Chapter 6

User-defined Types

Chapter 1 introduced the idea of categorizing program data into various types and the importance of a strong type system has been emphasized throughout this book. However, the available built-in types are somewhat limited, in that they do not always directly model the complex relationships inherent in "real-world" data. This chapter presents a mechanism by which a programmer can create new types. This allows the creation of data structures which are better able to model "real-world" relationships and it encourages the use of the type system to provide *built-in validation* of data.

6.1 The need for new types

A type may be defined with precision by listing the collection of values which belong to the type (that is, the values which exist in the type domain). For example, the type `bool` is precisely defined by listing the two values in its domain: `True` and `False`.[1] This section discusses the need for new types through several abstract examples, concentrating on the collection of values which define a type. This approach is utilized in Section 6.2, where Miranda's actual mechanism for creating new types will be presented.

New types

Miranda provides a small number of useful built-in types. However, the choice of built-in types is a matter for the language designer and it is likely that some other useful types have not been provided. For example, other programming languages do not have a built-in Boolean type. If the type `bool` did not already exist in Miranda, it could be created by specifying the permitted values as { *True or False* }.

[1] As explained in Chapter 1, we shall ignore the undefined, or error, value in the type domain.

If the type `num` did not already exist, an attempt could be made to create it in a similar way: { *1 or 2 or 3 or 4 or 5 or ...* }. However, this type presents some difficulty because of the very large number of alternative values (in practice limited by the computer architecture). The solution to the problem of a large number of values is to describe the values recursively, as will be seen later in this chapter (Section 6.4).

Special versions of types

Although the type `bool` is a built-in type for Miranda, it is possible that a programmer might wish to create several special versions of the type and make use of the fact that the type system will ensure that they are used consistently in the program and will not be mixed. For example, { *Windows_true or Windows_false* } could be used to indicate whether the program is connected to a display which supports a windowing environment, whilst { *Multi_user_true or Multi_user_false* } could be used to indicate whether the program is available to a single user or to many users.

This facility for specialization is not limited to Boolean values. For instance, dice-playing games require special meaning to be given to the numbers from one to six, so { *Dice_one or Dice_two or Dice_three or Dice_four or Dice_five or Dice_six* } could be used to indicate the full range of values for a new type called *dice*.

By using these new types, not only can a program be made more readable but there is the considerable advantage that the type system can be used to provide built-in validation (so that *Windows_true* and `True` can never be confused).

Structured types and constructors

In the pursuit of well-structured data, it is desirable to be able to specify new types in terms of previously defined types. In particular, it should be possible to build upon existing types (including previously specified user-defined types) in order to produce a new type which is able to represent the structured data which occurs in many applications.

A convenient mechanism to achieve this aim is to allow each value of a user-defined type possibly to be followed by a type name to indicate a further sub-level of values. This leads to a hierarchical specification of the values which are valid for a given type. For example, it is possible to define a single type which contains information about whether multiple simultaneous users are allowed, with each of the two values having a further underlying value to indicate whether a windowing system is being used: { *Multi_user bool or Single_user bool* }. In this example, the domain for the new type is fully defined by the values *Multi_user False*, *Multi_user True*, *Single_user False* and *Single_user True*. However, the underlying types need not have finite domains, as shown by the following example: { *Discrete num or*

Continuous [num] }.

In the above example, the constant values *Discrete* and *Continuous* provide a unique way to determine the structure of a particular value in the domain of this type; for this reason they are often referred to as *constructors*. Notice that there need not be any underlying type to be "constructed"; `True` and `False` are both simple data values and are also known as constructors. Furthermore, the different constructors for a type can have mixed structure—some may have a complex underlying type, whereas others may have no underlying type at all.

Mixed types

A programmer might wish to create a new type, whose values could be drawn from any one of a mixture of built-in types. For example, a company might describe a customer using a string (that is, type `[char]`) if the customer is an individual, by a `num` or `[num]` if the customer is an internal department or group of departments, or by a `([char],num)` or `([char],[num])` pair if the customer is a department or group of departments of an external company. In this example, the programmer may wish to encapsulate these options into a new type with the alternatives: { *Individual [char] or Department num or Group [num] or Ext_dept ([char],[num]) or Ext_group ([char], [num])* }. Examples of values of this new type are *Individual "Winston"*, *Department 45* and *Group [1,3,67]*; thus the domain for this new type contains all the values of the `[char]` domain *together with* all the values of the `num` domain *together with* all the values of the `[num]` domain and so on.

Towards type abstraction

A programmer might wish to specify which values of an existing type are legal for a particular task; for example a twenty-four hour *clock* type should only allow numbers in the range 0 to 23. If the number of alternative values is small then a new type can be generated such as { *Clock_zero or Clock_one or Clock_two or Clock_three or ... or Clock_twenty_three* }.

An alternative would be to create a new type which uses all the values of an underlying type (such as `num`). However, in this case it would be necessary to define a set of operations for the new type, so that values outside of the range 0 to 24 are considered illegal and so that inappropriate operations (such as multiplying together values which represent the hours of the day) are not allowed. Chapter 7 will present the *abstract type* mechanism which allows a programmer to package a new type, together with the operations which are appropriate for that type.

The next section will present the full Miranda syntax for creating new types, which is based on a general mechanism for describing and creating types by means of *defining the permitted values of the type*.

6.2 Algebraic types

In Miranda, a user-defined type is known as an *algebraic type* and is created by a general-purpose mechanism based on the idea of *constructors*, which were first discussed in Section 3.2.1. The fundamental principle is that each permitted value of an algebraic type is distinguished from permitted values of other types by means of a special tag, known as a constructor. This gives a *unique and unambiguous* representation for every value of an algebraic type, in exactly the same way that the built-in constructors `:` and [] give a unique and unambiguous representation of any list. A general template for an algebraic type definition is:

new_type_name ::= *Value1*
| *Value2*
...
| *ValueN*

A new type must have at least one value; alternatives[2] are denoted by the vertical bar `|` and each value may either be *nullary* or may be constructed from an underlying type. A nullary value is a constant value, such as `True` or `False`, and is also known as a *nullary constructor* (sometimes known as a "constant constructor" for self-evident reasons). A value therefore has one of two formats:

Nullary_constructor_name

or

Constructor_name underlying_type

The *underlying_type* may be simple or aggregate, built-in or another algebraic type previously defined in the program.

Algebraic types are characterized by two important features:

1. They are *not* type synonyms. A type synonym is merely a shorthand denotation for an already existing type, whereas an algebraic type is a totally new type; a type synonym may be mixed with its actual type, whereas algebraic types may not be mixed with other types. Thus, in the definition:

   ```
   positive ::= Positive num
   ```

   the new type name is `positive` and the constructor name is `Positive`. The value `(Positive 3)` may *not* be substituted for the value `3` because they are of different types.
2. The existing properties of any underlying types are *not* inherited; equality and inequality are the only operations which may be legal upon two instances of an algebraic type. For example, although the operator `&` is defined for the built-in type `bool`, it is not defined for the type `windows ::= Windows bool`

[2]Note that alignment of alternatives follows the offside rule detailed in Chapter 2.

and therefore the expression `(Windows False) & (Windows True)` is meaningless.
Equality and relational tests provide an arbitrary but reproducible ordering. Because the order is not defined, this behaviour should not be relied upon in programs. However, Miranda defines equality to be `True` if both the constructor names are equal (and any existing underlying values are also equal). Hence, the expression `Constructor1 < Constructor2` will return `True` if `Constructor1` comes before `Constructor2` in the original definition of the algebraic type, or if they are the same constructor and their underlying values return `True` when tested by `(<)`.

Further examples:

```
switch ::= On | Off

colour ::= Rgb (num,num,num)   || Red, Green, Blue
         | Hsl (num,num,num) || Hue, Saturation, Luminance

radius   ::= Radius num

sphere   ::= Sphere radius colour

customer ::= Individual [char]
           | Department num
           | Group [num]
           | Company ([char],num)
```

The rest of this section discusses the various kinds of algebraic types that can be defined, including algebraic types with just one constructor and one underlying type, algebraic types with many constructors with different underlying types, and algebraic types which do not have an underlying type.

6.2.1 Simple algebraic type definition and usage

Algebraic type definition

A simple example of algebraic type definition is:

```
coords ::= Coords (num,num,num)
```

This definition serves two related purposes:

1. To create new algebraic type named `coords`.
2. To create a new prefix *constructor* `Coords`

This new constructor takes a number triple and converts it to the new algebraic type; as such, a constructor might be considered as a special form of function without a function body, as can be seen from Miranda's responses:

```
Miranda Coords
<function>

Miranda Coords ::
(num,num,num) -> coords
```

However, there are certain differences between constructors and functions. The differences between functions and constructors are explored in more depth in Section 6.3.

Algebraic type naming

Algebraic type names must be legal identifiers and conform to the rules for Miranda identifiers described in Chapter 1. Constructors must also be legal Miranda identifiers: *except their initial character must be an Upper case character*. There are no further restrictions and constructor names may look just like any other identifier.

For example, legal Miranda identifiers cannot start with a digit. Thus, in particular, it is not possible to model the integers with the definition:

```
wrong_dice ::= 1 | 2 | 3 | 4 | 5 | 6
```

It is equally wrong to use characters or strings as constructors, since a character or string is not a legal Miranda identifier:

```
wrong_dice ::= "One" | "Two" | "Three"
               | "Four" | "Five" | "Six"
```

A legal definition would be:

```
legal_dice ::= One | Two | Three
               | Four | Five | Six
```

As with value identifiers and function identifiers, it is not possible to reuse a constructor name within a program.

Algebraic type instantiation

New instances of `coords` may be created by supplying the `Coords` constructor with its expected argument:

```
point_origin = Coords (0.0, 0.0, 0.0)
point_max = Coords (1000.0, 1000.0, 1000.0)
```

The `Coords` constructor is used in the same manner as the built-in list constructor `:`. This is illustrated by comparing its use with that of the prefix version of the latter:

```
Miranda (:) 1 [2,3]
[1,2,3]

Miranda Coords (3.0, 4.0, 5.0)
Coords (3.0,4.0,5.0)
```

Algebraic type usage

Using algebraic types in functions is just as easy as using existing types and constructors. If there is a requirement to write a function to find the midpoint of two `coords` then they may be deconstructed to their underlying type using pattern matching:

```
midpoint ::  coords -> coords -> coords
midpoint (Coords (x1,y1,z1)) (Coords (x2,y2,z2))
     = Coords ((x1 + x2)/2, (y1 + y2)/2, (z1 + z2)/2)
```

In the application of the function `midpoint` to the two parameters `point_origin` and `point_max`:

```
Miranda midpoint point_origin point_max
Coords (500.0,500.0,500.0)
```

the formal parameters `Coords (x1,y1,z1)` and `Coords (x2,y2,z2)` will be substituted with the actual values: `Coords (0.0, 0.0, 0.0) and Coords (1000.0, 1000.0, 1000.0)`, respectively, and so will be successfully matched.

It can be seen that constructors are used in function patterns in order to *deconstruct* an algebraic type and that they are used in function bodies to *construct* a value of an algebraic type. Thus, for deconstruction purposes there is no practical difference between the extraction of the head of a list `front` from the constructed list `(front : rest)` and the extraction of the numbers `x1`, `y1` and `z1` from the constructed `Coords (x1,y1,z1)`. Similarly, for construction purposes there is no practical difference between the creation of a new list `(front : rest)` from the item `front` and the list `rest` and the creation of a new `coords` instance by means of the application `Coords (x1,y1,z1)`.

Note that, just as with patterns involving the list constructor `:`, it is necessary to bracket the `Coords` constructor with its argument otherwise a syntax error will occur. This is because a constructor pattern must always be complete (see Section 6.3.1).

Exercise 6.1
Write a function to calculate the distance between a pair of `coords`.

Algebraic types are strongly typed

It must be emphasized that using the keyword `::=` creates a *new* type which conforms to the Miranda strong typing philosophy. This has the considerable advantage that the system will perform data validation and automatically catch any accidental attempts to use an algebraic type in an illegal manner. In the above example, it is assumed that the programmer wishes to express the relationship between three numbers as a new type and does *not* wish to mix a number triple with a value drawn from this type. Hence, a value that has been defined as a number triple cannot be legally tested against an instance of a `coords` for equality:

```
Miranda point_origin = (0.0, 0.0, 0.0)

type error in expression
cannot unify coords with (num,num,num)
```

In a similar manner, any function defined over a tuple of type `(num,num,num)` (even if given a name using the `==` type synonym mechanism) cannot be applied to `coords` arguments.

6.2.2 Algebraic types with multiple constructors

An algebraic type may also have more than one constructor. This is shown in the following two examples, the first of which shows an algebraic type with more than one constructor over the same underlying type; the second shows the benefit of having constructors with different underlying types.

Multiple constructors over the same type

A new type `fluid` is now defined to express the fact that fluid measurements are different in different countries. The subsequent definition of the function `addFluids` is designed to eliminate the possibility of a programmer attempting to mix operations on fluid measures of differing kinds:

```
fluid ::= USgallons num
          | UKgallons num
          | Litres num

addFluids ::  fluid -> fluid -> fluid

addFluids (USgallons x) (USgallons y)
   = USgallons (x + y)
addFluids (UKgallons x) (UKgallons y)
   = UKgallons (x + y)
addFluids (Litres x) (Litres y)
   = Litres (x + y)
addFluids x y
   = error "addFluids:  illegal constructor"
```

It is now guaranteed that inappropriate operations such as adding amounts of different measurements or attempting to multiply two fluid measurements are not performed accidentally. Thus, the following applications will fail:

```
Miranda addFluids (USgallons 3.0) (Litres 54.0)
program error: addFluids: illegal constructor

Miranda (USgallons 3.0) * (USgallons 54.0)
type error in expression
cannot unify fluid with num

Miranda (USgallons 3.0) * (Litres 54.0)
type error in expression
cannot unify fluid with num
```

The last two applications fail because the standard arithmetic and relational operators are not overloaded for algebraic types. Once again, programmers are obliged to think carefully about their intentions and are helped to avoid mistakes by the type checker.

Multiple constructors over different types

It is also possible to have an algebraic type with different underlying types, as is now demonstrated with the following declaration, where an identification code can be either a number or a string:

```
idcode ::= Ncode num | Scode [char]
```

It is now necessary to define a new function to allow comparison between the two types of `idcode`s:

```
idless ::  idcode -> idcode -> bool

idless (Ncode x) (Ncode y) = x < y
idless (Scode x) (Scode y) = x < y
idless (Ncode x) (Scode y)
    = (int_to_string x) < y
idless (Scode x) (Ncode y)
    = x < (int_to_string y)
    || where int_to_string is as defined in Section 2.10
```

This new comparison function can now be used as any other function, for instance to construct a new sorting function, shown in Section 4.4.1:

```
idsort ::  [idcode] -> [idcode]
idsort = foldr (insert idless) []
```

Of course, it is still necessary to state explicitly the intended constructor for each new `idcode` and Miranda will always respond by echoing the constructor as well as the actual values:

```
Miranda idsort [Ncode 30, Scode "12", Ncode 1, Scode "10"]
[Ncode 1, Scode "10", Scode "12", Ncode 30]
```

6.2.3 Underlying types for algebraic types

There is no restriction on the underlying types for constructors; they can be simple types, aggregate types, polymorphic types, functions or previously user-defined types. This section shows some of these possibilities.

Polymorphic algebraic types

The algebraic type facility parallels the == facility in that it is also legal (and often very useful) to have polymorphic algebraic types. For example:

```
samepair * ::= SamePair (*,*)

pair_to_list ::  samepair * -> [*]
pair_to_list (SamePair (a,b)) = [a,b]
```

As with type synonyms, it is necessary to follow the algebraic type name with a declaration of the names of the polytypes involved in the right-hand side of the definition. If there are many different polymorphic types then the algebraic type name must be followed by all of the relevant polytypes:

```
mixedpair * ** ::= MixedPair (*,**)

mixedfst ::  mixedpair * ** -> *
mixedfst (MixedPair (a,b)) = a
```

New algebraic types from old

So far, all the examples of constructors have had built-in underlying types; however, it is often useful to build upon user-defined algebraic types to create more complex types that better represent the real-world data. The following example uses the algebraic type `coords` to create a new, curried, algebraic type `line` which represents a straight line in three-dimensional space:

```
line ::= Line (coords) (coords)
```

The following example using `line` shows both:

1. A name definition using the constructor `coords` inside the constructor `Line`.
2. A function definition `line_midpoint`, where the function pattern only needs to deconstruct the outer layer and therefore only uses the constructor `Line`:

```
line ::= Line coords coords

aline = Line (Coords(0,0,0)) (Coords (10.0,10.0,10.0))

midpoint ::  coords -> coords -> coords
midpoint (Coords (x1,y1,z1)) (Coords (x2,y2,z2))
    = Coords ((x1 + x2)/2, (y1 + y2)/2, (z1 + z2)/2)

line_midpoint ::  line -> coords
line_midpoint (Line x y) = midpoint x y
```

Functional algebraic types

Just as functions have been considered as values that may appear as the components of lists or tuples, it is possible for them to provide the underlying type for algebraic types, which is often useful when trying to model a dynamic relationship between objects. This is demonstrated in the following example, where a new type is created to hold both a list of `component`s, together with a function which operates on their price to cater for accounting details such as calculating profit margins:

```
component == (num,[char],num)
complist == [component]
|| the type synonym component represents a list of
|| key, description and price

|| sample component list
net_stock_list
     = [(1,"yoghurt",0.84), (2,"peas",1.3),
        (3,"icecream", 2.5)]

stock ::= Stock complist (complist -> complist)
|| a stock item represents a component list
|| together with a function to change the value
|| of each item in the list
```

New instances of `stock` can be created as follows:

```
gross_stock_list = Stock net_stock_list addTAX
```

where `addTAX` is defined, for example, in terms of the general-purpose function `adjust`:

```
adjust ::  num -> complist -> complist
adjust factor cl = [(key,description,newprice)
                    | (key,description,price) <- cl
                    ; newprice <- [price * factor]]

addTAX = adjust 1.175
```

The following function will now generate the information details of a particular `stock` instance:

```
stockdetails ::  stock -> complist
stockdetails (Stock slist acc_fn) = acc_fn slist
```

The advantage of this approach is that each instance of a `stock` can have a different accounting function, as long as it is of the correct type. Thus, it is possible to associate different taxation ratings or retail prices to a particular list of components. For example:

```
taxrate = stockdetails gross_stock_list
```

6.2.4 Enumerated algebraic types

The keyword `::=` can also be used to create *enumerated* (or extensional) types, which provide a set of names representing the full range of values for the type.[3] This

[3]This is very similar to enumeration in imperative programming languages.

facility has already been seen with the built-in names `True` and `False` which are the only values for the `bool` type. In fact, they are instances of *nullary constructors*; that is, constructors which have no parameters. By contrast, constructors which take a single or tuple argument are known as *unary constructors*. Similarly, Miranda allows constructors to have a higher *arity* with two or more curried arguments.

Enumerated algebraic type definition

The following example introduces an algebraic type representing the possible states of a set of traffic lights. In the UK a set of traffic lights has three colours (red, amber and green) and cycles between four states; the two primary states *green* (go) and *red* (stop), plus two intermediate states *amber* and *(red + amber)*.[4] The sequence of states is *green*, *amber*, *red*, *(red + amber)*.

```
traffic_light ::= Green | Amber
                  | Red | Red_amber
```

`Green`, `Amber`, `Red` and `Red_amber` are actually constructors for `traffic_light`; though in this new sort of construction the constructors construct nothing but themselves!

Enumerated algebraic type usage

The following session shows that enumerated algebraic types may be used in the same manner as any other type:

```
next_state ::  traffic_light -> traffic_light
next_state Green = Amber
next_state Amber = Red
next_state Red = Red_amber
next_state Red_amber = Green
```

```
Miranda map next_state [Green, Amber, Red, Red_amber]
[Amber,Red,Red_amber,Green]
```

It must also be noted that pattern matching on enumerated algebraic types follows the same rules as for all other types, in that every possible enumeration should be matched. For example, the following definition of `traffic_light` does not match the constructor `Red`:

[4]The combination of two colours for one of the intermediate states makes it possible to predict the next state in the sequence without the expense of adding a fourth coloured light. Although four states could be represented by only two coloured lights, there would be an ambiguity between one of the states and a power failure!

```
prior_state ::  traffic_light -> traffic_light
prior_state Green = Red_amber
prior_state Amber = Green
prior_state Red_amber = Red
```

This will compile successfully but generate a run-time error if applied to `Red`:

```
program error: missing case in definition of prior_state
```

Exercise 6.2

Given the algebraic type

```
action ::= Stop | No_change | Start
         | Slow_down | Prepare_to_start
```

write a function to take the appropriate action at each possible change in state for `traffic_light`.

Exercise 6.3

A Bochvar three-state logic has constants to indicate whether an expression is true, false or meaningless. Provide an algebraic type definition for this logic together with functions to perform the equivalent three-state versions of `&`, `\/` and logical *implication*. Note that if any part of an expression is meaningless then the entire expression should be considered meaningless.

Grep revisited

Enumeration can be used to improve the *grep* program by representing the `"ONCE"` and `"ZERO_MORE"` *match types* as follows:

```
mtype ::= ZERO_MORE | ONCE
```

This is a more elegant solution than using strings to represent the match types. It is also safer and guarantees consistency across functions because Miranda will only allow pattern matching with the two constructors. Otherwise, using the approach shown in Chapter 3, it would be possible accidentally to enter a meaningless string, such as `"ZERO_MORE"` (where the digit `0` is mistakenly used instead of the character `O`) in one of the function patterns for `startswith`. This is a legal string—but it will never be matched. Enumeration ensures that only legal options are considered.

6.3 Constructors and functions

This section summarizes how constructors differ from functions in their use in function patterns and how they are similar for other purposes. Finally, there is

a discussion of the consequences of having too many constructors in an algebraic type.

6.3.1 Pattern matching with constructors

There are three kinds of object that can appear in a valid pattern:

1. Constants, such as numbers and strings.
2. Constructors, either nullary or of any arity. However, a constructor pattern *must be complete*; that is, a non-nullary constructor pattern must contain a valid pattern for its parameter(s).
3. Formal parameter names, which cannot be constants or constructors

It must be emphasized that although the application of a constructor to its argument may appear as a pattern, the application of a function *is not* a legal pattern.

The above constraints mean that the following two definitions are incorrect; the first because `Measure1` is not defined as a constructor; the second because `Litres` does not have its parameter:

```
wrong_litre_convert (Measure1 x)
       = Litres (x * 3.7852), if Measure1 = USgallons
         ...., otherwise

wrong_general_convert (USgallons x, Litres)
       = Litres (x * 3.78532)
         ....
```

6.3.2 Constructors as functions

Constructors are similar to functions in that:

1. They translate values from one type to another.
2. Equality is only defined upon an algebraic type value. This means that two nullary constructors can be compared because they each represent a legal value of the type, but two non-nullary constructors cannot be compared because they do not represent any value until they are given the value of their underlying type. Thus `True = Meaningless` is a legal comparison, but `USgallons = UKgallons` is illegal. Comparisons of the form `USgallons x = UKgallons y` are legitimate, but will only return `True` if both constructors are the same and both underlying values are the same.
3. The constructor name can be composed or passed as a parameter to other functions, for instance:

```
Miranda map Litres [1.0,2.0,3.0]
[Litres 1.0, Litres 2.0, Litres 3.0]
```

If a constructor takes two or more underlying parameters then it may be partially applied in just the same way as a curried function may be partially applied. A partially applied constructor has function type, emphasizing the role of the constructor in converting from one type to another (just as a function translates values of one type to values of another type):

```
Miranda map Sphere [Radius 1, Radius 2, Radius 3]
[<function>,<function>,<function>]

Miranda hd (map Sphere [Radius 1, Radius 2]) (Rgb (3,2,3))
Sphere (Radius 1) (Rgb (3,2,3))
```

Constructors differ from functions in that they have a sense of order. Though intrinsically meaningless, it is possible to evaluate expressions such as:

```
Green < Red
```

The ordering is left to right from the point of definition. However, because it is not sensible to treat constructor names as representing some underlying ordinal type, it is *not* recommended to rely on this language feature.

6.3.3 The dangers of too many constructors

There is sometimes a temptation when programming with algebraic types to define a new type which represents too many things; that is, it has too many constructors. Consider the problem of providing a mechanism for the `fluid` algebraic types to enable values of any one of the three constructors to be converted to values of any one of the other constructors. The sledgehammer approach is to define a function for each conversion, giving rise to six functions: `UStoUK`, `UStoLitre`, `UKtoUS`, etc. The more elegant approach is to provide one general-purpose function; now the problem is how best to parameterize the conversion function to indicate the target constructor. One tempting approach has already been discounted—that of `wrong_general_convert`, shown in Section 6.3.1:

```
wrong_general_convert (USgallons x, Litres)
        = Litres (x * 3.78532)
            ...
```

The approach was correct in attempting to represent the target as a constructor name, but was syntactically illegal because constructor patterns must be complete. A variation on this approach is to represent the target constructor as an enumeration, as is now shown:

```
fluid    ::=  USgallons num
              | UKgallons num
              | Litres num
              | Fluid_USGALLONS
              | Fluid_UKGALLONS
              | Fluid_LITRES

convert  ::   fluid -> fluid -> fluid
convert       (USgallons x) Fluid_UKGALLONS
         =    UKgallons (x * 0.8327)
convert       (USgallons x) Fluid_LITRES
         =    Litres (x * 3.78532)
              ...
```

This solves the problem, but is rather unwieldy. What has happened is that the pattern matching requirements having been extended from three constructors to six for every function. Yet for this function not every permutation is necessary, and for many other functions the enumerations are not at all necessary. In brief, `fluid` has been "semantically overloaded".[5]

In the above example, it is probably more natural to have types that are less tightly linked, with some appropriate comment:

```
fluid       ::=  Fluid (fluid_name, num)
                 || requires the definition of fluid_name

fluid_name  ::=  Fluid_USGALLONS
                 | Fluid_UKGALLONS
                 | Fluid_LITRES
```

6.4 Recursive algebraic types

This section extends the principle of creating algebraic types that are closer models of the real world, to show how recursive algebraic types may be defined. Recursive types provide a mechanism to define new types with very many (potentially infinitely many) values. The first example shows how the built-in aggregate `list` type could be implemented; the second example shows the `tree` type, which is a more complex data structure.

6.4.1 Simple recursive algebraic types

The built-in list type has already been semi-formally specified in Section 3.1 as:

[5]Having too many constructors often interacts badly with an inductive style of program development, because of the cumbersome number of base cases to consider.

1. *empty*
2. *or an element of a given type together with a list of that given type.*

This type is defined recursively and there is no restriction on the number of items in a list (other than the amount of memory available in the computer). The specification meets the structural induction requirements of having a terminating case (the empty list) and a general case (the non-empty list). The built-in constructor for the empty list is `[]` and the built-in constructor for the non-empty list is `:` which, considered as a prefix operator, takes an item and a list, and constructs a new list from it.

This built-in recursive type may be denoted using the general mechanism described for algebraic type definition; all that is required is to use the name of the new type being defined as the underlying type for one of the constructors. Thus, in order to provide a user-defined type called `list` which mimics the built-in polymorphic `[*]` type, all that is necessary is to translate the above specification directly into the following Miranda definition:

```
list * ::= Nil | Cons * (list *)
```

Instances of lists constructed in this manner are displayed,[6] with their construction made explicit:

```
alist = Nil
blist = Cons 1 alist
clist = Cons 2 blist
```

```
Miranda blist ::
list num

Miranda clist
Cons 2 (Cons 1 Nil)
```

6.4.2 Tree recursive algebraic types

A more general data structure that is not a Miranda built-in type is the *tree*. Of the numerous variations on the tree concept, the following informal specification introduces one of the definitions of a *binary tree* (Standish, 1980).

[6]Note that it is not possible to simulate Miranda's alternative square bracket notation. However, for all other purposes Miranda lists and user-defined `list`s are exactly the same.

A binary tree is either:

1. *empty*
2. *or a node which contains a value, together with a left and a right binary subtree.*

This definition differs from the list, which is essentially a linear structure where each element may follow the next in only one way. With a binary tree, there are two branches at each node and therefore a choice has to be made at each node which branch to follow. The consequence is that tree creation, traversal and manipulation are more complex than their equivalent list operations.

The main advantage of the binary tree structure is for searching; the average number of inspections to extract a given element from a linear list is half the length of the list. However, the average number of inspections to find a member of a averagely balanced sorted binary tree is significantly less,[7] as can be seen from Figure 6.1. The worst case involves four comparisons (that is, half the number of items in the tree) and the average number of comparisons is three.

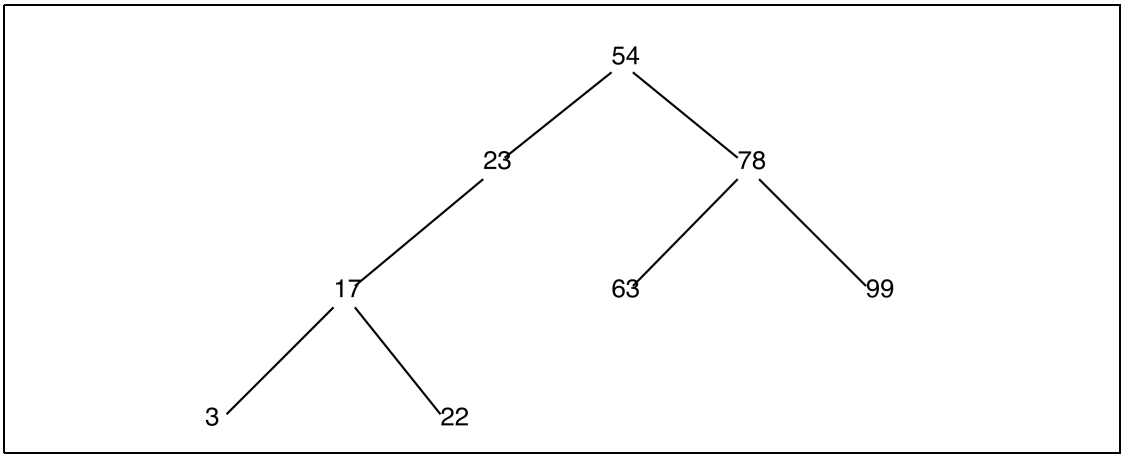

Figure 6.1 A sample tree.

Tree definition

The following translates the informal specification of a tree directly into a Miranda definition of a polymorphic tree, the ordering of which has not yet been determined:

```
tree * ::= Tnil | Tree (tree *, *, tree *)
```

[7]For a fully balanced tree (that is, where the items are equally divided between subtrees of the same length) it is actually $\log_2 N$ where N is the number of items in the tree.

There are a number of possibilities as to how this kind of tree can be organized; the most common organization is a sorted tree. The tree presented in Figure 6.1 meets the recursive specification that all the elements of a node's left subtree have a value that is less than the node value itself. Similarly all the elements of a node's right subtree must have a value that is not less than the node's value. The base case of such a tree is the empty tree `Tnil` which is defined to be sorted.

Growing a tree

The process of making a new (sorted) tree is very similar to making a sorted list using *insertion sort*, first described in Section 3.7.3. There, starting from an empty (sorted) list, elements are inserted one at a time to create an increasingly larger sorted list. Hence, to create a sorted tree, it is first necessary to create an empty (sorted) tree and then provide a function that takes any already sorted tree together with a new item, and returns a new sorted tree.[8]

The only real difference between tree insertion sort and list insertion sort is that the structure of a tree does not require that the existing tree is reordered when a new element is added. All that is necessary is to traverse the tree (according to its ordering specification) until an appropriate empty subtree is encountered and then add the new element. Naturally, for a polymorphic tree, it is also necessary to provide an ordering function as a parameter (as with `isort`, in Section 4.4.1).

The following two definitions show how an empty tree may be defined and how an item may be inserted into an already sorted tree. Note that the `insertleaf` function is drawn directly from the definition of the tree. It has the terminating condition of an empty tree and the choice at each non-empty tree as to whether to inspect the left or right subtree. To make this choice requires that tree be deconstructed to obtain the node value. Afterwards a new tree must be reconstructed using the item, the node and the rest of the tree.

```
ordering * == (* -> * -> bool)

insertleaf ::  ordering * -> tree * -> * -> tree *

insertleaf order Tnil item
    = Tree (Tnil, item, Tnil)
insertleaf order (Tree (ltree, node, rtree)) item
    = Tree (put ltree item, node, rtree),
        if order item node
    = Tree (ltree, node, put rtree item), otherwise
      where put = insertleaf order
```

[8]It should, of course, be stressed that *tree insertion* is a figurative term; the original tree is not actually altered, but a fresh copy is generated for each additional element.

From list to tree

Just as with the function `isort` in Chapter 4, it is possible to make use of a higher order function to grow a tree from a list without explicit recursion:

```
list_to_tree ::  ordering * -> [*] -> tree *
list_to_tree order itemlist
       = foldl (insertleaf order) Tnil itemlist
```

Or, more concisely:

```
list_to_tree ::  ordering * -> [*] -> tree *
list_to_tree order
       = foldl (insertleaf order) Tnil
```

From tree to list

The complementary function `tree_to_list` also follows directly from the `tree` definition. In effect, the constructor `Tree` has been replaced by the function `tree_to_list` and the tuple notation has been replaced by the append operator `++`:

```
tree_to_list ::  tree * -> [*]

tree_to_list Tnil = []
tree_to_list (Tree (ltree, node, rtree))
   = tree_to_list ltree ++ [node] ++ tree_to_list rtree
```

Because the tree's branching nature has been eliminated, this function is often known as `flatten`.

Exercise 6.4

Explain why it is not sensible to attempt to mirror the tree data structure using nested lists.

Exercise 6.5

A number of useful tree manipulation functions follow naturally from the specification of a binary tree. Write functions to parallel the list manipulation functions `map` and `#` (in terms of the number of nodes in the tree).

Exercise 6.6

What would have been the consequence of writing the function `list_to_tree` as:

```
list_to_tree order
    =  reduce (insertleaf order) Tnil
```

Exercise 6.7

Write a function to remove an element from a sorted tree and return a tree that is still sorted.

6.5 Placeholder types

This section introduces *placeholder types*, which are used to allow programmers to defer decisions concerning the type a function or set of functions should have. This may be useful, in that, at different stages in program development, the programmer deals with different problems or different levels of abstraction. Thus, at one stage, the programmer may be concerned with process design rather than data structure design.

For example, early in the design process, it may be decided that a function needs to process data of a given type, perhaps returning a count of the number of items in a aggregate type. At this level of abstraction it is more important that the type is aggregate than its actual representation, which could be a string, a user-defined set, ordered tree or whatever. The choice of aggregate type can better be made after more detailed consideration of the rest of the program.

As a temporary measure, the designer *could* select any arbitrary type that does not give an error, but then they will have to *remember to change* the type if the detailed specification requires it. Alternatively, the designer could choose a polymorphic representation, but this may also prove to be inappropriate or to be too general for the eventual specification. In this case, it is better to defer the decision by *declaring* a new type, using a *placeholder* and to *define* the type later (either by reusing an existing type or using constructors to create an entirely new type). Placeholder types are similar to type synonyms, except that :

1. The symbol **::** is used rather than ==.
2. The actual type representation has not yet been decided and instead the word **type** is used.

The above situation can now be represented using placeholders:

```
items ::  type

number_of_items ::  items -> num

main_function::  items -> num
main_function x = (number_of_items x) + 5
```

This code can now be processed by Miranda to check for type consistency. If all is well, Miranda will report that functions such as `number_of_items` are `SPECIFIED BUT NOT DEFINED` but there will be no actual type errors.

Notice that, unlike a type synonym, Miranda treats `items` as a new type:

```
Miranda number_of_items ::
items->num
```

At a later stage in the program development, the actual type will be determined and the code completed:

```
items == [char]

number_of_items ::  items -> num
number_of_items x = # x

main_function ::  items -> num
main_function x = (number_of_items x) + 5
```

Miranda has now been given a type synonym; hence the actual type of the function which accesses the data becomes clear:

```
Miranda number_of_items ::
[char]->num
```

Placeholder types are not particularly useful for small programs, but can be useful to check the type consistency within larger programs—without the need to make a full implementation decision.

6.6 Summary

The strong type system, as presented in Chapter 1, provides a means for enforcing correct usage of Miranda's base, aggregate and function types. The strengths of such a type system are that it promotes good programming style and detects many errors early in the software design cycle. The weakness of such a simple type system is that it imposes strict limitations on what may be expressed.

The first important step towards providing more expressive power was the ability to define functions with polymorphic types, discussed in Chapter 2. In this chapter, the type system was further expanded, using algebraic types, so that the programmer is no longer restricted to the basic Miranda types, but may build new types from the old ones and may expect the type system to apply the same rigour to the new types as to the old.

Finally, this chapter introduced placeholder types, which let the programmer defer the choice of a function's type whilst the program is still in the design stage.