` and then use it again as a pointer.

Finally, we have safe languages for which a theorem guarantees that the execution of a typed program can never generate a hidden error "induced by the violation of a type." In this category are languages with dynamic type checking like Lisp and Scheme, as well as languages with static checking such as ML, as well as languages with static checking but with many checks also performed at runtime (as in Java).

8.10 Avoiding Dangling References

In this section, we will tackle the question of which mechanisms can be included in an abstract machine that will dynamically prevent dereference of dangling pointers. This is a problem we first discussed in Sect. 8.4.5. We will first present a radical solution which works in the general case of pointers into the heap or into the stack. We will then consider a somewhat less demanding mechanism that, however, works only for pointers into a heap (and then under some probabilistic assumptions).

[16]In Pascal, pointers are allocated only on the heap by an allocation request (the `new` function).They can be assigned but there is no primitive method to access the value of a pointer.

8.10.1 Tombstone

Using tombstones, an abstract machine can detect every attempt at dereferencing a dangling reference. Every time an object is allocated in the heap (to be then accessed by pointer), the abstract machine also allocates an extra word in memory. This word is called the *tombstone*. In a similar fashion, a tombstone is allocated every time that a stack-referencing pointer is allocated (that is, to a certain approximation, every time that the "indirection-to" operator (`&`) is used). The tombstone is initialised with the address of the allocated object and the pointer receives the address of the tombstone. When a pointer is dereferenced, the abstract machine inserts a second level of indirection, so that it first accesses the tombstone and then uses what it finds there to access the object that is pointed to. When one pointer is assigned to another, it is the contents of the pointer (and not the tombstone) that is modified. Figure 8.10 shows this operation graphically.

On deallocation of an object, or when an indirection on the stack becomes invalid because it is part of an activation record that has been popped,[17] the associated tombstone becomes invalidated, and a special value is stored in it to signal that the data to which the pointer refers is dead (this is where the name of this technique derives from). An appropriate choice of such a value must be made so that every attempt to access the contents of an invalid tombstone is captured by the address-protection mechanism of the underlying physical machine.

Tombstones are allocated in a particular area of memory in the abstract machine (this, appropriately enough, is called the *cemetery*). The cemetery can be managed more efficiently than the heap because all tombstones are of the same size.

For all its simplicity, the tombstone mechanism imposes a heavy cost in terms both of time and space. As far as time is concerned, we must consider the time required for the creation of tombstones, that for their checking (which can be reduced, as we have seen, if an underlying hardware protection mechanism can be used) and, most importantly, the cost of two levels of indirection. Tombstones are also expensive in terms of storage. They require a word of memory for every object allocated on the heap; a word is also required every time a pointer into the stack is created. If there are lots of allocations of small objects, the percentage of memory required by tombstones can be significant. Moreover, invalid tombstones always remain allocated, with the consequence that they might exhaust the space available in the cemetery, even though there might a great deal of space available on the heap. To prevent this last problem, it is possible to reuse those tombstones that are no longer in use (that is, those to which no pointer points) using a small reference-counting garbage collector (Sect. 8.11.1); this further increases the time cost of the mechanism.

[17]The operation can be quite opaque. Consider the case of a variable passed by reference to a function. Inside the function, a pointer is created to point to the variable. The pointer will be created using an associated tombstone. The tombstone must be invalidated when the lifetime of the variable forming the actual parameter ends, not when the function terminates.

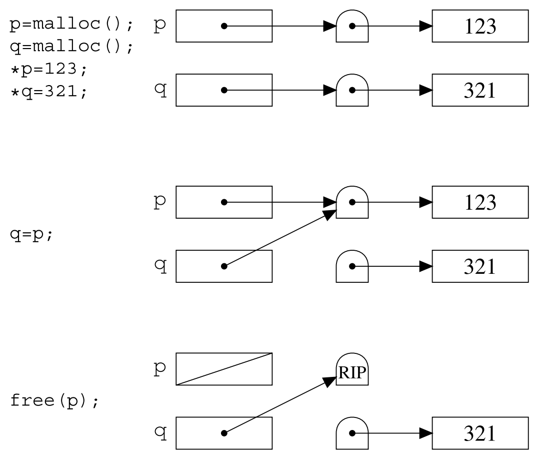


Fig. 8.10 Tombstones

8.10.2 Locks and Keys

An alternative to the tombstone technique is called *locks and keys*. This solves the problem of dangling references into a heap using a mechanism which does not suffer from the accumulated problems of tombstones.

Every time an object is created on the heap, the object is associated with a *lock* which is a word of memory in which an arbitrary value is stored. (Strictly speaking, it should be a value that can be sequentially incremented every time an object is allocated, but which avoids values such as 0 and 1, the code of frequently used characters, and so on.) In this approach, a pointer is composed of a pair: the address proper and a *key* (a word of memory that will be initialised with the value of the lock corresponding to the object being pointed to). When one pointer is assigned to another, the whole pair is assigned. Every time the pointer is dereferenced, the abstract machine checks that the key opens the lock. In other words, it verifies that the information contained in the key coincides with that in the lock. In the case in which the values are not the same, an error is signalled. When an object is deallocated, its lock is invalidated and some canonical value (for example, zero) is stored, so that all the keys which previously opened it now cause an error (see Fig. 8.11). Clearly, it can happen that the area of memory previously used as a key is reallocated (for another lock or for some other structure). It is statistically highly improbable that an error will be signalled as a result of randomly finding an ex-lock that has the same value that it had prior to being clearing.

Locks and keys also have a non-negligible cost. In terms of space, they cost even more than tombstones because it is necessary to associate an additional word with each pointer. On the other hand, locks as well as keys are deallocated at the same time as the object or the pointer of which they are a part. From an efficiency viewpoint, it is necessary to take into account both the cost of creation, and, more

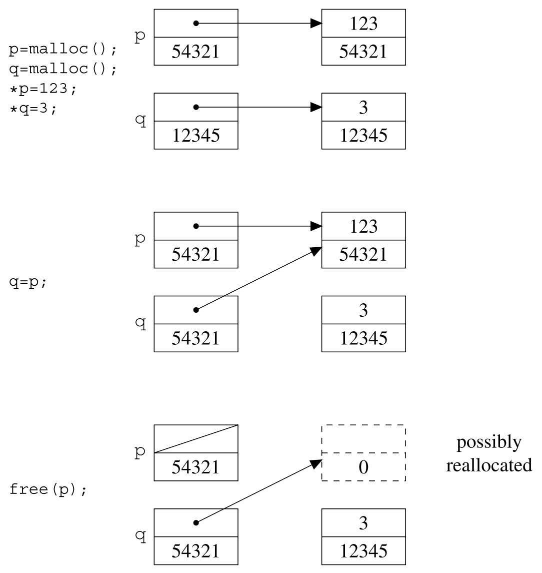


Fig. 8.11 Locks and keys

importantly, the high cost of pointer assignment and of determining whether the key opens a lock, something that happens every time a pointer is dereferenced.

8.11 Garbage Collection

In languages without explicit memory deallocation, it is necessary to equip the abstract machine with a mechanism which could automatically reclaim the memory allocated on the heap that is no longer used. This is done by a *garbage collector*, introduced for the first time in LISP around 1960, and since then included in many languages initially mostly functional, and later imperative. Java has a powerful and efficient garbage collector.

From the logical point of view, the operation of a garbage collector consists of two phases:

1. Distinguish those objects that are still alive from those no longer in use (*garbage detection*);
2. Collect those objects known no longer to be in use, so that the program can reuse them.

In practice these two phases are not always temporally separate, and the technique for reclaiming objects essentially depends on the one in which the objects no longer in use are discovered. We will see, then, that the "no longer in use" concept in a garbage collector is often a conservative approximation. For reasons of efficiency,[18] not all the objects that can be used again are, in reality, determined by the garbage collector to be so.

The educational aim of this text does not permit us to describe a garbage collector for a real language, or to provide an exhaustive overview of the different techniques available (there are bibliographic references to exhaustive treatments). We will limit ourselves to presenting in some detail the main points in the light of the most common techniques. Real garbage collectors are variations on these techniques. In particular the collectors that are of greatest interest today are based upon some form of incremental reclamation of memory and are beyond the scope of this book.

We can classify classical garbage collectors according to how they determine whether an object is no longer in use. We will have, therefore, collectors based on *reference counting*, as well as *marking* and *copying*. In the next few sections, we will present these collectors. We will discuss these techniques in terms of pointers and objects but the argument can equally be applied to languages without pointers which use the reference variable model.

Finally, we will see that all the techniques that we consider need to be able to recognize those fields inside an object that correspond to pointers. If objects are created as instances of statically-defined types (so it is statically known where pointers are inside instances), the compiler can generate a descriptor containing offsets to the pointers in each type. Each object in the heap is associated with the type of which it is an instance (for example, via a pointer to its type descriptor). When an object is to be deallocated, the garbage collector accesses the type descriptor to access the pointers inside the object. Similar techniques are used to recognise which words in an activation record are pointers. If types are only known dynamically, descriptors have to be completely dynamic and allocated together with the object.

8.11.1 Reference Counting

A simple way to tell whether an object is not in use is to determine whether there are any pointers to it. This is the reference counting technique. It defines what is probably the easiest way of implementing a garbage collector.

When an object is created on the heap, an integer is allocated at the same time. This integer is the *reference counter* or *reference count* for the object. The reference count is inaccessible to the programmer. For each object, the abstract machine must ensure that the counter contains the number of pointers to this object that are still active.

In order to do this, when an object is created (that is, a pointer for it is assigned for the first time), its counter is initialised to 1. When we have a pointer assignment:

```
p = q;
```

[18] As well as decidability.

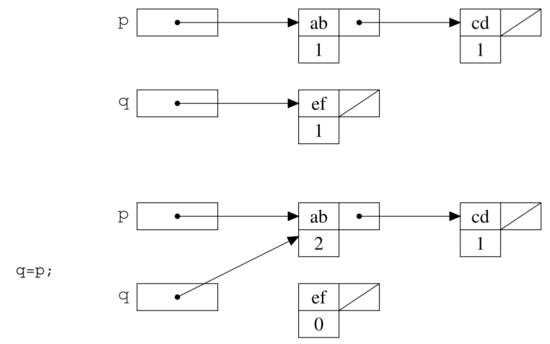


Fig. 8.12 Reference counting

the counter belonging the object pointed to by q is incremented by one, while the counter of object pointed to by p is decremented by one. When a local environment is exited, the counters of all the objects pointed to by pointers local to the environment are decremented. Figure 8.12 shows operation of this technique in diagrammatic form.

When a counter returns to the value 0, its associated object can be deallocated and returned to the free list. This object, moreover, might contain internal pointers, so before returning the memory occupied by an object to the free list, the abstract machine follows the pointers inside the object and decrements by one the counters of the objects they point to, recursively collecting all the objects whose counter has reached 0.

From the viewpoint of the abstract division into two phases of a garbage collector, the update and checking of counters implement the garbage-detection phase. The collection phase occurs when the counter reaches zero.

A clear advantage of this technique is that it is *incremental* because checking and collection are interleaved with the normal operation of the program. With a few adjustments, real-time systems (that is systems where there are absolute deadlines to response time) can employ this technique.

The biggest defect, at least in principle, of this technique is in its inability to deallocate circular structures. Figure 8.13 shows a case in which a circular structure has no more access paths. However it cannot be collected because, clearly, its counters are not zero. It can be seen that the problem does not reside so much in the algorithm, as in the definition of what a useless object is. It is clear that all objects in a circular structure are not usable any more, but this is not at all captured by the definition of not being pointed at.

Reference counting, despite its simplicity, is also fairly inefficient. Its cost is proportional to the combined work performed by the program (and not to the size of the heap or to the percentage of it in use or not in use at any time). One particular case is that of updating the counters of parameters of pointer type which are passed to functions that execute only for a short time. These counters are allocated and, after a particularly short time, have their value returned to zero.

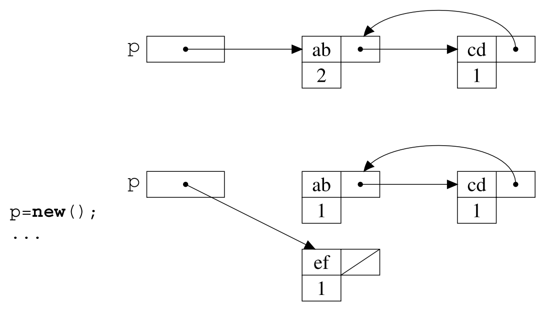


Fig. 8.13 Circular structures with reference counts

8.11.2 *Mark and Sweep*

The *mark and sweep* method takes its name from the way in which the two abstract phases we mentioned at the start are implemented:

- *Mark*. To recognize that something is unused, all the objects in the heap are traversed, marking each of them has "not in use". Then, starting from the pointers which are active and present on the stack (the *root set*), all the data structures present in the heap are traversed recursively (the search is usually performed depth-first) and every object that is encountered is marked as "in use".
- *Sweep*. The heap is swept—all blocks marked "in use" are left alone, while those marked "not in use" are returned to the free list.

It can be seen that to implement both phases, it is necessary to be able to recognize allocated blocks in the heap. Descriptors might be necessary to give the size (and organisation) of every allocated block.

Unlike the reference-counting garbage collector, a mark and sweep collector is not incremental. It it will be invoked only when the free memory available in the heap is close to being exhausted. The user of the program can therefore experience a significant degradation in response time while waiting for the garbage collector to finish.

The mark and sweep technique suffers from three main defects. In the first place, and this is also true for reference counting, it is asymptotically the cause of external fragmentation (see Sect. 5.4.2): live and no longer live objects are arbitrarily mixed in the heap which can make allocating a large object difficult, even if many small blocks are available. The second problem is efficiency. It requires time proportional to the total length of the heap, independent of the percentages of used and free space. The third problem relates to locality of reference. Objects that are "in use" remain in their place, but it is possible that objects contiguous with them will be collected and new objects allocated in their place. During execution, we find that objects with very different ages appear next to each other. This, in general, drastically reduces locality of reference, with the usual degradation in performance observed in systems with memory hierarchies.

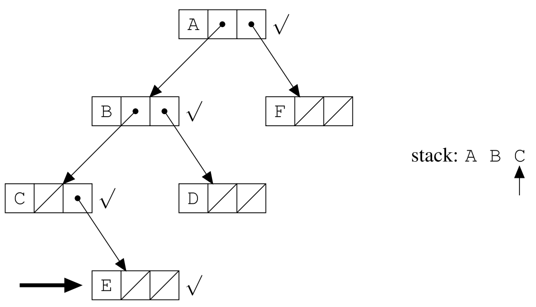


Fig. 8.14 A stack is necessary for depth-first traversal

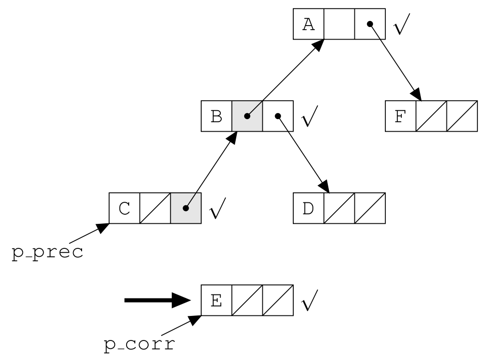
8.11.3 *Interlude: Pointer Reversal*

Without some precautions, every marking technique runs the risk of being completely unusable in a garbage collector. The collector, indeed, goes into action when the memory is near to being exhausted, while the marking phase consists of the recursive traversal of a graph, which requires the use of a stack to store the return pointers.[19] It is necessary, when marking a graph under these circumstances, to use carefully the unused space present in pointers, using a technique called *pointer reversal*.

As shown in Fig. 8.14, when visiting a chained structure, it is necessary to mark a node and recursively visit the substructures whose pointers are part of that node. In this recursive scheme, it is necessary to store the addresses of blocks when they are visited in a stack, so that they can be revisited when the end of the structure is reached. In the Figure, using depth-first search, node E has been reached (marked blocks are indicated with a tick). The stack contains nodes A, B and C. After visiting E, it takes C from the stack and follows any pointers leading from this node; since there are none, it takes B from the stack and follows the remaining pointer, pushing B onto the stack, and visiting D. Using pointer reversal, this stack is stored in the pointers that form the structure, as shown in Fig. 8.15. When the end of a substructure is reached, (and a pop of the stack is required), the pointer is returned to its original value, so that at the end of the visit, the structure is exactly the same as it was at the start (apart from marking, clearly). In the figure, we are visiting node E. It can be seen how the stack in Fig. 8.14 is stored in space internal to the structure itself (pointers marked with a grey background). Only two pointers (`p_prec` and `p_curr`) are required to perform the visit. After visiting E, using `p_prec`, the structure is retraversed in a single step. Using the reversed pointer, we return to B, resetting the correct value of the pointer in C using `p_curr`.

[19]In many abstract machines, stack and heap are implemented in a single area of memory, with stack and heap growing in opposite directions starting from the two ends. In such a case, when there is no space in the heap, there is no space for the stack.

Fig. 8.15 Pointer reversal



8.11.4 Mark and Compact

To avoid the fragmentation caused by the mark and sweep technique, we can modify the sweep phase and convert it into a compaction phase. Live objects are moved so that they are contiguous and thereby leave a contiguous block of free memory. Compaction can be performed by moving linearly across the heap, moving every live block encountered and making it contiguous with the previous block. At the end, all free blocks are contiguous, as are all unused blocks.

This is a technique which, like mark and sweep, requires more than one pass over the heap (and the time required is therefore proportional to the heap size). Compaction, on its own, requires two or three passes. The first is to compute the new position to be taken by the live blocks; a second updates the internal pointers and a third actually moves objects. It is, therefore, a technique that is substantially more expensive than mark and sweep if there are many objects to be moved.

On the other hand, compaction has optimal effect on fragmentation and on locality. It supports treating the free list as a single block. New objects are allocated from the free list by a simple pointer subtraction.

8.11.5 Copy

In garbage collectors based on copy there is no phase during which the "garbage" is marked. There is just the copying and compaction of live blocks. The lack of an explicit mark phase and the completely different way space is handled make its costs substantially different from those of algorithms based on marking.

In the simplest copy-base garbage collector (called a *stop and copy* collector), the heap is divided into two equally-sized parts (two *semi-spaces*). During normal execution, only one of the two semi-spaces is in use. Memory is allocated at one end of the semi-space, while free memory consists of a unique contiguous block which reduces its size every time there is an allocation (see Fig. 8.16). Allocation is extremely efficient and there is no fragmentation.

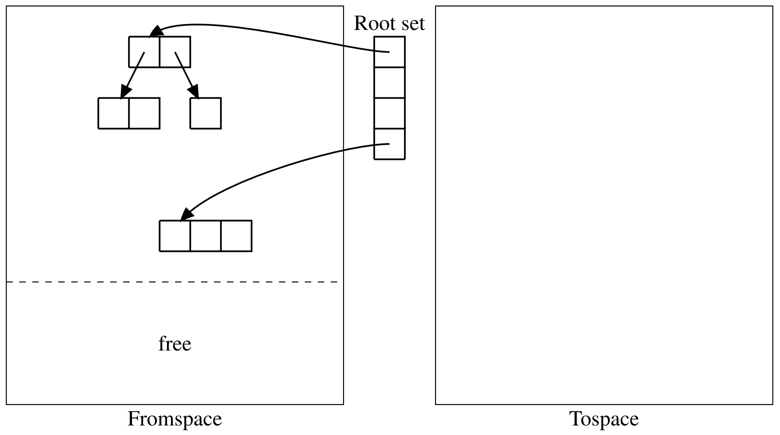


Fig. 8.16 Stop and copy before a call to the garbage collector

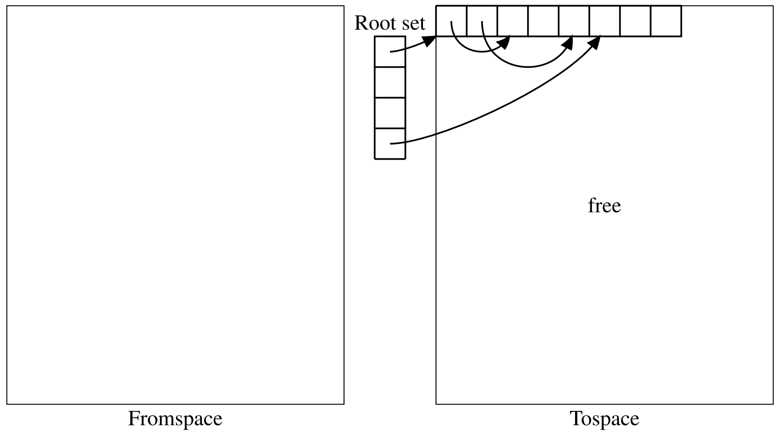


Fig. 8.17 Stop and copy after the execution of the garbage collector

When the memory in the semi-space is exhausted, the garbage collector is invoked. Starting with pointers in the stack (the root set), it begins visiting the chain of structures held in the current semi-space (the *fromspace*), copying the structures one by one into the other semi-space (the *tospace*), compacting them at one end of the latter (see Fig. 8.17). At the end of this process, the role of the two semi-spaces is swapped and execution returns to the user program.

The visit and copy of the live part can be executed in an efficient manner using the simple technique known as Cheney's algorithm (Fig. 8.18). Initially, all the objects which are immediately reachable from the root set are copied. This first set of

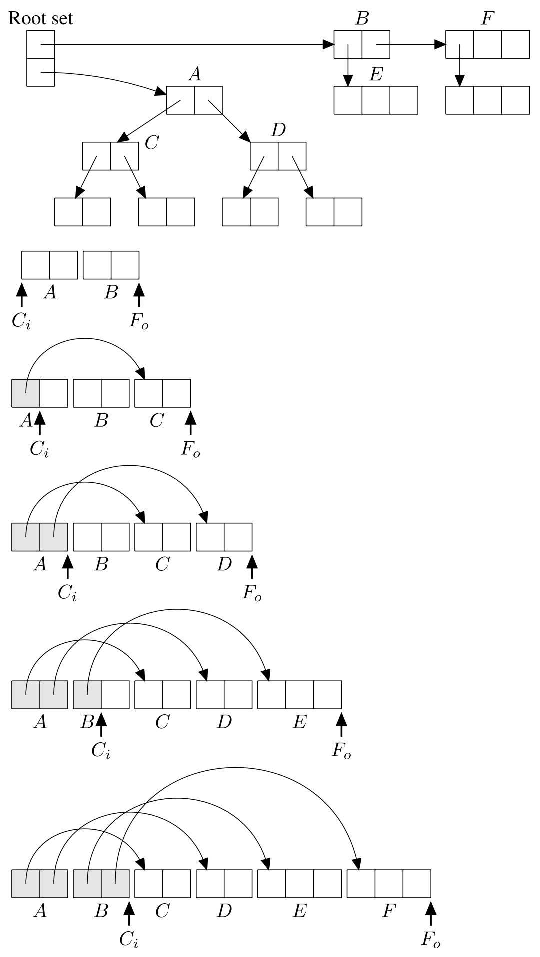


Fig. 8.18 Cheney's algorithm

objects is copied in a contiguous fashion into the tospace; it is handled as if it were a queue. Consider now the first of these objects, and add to the end of the queue (that is, copy into the tospace) the objects pointed to by the pointers present in the first object, while, at the same time, these pointers are modified. In this way, we have copied into the tospace all the first object's children. We keep processing the queue until it remains empty. At this point, in the tospace we have a copy of the live objects in the fromspace.[20]

[20]Some precautions must be taken to prevent objects accessible via multiple pointers being copied more than once.

A stop and copy garbage collector can be made arbitrarily efficient, provided that there is enough memory for the two semi-spaces. In fact, the time required for a stop and copy is proportional to the number of live objects present in the heap. It is not unreasonable to assume that this quantity will be approximately constant at any moment during the execution of the program. If we increase the memory for the two semi-spaces, we will decrease the frequency with which the collector is called and, therefore, the total cost of garbage collection.

8.12 Chapter Summary

This chapter has dealt with a crucial aspect in the definition of programming languages: the organisation of data in abstract structures called data types. The main points can be summarised as follows.

- *Definition of type* as a set of values and operations and the role of types in design, implementation and execution of programs.
- *Type systems* as the set of constructs and mechanisms that regulate and define the use of types in a programming language.
- The distinction between *dynamic* and *static* type checking.
- The concept of type-safe systems, that is safe with respect to types.
- The primary *scalar types*, some of which are *discrete* types.
- The primary *composite types*, among which we have discussed *records*, *variant records* and *unions*, *arrays* and *pointers* in detail. For each of these types, we have also presented the primary storage techniques.
- The concept of *type equivalence*, distinguishing between equivalence by name and structural equivalence.
- The concept of *compatibility* and the related concepts of coercion and conversion.
- The concept of *overloading*, when a single name denotes more than one object and static disambiguation.
- The concept of *universal polymorphism*, when a single name denotes an object that belongs to many different types, finally distinguishing between parametric and subtype polymorphism.
- *Type inference*, that is mechanisms that allow the type of a complex expression to be determined from the types of its elementary types.
- Techniques for runtime checking for *dangling references*: tombstones and locks and keys.
- Techniques for *garbage collection*, that is the automatic recovery of memory, briefly presenting collectors based on reference counters, mark and sweep, mark and compact and copy.

Types are the core of a programming language. Many languages have similar constructs for sequence control but are differentiated by their type systems. It is not possible to understand the essential aspects of other programming paradigms, such as object orientation, without a deep understanding of the questions addressed in this chapter.

8.13 Bibliographic Notes

Ample treatments of programming language types can be found in [14] and the rather older [4]. Review articles which introduce the mathematical formalisms necessary for research in this area are [2, 11]. A larger treatment of the same arguments is to be found in [13]. On overloading and polymorphism in type systems, a good, clear review is [3] (which we have largely followed in this chapter).

Tombstones originally appeared in [8] (also see, by the same author, [9]). Fisher and Leblanc [5] proposed locks and keys, as well as techniques so that an abstract Pascal machine can make variant records secure.

The official definition of ALGOL68 (which is quite a read) is [15]. A more accessible introduction can be found in [12]. The definition of ML can be found in [10], while an introductory treatment of type inference is to be found in [1].

There is a large literature on garbage-collection techniques. A detailed description of a mark and sweep algorithm can be found in many algorithm books, for example [6], while [16] is a good summary of classical techniques. For a book entirely devoted to the problem, that contains pseudocode and a more-or-less complete bibliography (up to the time of publication), [7] is suggested.

8.14 Exercises

1. Consider the declaration of the multi-dimensional array:

```
int A[10][10][10]
```

We know that an integer can be stored in 4 bytes. The array is stored in row order, with increasing memory addresses (that is, if an element is at address i, its successor is at $i + 4$, and so on). The element `A[0][0][0]` is stored at address 0. State the address at which element `A[2][2][5]` is stored.

2. Instead of the contiguous multidimensional array allocation that we discussed in Sect. 8.4.3, some languages allow (e.g., C), or adopt (Java), a different organisation, called *row-pointer*. Let us deal with the case of a 2-dimensional array. Instead of storing rows one after the other, every row is stored separately in some region of memory (for example on the heap). Corresponding to the name of the vector, a vector of pointers is allocated. Each of the pointers points to a proper row of the array. (i) Give the formula for accessing an arbitrary element `A[i][j]` of an array allocated using this scheme. (ii) Generalise this memory technique to arrays of more than 2 dimensions. (iii) Discuss the advantages and disadvantages of this organisation in the general case. (iv) Discuss the advantages and disadvantages of this technique for the storage of 2-dimensional arrays of characters (that is, arrays of strings).

3. Consider the following (Pascal) declarations:

```
type string = packed array [1..16] of char;
```

```
type string_pointer =  ^string;
type person = record
                 name : string;
                 case student: Boolean of
                    true:  (year: integer);
                    false: (socialsecno: string_pointer)
               end;
```

and assume that the variable `C` contains a pointer to the string `"LANGUAGES"`. Describe the memory representation for the `person` record after each of the following operations:

```
var pippo : person;
pippo.student := true;
pippo.year := 223344;
pippo.student := true;
pippo.socialsecno := C;
```

4. Show that the integer type can be defined as a recursive type starting with just the value `null`. Then write a function that checks if two integers defined in this way are equal.
5. Given the following type definitions in a programming language which uses structural type equivalence:

```
type T1 = struct{
             int a;
             bool b;
            };
type T2 = struct{
             int a;
             bool b;
            }
type T3 = struct{
             T2 u;
             T1 v;
            }
type T4 = struct{
             T1 u;
             T2 v;
            }
```

In the scope of the declarations `T3 a` and `T4 b`, it is claimed that the assignment `a = b` is permitted. Justify your answer.

6. Which type is assigned to each of the following functions using polymorphic type inference?

```
fun G(f,x){return f(f(x));}
fun H(t,x,y){if (t(x)) return x;
         else return y;}
fun K(x,y){return x;}
```

7. Using tombstones, it is necessary to invalidate a tombstone when its object is no longer meaningful. The matter is clear if the object is on the heap and is explicitly deallocated. It is less clear when the tombstone is associated with an address on the stack. In this case, indeed, it is necessary for the abstract machine to be able to determine all the tombstones that are potentially associated with an activation record. Design a possible organisation which makes this operation reasonably efficient (recall that tombstones are not allocated in activation records but in the cemetery).
8. Consider the following fragment in a pseudo-language with reference-based variables and which uses locks and keys (C is a class whose structure is of no importance):

```
C foo = new C();  // object OG1
C bar = new C();  // object OG2
C fie = foo;
bar = fie;
```

   Give possible values *after* the execution of the fragment for all the keys and all the locks involved.
9. Under the same assumptions as in Exercise 8, consider the following code fragment in which `free(p)` denotes the explicit deallocation of the object referred to by the pointer `p`:

```
class C { int n; C next;}
C foo = new C();  // object OG1
C bar = new C();  // object OG2
foo.next = bar;
bar.next = foo;
free(bar);
```

   For all the pointers in the fragment, give possible key values. For each object in the fragment, give possible lock values. In both cases, the values should be those *after* execution of the fragment terminates. After execution of the fragment, also execute the code `foo.n = 1; foo.next = 0;`. State what a possible result of this execution might be.
10. Consider the following fragment which is written in a pseudo-language with a reference-model for variables and a reference-counting garbage collector. If OGG is an arbitrary object in the heap, indicate by `OGG.cont` its (hidden) contents:

```
class C { int n; C next;}
C foo(){
   C p = new C();      // object OGG1
   p.next = new C();   // object OGG2
   C q = new C();      // object OGG3
   q.next = p.next;
   return p.next;
   }
C r = foo();
```

   State what are the values of the reference counters for the three objects after execution of line 6 and then of line 9.
11. Under the same assumptions as in Exercise 10, state what the values of the reference counters are for the two objects after the execution of the following fragment. Which of the two can be returned to the free list?

```
C foo = new C();  // object OG1
C bar = new C();  // object OG2
C fie = foo;
bar = fie;
```

12. Under the same assumptions as in Exercise 10, state what the reference-counter values for the three objects after execution of the following fragment are. Which of them can be returned to the free list?

```
class C { int n; C next;}
C foo = new C();  // object OG1
bar = new C();    // object OG2
foo.next = bar;
bar = new C();    // object OG3
foo = bar;
```

13. Using your favourite programming language, define the data structures required to represent a binary tree. Then write detailed code that performs a preorder traversal of a tree without using the system stack and instead using the pointer-reversal technique.

References

1. L. Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, 1987. citeseer.ist.psu.edu/cardelli88basic.html.
2. L. Cardelli. Type systems. In A. B. Tucker, editor, *The Computer Science and Engineering Handbook*. CRC Press, Boca Raton, 1997. citeseer.ist.psu.edu/cardelli97type.html.
3. L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985. citeseer.ist.psu.edu/cardelli85understanding.html.
4. J. C. Cleaveland. *An Introduction to Data Types*. Addison-Wesley, Reading, 1986.
5. C. N. Fisher and R. J. LeBlanc. The implementation of run-time diagnostics in Pascal. *IEEE Trans. Softw. Eng.*, 6(4):313–319, 1980.
6. E. Horowitz and S. Sahni. *Fundamentals of Data Structures in Pascal*. Freeman, New York, 1994.
7. R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, New York, 1996.
8. D. B. Lomet. Scheme for invalidating references to freed storage. *IBM Journal of Research and Development*, 19(1):26–35, 1975.
9. D. B. Lomet. Making pointers safe in system programming languages. *IEEE Trans. Softw. Eng.*, 11(1):87–96, 1985.
10. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML—Revised*. MIT Press, Cambridge, 1997.

Chapter 9
Data Abstraction

The physical machine manipulates data of only one type: bit strings. The types of a high-level language impose an organisation on this undivided universe, giving each value a sort of "wrapping". Each value is wrapped in an encapsulation (its type) which provides the operations that manipulate it. The type system for a language establishes how transparent this encapsulation is. In type-safe languages, the encapsulation is completely opaque, in the sense that it does not allow access to the representation (or better: every access can only take place using or by means of the encapsulation itself).

The last chapter presented in detail many of the predefined types and the principal mechanisms for defining new ones. The latter, however, are fairly limited: finite homogeneous aggregations (arrays) and heterogeneous ones (records), recursive types and pointers. The operations possible on these composite types are predefined by the language and the programmer is restricted to use them. It is clear that the programmer, using just the mechanisms we discussed in Chap. 8, cannot really define a *new* type, when it is understood, using our definition, as a collection of (homogeneous and effectively presented) values equipped with a set of operations. The user of a language can only make use of the existing capsules and only has highly limited ways to define new types: there are few *data abstraction* mechanisms.

In this chapter, we will present some of the main ways in which a language can provide more sophisticated mechanisms for defining abstractions over data. Among these, we will discuss the so-called abstract data type, which in different forms is provided by several languages. We then look at modules, a largely similar concept, but which mainly applies to programming in the large. These abstraction mechanisms also constitute an introduction to some themes that we will encounter again with object-oriented programming. The key concepts that we will investigate in this chapter are the separation between *interface* and *implementation* and the concepts associated with *information hiding*.

9.1 Abstract Data Types

The introduction of a new type using the mechanisms discussed in the last chapter

DOI 10.1007/978-1-84882-914-5_9, © Springer-Verlag London Limited 2010

```
type Int_Stack = struct{
                    int P[100];  // the stack proper
                    int top;     // first readable element
}
Int_Stack create_stack(){
   Int_Stack s = new Int_Stack();
   s.top = 0;
   return s;
}
Int_Stack push(Int_Stack s, int k){
   if (s.top == 100) error;
   s.P[s.top] = k;
   s.top = s.top + 1;
   return s;
}
int top(Int_Stack s){
   return s.P[s.top];
}
Int_Stack pop(Int_Stack s){
   if (s.top == 0) error;
   s.top = s.top - 1;
   return s;
}
bool empty(Int_Stack s){
   return (s.top == 0);
}
```

Fig. 9.1 Stack of integers

does not permit the user of a language to define types at the same level of abstraction as that enjoyed by the predefined types in these languages.

By way of an example, Fig. 9.1 shows one possible definition in our pseudo-language of the stack of integers data type. We assume a reference model for variables. When defining a type of this kind, it is probably intended that a stack of integers will be a data structure that can be manipulated by the operations of creation, insertion, access to the top element, removal of the top element. However, the language does not guarantee that these are the *only* ways in which a stack can be manipulated. Even if we adopt a strict type equivalence by name discipline (so that a stack is introduced by a declaration of the type `Int_Stack`) nothing prevents us from directly accessing its representation as an array:

```
int second_from_top()(Int_Stack c){
   return c.P[s.top - 1];
   }
```

From a general viewpoint, therefore, while languages provide predefined data abstractions (types) which hide implementations, the programmer cannot do this for themselves. To avoid this problem, some programming languages allow the definition of data abstractions which behave like predefined types as far as the (in)accessibility of representations is concerned. This mechanism is called an *abstract data type* (ADT). It is characterised by the following primary characteristics:

1. A name for the type.
2. An implementation (or representation) for the type.
3. A set of names denoting operations for manipulating the values of the type, together with their types.
4. For every operation, an implementation that uses the representation provided in point 2.
5. A security capsule which separates the name of the type and those of the operations from their implementations.

One possible notation for the stack of integers ADT in our pseudo-language could be that depicted in Fig. 9.2.

```
abstype Int_Stack{                                                 1
   type Int_Stack = struct{
                      int P[100];                                  3
                      int n;
                      int top;                                     5
   }
   signature                                                       7
      Int_Stack create_stack();
      Int_Stack push(Int_Stack s, int k);                          9
      int top(Int_Stack s);
      Int_Stack pop(Int_Stack s);                                  11
      bool empty(Int_Stack s);
   operations                                                      13
      Int_Stack create_stack(){
         Int_Stack s = new Int_Stack();                            15
         s.n = 0;
         s.top = 0;                                                17
         return s;
      }                                                            19
      Int_Stack push(Int_Stack s, int k){
         if (s.n == 100) error;                                    21
         s.n = s.n + 1;
         s.P[s.top] = k;                                           23
         s.top = s.top + 1;
         return s;                                                 25
      }
      int top(Int_Stack s){                                        27
         return s.P[s.top];
      }                                                            29
      Int_Stack pop(Int_Stack s){
         if (s.n == 0) error;                                      31
         s.n = s.n - 1;
         s.top = s.top - 1;                                        33
         return s;
      }                                                            35
      bool empty(Int_Stack s){
         return (s.n == 0);                                        37
      }
}                                                                  39
```

Fig. 9.2 ADT for stacks of integers

A definition of this kind must be interpreted as follows. The first line introduces the name of the abstract data type. Line 2 provides the representation (or *concrete type*) for the abstract type `Int_Stack`. Lines 7 to 12 (introduced by `signature`) define the names and types of the operations that can manipulate an `Int_Stack`. The remaining lines (introduced by `operations`) provide the implementation of the operations. The important point of this definition is that inside the declaration of type, `Int_Stack` is a synonym for its concrete representation (and therefore the operations manipulate a stack as a record containing an array and two integer fields), while outside (and therefore in the rest of the program), there is no relation between an `Int_Stack` and its concrete type. The only possible ways to manipulate an `Int_Stack` are provided by its operations. The function `second_from_top()`, which we defined above, is now impossible because type checking does not permit the application of a field selector to an `Int_Stack` (which is *not* a record outside of its definition.)

An ADT is an opaque capsule. On the outside surface, visible to anyone, we find the name of the new type and the names and types of the operations. Inside, invisible to the outside world, there are the implementations of the type and its operations. Access to the inside is always controlled by the capsule which guarantees the consistency of the information it encloses. This external surface of the capsule is called the *signature* or *interface* of the ADT. On its inside is its *implementation*.

Abstract data types (in languages that support them, for example ML, and CLU[1]) behave just like predefined types. It is possible to declare variables of an abstract type and to use one abstract type in the definition of another, as has been done, by way of example, in Fig. 9.3. The code in Fig. 9.3 implements (very inefficiently) a variable of type integer using a stack. Inside the implementation of `Int_Var`, the implementation of `Int_Stack` is invisible because it was completely encapsulated when it was defined.

9.2 Information Hiding

The division between interface and implementation is of great importance for software development methods because it allows the separation a component's use from its definition. We also saw this distinction when we looked at control abstraction. A function abstracts (that is, hides) the code constituting its body (the implementation), while it reveals its interface, which is composed of its name and of the number and types of its parameters (that is, its signature). Data abstraction generalises this somewhat primitive form of abstraction. Not only is how an operation is implemented hidden but so is the way in which the data is represented, so that the language (with its type system) can guarantee that the abstraction cannot be violated. This phenomenon is called *information hiding*. One of the more interesting

[1]In object-oriented languages, we will see that it is possible to obtain the same abstraction goal using a similar but more flexible method.

Constructors, transformers and observers

In the definition of an abstract data type, `T`, the operations are conceptually divided into three separate categories:

- *Constructors*. These are operations which construct a new value of type `T`, possibly using values of other known types.
- *Transformers* or *operators*. These are operations that compute values of type `T`, possibly using other values (of the same ADT or of other types). A fundamental property of a transformer `t` of type $S_1 \times \cdots \times S_k$ `->` `Y` is that, for every argument value, it must be the case that $t(s_1, \ldots, s_k)$ is a value constructable using only constructors.
- *Observers*. These are operations that compute a value of a known type that is different from `T`, using one or more values of type `T`.

An ADT without constructors is completely useless. There is no way to construct a value. In general, an ADT must have at least one operation in each of the above categories. It is not always easy to show that an operation is really a transformer (that is, that each of its values is in reality a value that can be obtained from a sequence of constructors).

In the integer stack example, `create_stack` and `push` are constructors, `pop` is a transformer and `top` and `empty` are observers.

```
abstype Int_Var{
   type Int_Var = Int_Stack;
   signature
      Int_Var create_var();
      int deref(Int_Var v);
      Int_Var assign(Int_Var v, int n);
   operations
      Int_Var create_var(){
         return push(create_stack(),0);
      }
      int deref(Int_Var v){
         return top(v);
      }
      Int_Var assign(Int_Var v, int n){
         return push(pop(v),n);
      }
}
```

Fig. 9.3 An ADT for an integer variable, implemented with a stack

consequences of information hiding is that, under certain conditions, it is possible to substitute the implementation of one ADT for that of another while keeping the interface the same. In the stack example, we could opt to represent the concrete type as a linked list (ignoring deallocation) allocated in the heap. This is de-

```
abstype Int_Stack{
   type Int_Stack = struct{
                       int info;
                       Int_stack next;
   }
   signature
      Int_Stack create_stack();
      Int_Stack push(Int_Stack s, int k);
      int top(Int_Stack s);
      Int_Stack pop(Int_Stack s);
      bool empty(Int_Stack s);
   operations
      Int_Stack create_stack(){
         return null;
      }
      Int_Stack push(Int_Stack s, int k){
         Int_Stack tmp = new Int_Stack(); // new element
         tmp.info = k;
         tmp.next = s;                    // chain on
         return tmp;
      }
      int top(Int_Stack s){
         return s.info;
      }
      Int_Stack pop(Int_Stack s){
         return s.next;
      }
      bool empty(Int_Stack s){
         return (s == null);
      }
}
```

Fig. 9.4 Another definition of the `Int_Stack` ADT

fined as in Fig. 9.4. Under certain assumptions, by substituting the first definition of `Int_Stack` by that in Fig. 9.4, there should be no *observable* effect on programs that use the abstract data type. These assumptions centre on what the clients of the interface expect from the operations. Let us say that the description of the semantics of the operations of an ADT is a *specification*, expressed not in terms of concrete types but general abstract relations. One possible specification for our ADT `Int_Stack` could be:

- `create_stack` creates an empty stack.
- `push` inserts an element into the stack.
- `top` returns the element at the top of the stack without modifying the stack itself. The stack must not be empty.
- `pop` removes the top elements from the stack. The stack must not be empty.
- `empty` is true if and only if the stack is empty.

Every client that uses `Int_Stack` making use of this specification *only*, will see no difference at all in the two definitions. This property has come to be called the *principle of representation independence*.

The specification of an abstract data type can be given in many different ways, ranging from natural language (which we have done) to semi-formal schemata, to completely formalised languages that can be manipulated by theorem provers. A specification is a kind of contract between the ADT and its client. The ADT guarantees that the implementation of the operations (which are unknown to the client) *matches* the specification; that is, all the properties stated in the specification are satisfied by the implementation. When this happens, it is also said that the implementation is *correct* with respect to the specification.

9.2.1 Representation Independence

We can state the representation independence property as follows:

> Two correct implementations of a single specification of an ADT are observationally indistinguishable by the clients of these types.

If a type enjoys representation independence, it is possible to replace its implementation by an equivalent (e.g., more efficient) one without causing any (new) errors in clients.

It should be clear that a considerable (and not at all obvious) part of representation independence consists of the guarantee that both implementations are correct with respect to the same specification. This can be hard to show, particularly when the specification is informal. There is, however, a weak version of the representation independence property that concerns only correctness with respect to the signature. In a type-safe language with abstract data types, the replacement of one ADT by another with the same signature (but different implementation) does not cause type errors. Under the assumptions we have made (of type safety and ADT), this property is a theorem which can be proved by type inference. Languages like ML and CLU enjoy this form of representation independence.

9.3 Modules

Abstract data types are mechanisms for programming in the small. They were designed to encapsulate *one* type and its associated operations. It is much more common, however, for an abstraction to be composed of a number of inter-related types (or data structures) of which it is desired to give clients a limited (that is, abstract) view (not all the operations are revealed and there is concealment of the implementation). The linguistic mechanisms which implement this type of encapsulation are called *modules* or *packages*. They belong to that part of a programming language dealing with programming in the large, that is with the implementation of complex systems using composition and assembly of simpler components. The module mechanism allows the static partitioning of a program into separate parts, each of which is equipped with data (types, variables, etc.) as well as with operations (functions, code, etc.). A module groups together a number of declarations (of data and/or

functions) as well as defining visibility rules for these declarations by means of which a form of encapsulation and information hiding is implemented.

Considering semantic principles, there is not much of a difference between modules and ADTs, mostly the ability to define more than one type at a time (according to the definitions we have given, an ADT is a particular case of a module). Pragmatically, on the other hand, the module mechanism affords greater flexibility, both in the ability to state how permeable its encapsulation is (indeed, it is possible to indicate on an individual basis which operations are visible or to choose the level of visibility), or in the possibility of defining generic (polymorphic) modules (recall Sect. 8.8). Finally, module constructs are often related to separate compilation mechanisms for modules themselves.[2]

Even given the enormous variety among existing languages, we can state the important linguistic characteristics of a module by discussing the example shown in Fig. 9.5. It is expressed, as usual, in an suitable pseudo-language. First of all, a module is divided, as is an ADT, into a public part (which is visible to all the module's clients) and a private part (which is invisible to the outside world). The private part of a module can contain declarations that do not appear at all in the public part (for example, the `bookkeep` function inside `Queue`). A module can mention some of its own data structures in the public part, so that anyone can use or modify them (the variable `c` in `Buffer`, for example). A client module can use the public part of another module by *importing* it (the `imports` clause). In our example, `Buffer` has an additional import clause in its private part.

We will not continue with this discussion, both because we would have to go into the details of the mechanisms in a specific language, and because we will return to many points when considering object-oriented programming. To conclude, let us merely observe that the module mechanism is often associated with some form of parametric polymorphism which requires link-time resolution. In our example, we could have defined a buffer of type `T`, rather than a buffer of integer, making the definition generic and then suitably instantiating it when it is used. Figure 9.6 shows the generic version of the buffer example. It can be seen how the buffer and the code are both generic. When the private part of `Buffer` imports `Queue`, it specifies that it must be instantiated to the same type (which is not specified) as `Buffer`.

9.4 Chapter Summary

This short chapter has presented a first introduction to data abstraction, which turns on the key concepts of *interface*, *implementation*, *encapsulation*, *data hiding*.

[2]Keep in mind that modules and separate compilation are independent aspects of a language. In Java, for example, it is not the module (package, in Java terminology) which is the unit of compilation, rather the class is.

```
module Buffer imports Counter{
   public
      type Buf;
      void insert(reference Buf f, int n);
      int get(Buf b);
      Count c; // how many times buffer has been used
   private imports Queue{
        type Buf = Queue;
      void insert(reference Buf b, int n){
         inqueue(b,n);
         inc(c);
      }
      int get(Buf b){
         return dequeue(b);
         inc(c);
      }
      init_counter(c);    // module initialisation part
}
module Counter{
   public
      type Count;
      void init_counter(reference Count c);
      int get(Count c);
      void inc(reference Count c);
   private
      type Count = int;
      void init_counter(reference Count c){
         c=0;
      }
      int get(Count c){
         return c;
      }
      void inc(reference Count c){
         c = c+1;
      }
}
module Queue{
   public
      type Queue;
      inqueue(reference Queue q, int n);
      int dequeue(reference Queue q);
      ...
   private
      void bookkeep(reference Queue q){
         ...
      }
      ...
}
```

Fig. 9.5 Modules

```
module Buffer<T> imports Counter{
   public
      type Buf;
      void insert(reference Buf f, <T> n);
      <T> get(Buf b);
      Count c; //  how many times buffer has been used
   private imports Queue<T>{
      type Buf = Queue;
      void insert(reference Buf b, <T> n){
         inqueue(b,n);
         inc(c);
         }
      <T> get(Buf b){
         return dequeue(b);
         inc(c);
         }
}
module Counter{
   public
      type Count;
      void init_counter(reference Count c);
      int get(Count c);
      void inc(reference Count c);
   private
      type Count = int;
      void init_counter(reference Count c){
         c=0;
         }
      int get(Count c){
         return c;
         }
      void inc(reference Count c){
         c = c+1;
         }
      init_counter(c);    // module initialisation
}
module Queue<S>{
   public
      type Queue;
      inqueue(reference Queue q, <S> n);
      <S> dequeue(reference Queue q);
      ...
   private
      void bookkeep(reference Queue q){
         ...
         }
      ...
}
```

Fig. 9.6 Generic modules

From a linguistic viewpoint, we have presented the following:

- *Abstract data type* mechanisms.
- Mechanisms to hide information and their consequences; that is, the *Principle of representation independence*.
- *Modules* which apply the concepts of encapsulation to programming in the large.

All of these concepts are treated in more depth in texts on software engineering. As far as this book is concerned, they are devices to help understanding object-oriented programming which is the subject of the next chapter.

9.5 Bibliographical Notes

The concept of module probably appears for the first time in the Simula language [1, 4], the first object-oriented language (see Chap. 13). The development of modules in programming languages is due to the work of Wirth [6], which includes the Modula and Oberon [7] projects.

The concept of information hiding made its first appearance in the literature in a classic paper by Parnas [5]. Abstract data types originate in the same context, as a mechanism guaranteeing abstraction that is different from modules. Among the languages that include ADTs, the most influential is certainly CLU [3] which is also the basis for the book [2].

9.6 Exercises

1. Consider the following definition of an ADT in our pseudo-language:

```
abstype LittleUse{
   type LittleUse = int;
   signature
      LittleUse prox(LittleUse x);
      int get(LittleUse x);
   operations
      LittleUse prox(LittleUse x){
         return x+1;
         }
      int get(LittleUse x){
         return x;
         }
}
```

   Why this type is useless?