Chapter 7

Abstract Types

This chapter concludes the presentation of Miranda's type system by introducing the *abstract type* mechanism, which allows the definition of a new type, together with the type expressions of a set of "interface" or "primitive" operations which permit access to a data item of the type. The definitions of these primitives are hidden, providing an abstract view of both code and data; this leads to a better appreciation of a program's overall meaning and structure.

The use of type expressions as a design tool and documentary aid has already been encouraged in previous chapters. The **abstype** construct (also known as an "abstract data type" or "ADT"[1]) defines the type expressions for a set of interface functions without specifying the function definitions; the benefits include the separation of specifications from implementation decisions and the ability to provide programmers with the same view of different code implementations. In summary, Miranda's **abstype** mechanism provides the following two new programming benefits:

1. Encapsulation.
2. Abstraction.

Encapsulation

There are considerable benefits if each programmer or programming team can be confident that their coding efforts are not going to interfere inadvertently with other programmers and that other programmers are not going to inadvertently affect their implementation details. This can be achieved by packaging a new type together with the functions which may manipulate values of that type.

Furthermore if all the code required to manipulate the new type is collected in

[1]However, there is a subtle difference between Miranda's **abstype** mechanism and traditional ADTs—see Section 7.1.2.

one place it is easier to test, modify or maintain and to make accessible to other programmers.

Abstraction

The set of type expressions in an **abstype** is called the *signature* of the **abstype**. The purpose of the interface signature in an **abstype** is to provide a high-level specification of the types of functions which operate on a new type. By using an interface signature, it is possible to "abstract" away from the decision of what underlying type the actual implementation will use. This has the bonus of allowing the implementor considerable freedom to alter the underlying representation at a later time without needing to change a single line of code in the rest of the program (as long as the type and the purpose of each interface function remain the same and the program is recompiled).[2]

For example, if one programmer develops a number of functions that manipulate a data item (perhaps the functions `validate_date` and `days_between_dates` which manipulate `dates`) then it is wasteful for other programmers to develop the same functions. Furthermore, as long as they know the type signatures of the functions and what actions the functions perform then *it is unnecessary for other programmers to understand how the functions are implemented.*

If these operations are linked with the creation of a new type then the advantages of built-in type validation are also available.

Syntax

The syntax for using an **abstype** is given by the following general template:

abstype *type_declaration*
with *function_type_signature1*
. . .
function_type_signatureN

type_declaration_instantiation

function_definition1
. . .
function_definitionN

[2]This can be particularly useful if the implementation proves too slow or the input data profile changes.

The functions declared within the **abstype** body provide the public interface to the new type; that is, values of this type can only be manipulated via the public interface functions.

Note that the **abstype** itself merely specifies the *name* of a new type and the *types* of the interface functions—the definitions of the type and the interface functions are given immediately afterwards, just as they would be if they were not encapsulated as an **abstype**. This enforces the intention that type information (in the signature) should be separated from implementation information (in the subsequent function definitions).

7.1 Simple abstract type definition and usage

The following example shows how the natural numbers (that is, the non-negative integers) can be considered as an **abstype** package, consisting of a numeric type together with frequently required associated functions. This package would typically be used to manipulate anything for which a negative representation would be meaningless, "prices" and "quantities" being two frequent cases.

It should be noted that the number of functions in the package is limited to mimic those provided by the built-in arithmetic and relational operators; as with the built-in types, there is no need to provide functions for every possible user-specified operation involving natural numbers.[3] The functions that are provided are often known as *primitives* because they can be considered the raw building blocks for the type.

Abstract type definition

```
|| abstype nat specification

abstype nat
with
    makeNat    ::  num -> nat
    natEqual   ::  nat -> nat -> bool
    natLess    ::  nat -> nat -> bool
    || similarly for the other comparison operators

    natMinus   ::  nat -> nat -> nat
    natPlus    ::  nat -> nat -> nat
    || similarly for the other arithmetic operators

    natDisplay ::  nat -> num
```

[3]See Section 7.4 for guidelines on which functions should be packaged.

```
|| abstype nat implementation details

nat == num

makeNat x
    = error "makeNat:  non-negative integer expected",
         if (x < 0) \/ (~(integer x))
    = x, otherwise

natDisplay x = x

natEqual = (=)

natLess = (<)

natMinus x y
    = error "natMinus:  negative result", if (x - y) < 0
    = x - y, otherwise

natPlus = (+)

|| and similarly for the other operators
```

The above code illustrates the following points:

1. The abstype is useless on its own—the programmer must also provide underlying definitions for the type and the interface functions.
2. Every function name declared within a signature must have a corresponding definition. If a definition is missing, an error arises:

```
abstype stack *
with
  emptystack::stack *
  pop ::  stack * -> *
  push ::  stack * -> * -> stack *

stack * == [*]

emptystack = []

pop [] = error "empty stack"
pop (x :  xs) = x
```

```
compiling script.m
checking types in script.m
SPECIFIED BUT NOT DEFINED: push;
```

This rule has the clear advantage that it prevents a programmer from specifying something and then forgetting to code it!

3. Any function definition whose name matches a name which appears within an **abstype** signature must be of the type declared in the signature.

```
abstype stack *
with
  emptystack::stack *
  pop ::  stack * -> *
  push ::  stack * -> * -> stack *

stack * == [*]

emptystack = []

pop [] = error "empty stack"
pop (x :  xs) = x

push xs x = (x, xs)
```

```
compiling script.m
checking types in script.m
abstype implementation error
"push" is bound to value of type: *->**->(**,*)
type expected: [*]->*->[*]
(line  14 of "script.m")
```

This rule reinforces the principle that programmers should first consider the nature of the input and output types of their programs before worrying about the coding details.

4. The interface function definitions could appear anywhere in the program, but it is sensible programming practice to put them next to their declarations.

Abstract type usage

The following session shows how the `makeNat` function can be used to create new instances of `nat`s. The function `natPlus` is then used to add these two instances:

```
quantity1 = makeNat 3
quantity2 = makeNat 3
```

```
Miranda quantity1
<abstract ob>

Miranda natPlus quantity1 quantity2
<abstract ob>
```

In all of the above cases, the system response `<abstract ob>` is new, indicating that the representation is hidden. The programmer cannot be tempted to bypass these primitives and attempt direct manipulation of the underlying type. This has the considerable benefit that the underlying types and implementation details may be changed with no need to change any application programs that use the abstract type primitives—this is often termed *abstraction*.

This point is reinforced by the fact that the only type information available is the name of the **abstype:**

```
Miranda quantity1 ::
nat
```

Notice that lazy evaluation has the effect that computation is not actually carried out if there is no value which can be viewed by the user. Thus, in the following example, the `error` function for `(makeNat (-3))` is not invoked because the expression is not evaluated:

```
Miranda natPlus (makeNat (-3)) (makeNat 3.1)
<abstract ob>
```

Hiding the implementation means that the only way to access actual values is by one of the `nat` primitives. In the final example below, the attempt to display the value of a `nat` has caused some evaluation and consequently the `error` function has reported that it is not possible to turn a negative number into a `nat`:

```
Miranda natdisplay (natLess quantity1 quantity2)
False

Miranda natdisplay (natPlus quantity1 quantity2)
6

Miranda natDisplay (natPlus (makeNat (-3)) (makeNat 3.1))
program error: makeNat : negative input
```

Exercise 7.1

Provide function definitions for the `nat` primitives if a recursive underlying data representation is used as follows: `algnat ::= Zero | Succ algnat`.

Exercise 7.2

A *date* consists of a day, month and year. Legitimate operations on a date include: creating a date, checking whether a date is earlier or later than another date, adding a day to a date to give a new date, subtracting two dates to give a number of days, and checking if the date is within a leap year. Provide an abstract type declaration for a date.

7.1.1 Polymorphic abstypes

An **abstype** may also be used to declare a new polymorphic type; in this case, the name of the new type must be followed by the polytypes which will be used in the underlying definition. This is demonstrated in the following example of a *sequence*.

A double-ended list (sometimes known as a *sequence*), can be defined such that all the operations normally occurring at the front of a list (`:`, `hd`, `tl`) have mirror operations occurring at the end of the list. The basic set of operations provided are `seqNil` (which returns an empty sequence), followed by `seqConsL`, `seqConsR`, `seqHdL`, `seqHdR`, `seqTlL` and `seqTlR` (which provide the normal list operations at both ends of the sequence); two further functions `seqAppend` and `seqDisplay` are also provided for convenience. This type could be defined recursively or, as shown below, by extending operations on the built-in list type:

```
abstype sequence *
with
   seqNil     ::  sequence *
   seqConsL   ::  * -> (sequence *) -> (sequence *)
   seqConsR   ::  (sequence *) -> * -> (sequence *)
   seqHdL     ::  (sequence *) -> *
   seqHdR     ::  (sequence *) -> *
   seqTlL     ::  (sequence *) -> (sequence *)
   seqTlR     ::  (sequence *) -> (sequence *)
   seqAppend  ::  (sequence *) -> (sequence *)
                  -> (sequence *)
   seqDisplay ::  (sequence *) -> [*]

sequence * == [*]

seqNil = []
seqConsL = (:)
seqConsR anyseq item = anyseq ++ [item]

|| etc

seqDisplay s = s
```

Exercise 7.3

Complete the implementation for the `sequence` abstract type.

7.1.2 Properties of abstract types

Constraints

Five important constraints are imposed on the use of abstract types:

1. The arithmetic operators do *not* work for values of an **abstype**, regardless of the underlying type:

```
Miranda quantity1 + quantity2
type error in expression
cannot unify nat with num
```

2. Because the underlying abstract type representation is hidden, it is not possible to pattern match on an abstract type instance:

```
wrong_equal_numbers  ::  nat -> num -> bool
wrong_equal_numbers  0 0
                  =  True
wrong_equal_numbers  quantity1 x
                  =  natEqual quantity1 (makeNat x)
```

```
incorrect declaration
specified wrong_equal_numbers :: nat -> num -> bool
inferred wrong_equal_numbers :: num -> num -> bool
```

   The above error has occurred because the constant pattern `0` has appeared as the first argument for the function. The use of `0` as a value of type `nat` is restricted to the interface functions; if `0` occurs elsewhere then it is seen to be a value of type `num`.

3. The type name for an **abstype** must be linked to an existing type name through the use of a type synonym. This defines the underlying type for the **abstype**. It is *not* possible to define the underlying type in any other way—a potential mistake is to attempt to define the underlying type directly as an algebraic type. The first example below shows this error, and the second example demonstrates a correct definition:

```
abstype wrong_lights
with
  start_light ::  wrong_lights
  next_light ::  wrong_lights -> wrong_traffic_lights

wrong_lights ::= Green | Red | Amber | RedAmber

start_light = Green

next_light Green = Amber
next_light Amber = Red
next_light Red = RedAmber
next_light RedAmber = Green
```

```
compiling script.m
syntax error: nameclash, "wrong_lights" already defined
error found near line 6 of file "script.m"
compilation abandoned
```

```
abstype traffic_lights
with
  start_light::traffic_lights
  next_light ::traffic_lights -> traffic_lights

lights ::= Green | Red | Amber | RedAmber

traffic_lights == lights

start_light = Green

next_light Green = Amber
next_light Amber = Red
next_light Red = RedAmber
next_light RedAmber = Green
```

4. Only the declared interface functions may access the specified **abstype**. In the last example, any other function can be defined to operate upon values of type `lights`; what is new is that only the declared interface functions can operate upon values of type `traffic_lights`. The fact that the underlying type for `traffic_lights` is `lights` is not relevant to the rest of the program because values of the two types cannot be mixed—they are treated as entirely different types.[4]

[4]Note that Miranda does *not* provide data-hiding as part of its **abstype** facility, unlike some other languages, where the underlying type would also be concealed.

5. An **abstype** can only be defined at the top-level of a Miranda session; for example, it is not legal to define an **abstype** within a **where** block or function body.

Ordering abstract type values

Values of an abstract type inherit the ordering characteristics of the underlying type. Thus, if equality and relational ordering are defined on the underlying type then they will be defined for the **abstype** values:

```
Miranda quantity1 <= quantity2
True
```

However, reliance on the use of the built-in equality and relational operators *severely limits* the degree of abstraction achieved by the **abstype**; if the underlying representation of the abstract type is changed then the use of the built-in operators such as `=` and `<` cannot be relied upon. It is therefore recommended that if ordering primitives are required then they should be provided explicitly as interface functions.

7.1.3 Converting between abstract types

Sometimes there will be a requirement to convert from one abstract type to another. Initially this may seem difficult, because only the interface functions may use an abstract type's data. For example, if a program contains a `tree` abstract type (see Section 6.4.2) and a `sequence` abstract type (see Section 7.1.1), where should the function `tree_to_sequence` be defined? It cannot be defined inside the `tree` abstract type because it will not have access to the `sequence` underlying representation; similarly, it cannot be defined inside `sequence` because it will not have access to the `tree` underlying representation.

This apparent limitation is solved pragmatically in one of two ways: firstly by converting to a common intermediate data type; alternatively by coalescing the two abstract types.

1. If the two abstract types are not closely linked semantically then it is likely that the conversion mentioned above will not occur often and in this case it is sufficient to provide a "display" function and a "make" function for each; these two functions will use an intermediate form based on built-in types in order to provide the conversion. For example, given the abstract type `tree` and the abstract type `sequence` then the `sequence_to_tree` function can thereafter be defined externally to both abstract types, as is shown in the following program extract:

```
|| Define the tree abstract type:
abstype tree *
with

|| etc

tree * == mytree *
mytree * ::= Tnil | Tree (mytree *) * (mytree *)

displayTree Tnil = []
displayTree (Tree ltree node rtree)
   = displayTree ltree ++ [node]
     ++ displayTree rtree

makeTree order alist
    = accumulate (insertleaf order) Tnil alist
       ...

|| Define the sequence abstract type:
abstype sequence *
with

|| etc

sequence * == mysequence *
mysequence * ::= Seq [*]

displaySequence (Seq s) = s

|| etc

|| Define a conversion function:

sequence_to_tree order
   = (makeTree order) .  displaySequence
```

2. By contrast, if conversion between two abstract types is a very frequent requirement then this implies that they are in fact very closely linked semantically and should therefore be defined in tandem. The `sequence_to_tree` function can then be defined inside the combined abstract type body and will have access to all the necessary data:

```
abstype seqtree *
  with
    sequence_to_tree::  seqtree * -> seqtree *
    tree_to_sequence::  seqtree * -> seqtree *

                    || etc

seqtree * == myseqtree *

mytree * ::= Tnil | Tree (mytree *) * (mytree *)

myseqtree * ::= Seqtree (mytree *) | Seq [*]


sequence_to_tree (Seq []) = Seqtree Tnil
sequence_to_tree (Seq (front :  rest)) =
                    || etc

tree_to_sequence (Seqtree Tnil) = Seq []
tree_to_sequence (Seqtree (Tree ltree node rtree)) =
                    || etc
```

7.2 Showing abstract types

Most objects in a Miranda program may be printed by using the built-in function `show`. However, there is no built-in method for converting objects with an abstract type to a printable form. Thus, if `show` is applied to an object of abstract type, it will normally print as:

```
<abstract ob>
```

This behaviour for the function `show` may be modified by providing a special "show" function for a given **abstype**. The rule for doing this is to include in the definition of the abstract type a function with the name `showfoo` (where "foo" is the name of the abstract type involved). Thereafter, if the built-in function `show` is applied to an object of type `foo` then the function `showfoo` will automatically be called.

For example, consider the type `traffic_lights` where it is required to represent three lights by characters displayed one above the other (red, amber and green from top to bottom):

```
Miranda red_light::
traffic_lights
```

```
Miranda show red_light
*
O
O

Miranda show green_light
O
O
*

Miranda show redamber_light
*
*
O
```

In order for `show` to work in the above manner, it is necessary to define an extra interface function called `showtraffic_lights`:

```
abstype traffic_lights
with
start_light         ::  traffic_lights
next_light          ::  traffic_lights -> traffic_lights
showtraffic_lights  ::  traffic_lights -> [char]

lights ::= Green | Red | Amber | RedAmber

traffic_lights == lights

start_light = Green

next_light Green    = Amber
next_light Amber    = Red
next_light Red      = RedAmber
next_light RedAmber = Green

showtraffic_lights Green    = "O\n O\n*\n"
showtraffic_lights Red      = "*\n O\nO \n"
showtraffic_lights Amber    = "O\n *\nO \n"
showtraffic_lights RedAmber = "*\n *\nO \n"
```

In the above example, the special-purpose "show" interface function takes one argument (of type `traffic_lights`) and returns a value of type `[char]`.

Showing polymorphic abstypes

If the **abstype** involved is polymorphic then the new "show" function must take an extra argument, which is a function: that is, a function which knows how to "show" an object of the polymorphic type. In practice, the programmer never has to provide this additional function when `show` is applied because it is automatically provided by Miranda; however, the definition of the new "show" function must assume that the function is passed as the first argument and must use that function appropriately. For example, consider an **abstype** which mirrors the built-in list type:

```
abstype list *
with
  empty       :: list *
  add_to_list ::   * -> list * -> list *
  showlist    ::   (* -> [char]) -> list * -> [char]

alglist * ::= Nil_list | List * (alglist *)

list * == alglist *

empty = Nil_list

add_to_list x y = List x y

showlist f Nil_list   = "[]"
showlist f (List x y) = "( " ++ (f x) ++ " :  "
                        ++ (showlist y) ++ " )"
```

The general rule is as follows. Let "foo" be an abstract type name. To make objects of type "foo" printable, it is necessary to define a "showfoo" such that:

if foo is a simple type (not polymorphic)
showfoo :: foo − > [char]

*if foo is polymorphic in one type variable (foo *)*
*showfoo :: (∗− >[char]) − > foo * − > [char]*

*if foo is polymorphic in two type variables (foo * **)*
showfoo :: (∗− >[char]) − > (∗ ∗ − >[char]) -> foo ∗ ∗ ∗− > [char]

. . . and so on. Note that the *show* function must be declared in the signature of the abstract type, and that the name of the function is significant—if it were to be called *banana* rather than *showfoo* then it would not have any effect on the

behaviour of `show`. Similarly if it is not of the correct type then again it will not effect `show`, though in this case the compiler will print a warning message.

Exercise 7.4

Provide a *show* function for the `sequence` abstract type.

Exercise 7.5

Assuming that the underlying type for an abstract `date` type is a three number tuple (day, month, year), provide functions to display the day and month in US format (month, day), UK format (day, month) and to display the month as a string, such as "Jan" or "Feb".

7.3 Further examples of abstract types

This section now presents two more examples of the use of abstract types. The first example collects together the functions over a binary tree and presents an elegant method for providing a generic package for trees. The second, larger, example shows how an *array* data structure might be implemented in more than one way without affecting the way it is used in existing programs.

7.3.1 Trees as abstract types—generic packaging

The major disadvantage of defining a polymorphic search tree is that the ordering function must always be passed as an explicit parameter to the interface functions for tree modification and manipulation. This difficulty can be overcome by incorporating the ordering function into the data structure definition; thus, it is only necessary to specify the ordering function when a tree is first created and thereafter all other interface functions can find the ordering function by inspecting the data structure. The advantage of using an **abstype** in this situation is that, once the tree has been established, the user of the interface functions need not be aware of the embedded ordering function.

```
ordering * == (* -> * -> bool)
tree *  ::= Tnil
            | Tree (tree *) * (tree *)
orderedTree * ::= OrderedTree (ordering *) (tree *)

abstype absTree *
  with
        newtree    ::  ordering * -> absTree *
        insertleaf ::  absTree * -> * -> absTree *
        flatten    ::  absTree * -> [*]

absTree * == orderedTree *

newtree order = OrderedTree order Tnil

insertleaf (OrderedTree order anytree) item
     =  OrderedTree order (insert anytree)
        where
        insert Tnil = Tree Tnil item Tnil
        insert (Tree ltree node rtree)
            =  Tree (insert ltree) node rtree,
               if (order item node)
            =  Tree ltree node (insert rtree),
               otherwise

flatten (OrderedTree order anytree)
     =  inorder anytree
        where
        inorder Tnil = []
        inorder (Tree ltree node rtree)
            =  inorder ltree ++ [node] ++ inorder rtree
```

Instances of this `absTree` can be created by passing the appropriate sorting function, for example:

```
lessthan ::  num -> num -> bool
lessthan = (<)

num_absTree ::  absTree num
num_absTree = newtree lessthan
```

Thus, an `absTree` in increasing numeric order might be as shown in Figure 7.1.

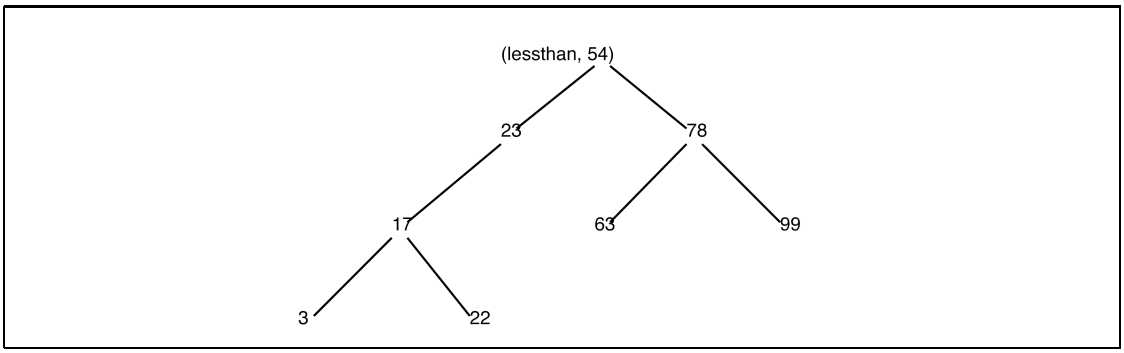

Figure 7.1 Sample Atree.

Exercise 7.6

An alternative representation of the `Atree` would be:

```
abstype other_tree *
with
    || declarations

ordering * == * -> * -> bool
other_tree * ::= Anil (ordering *)
                 | ATree (ordering *) (other_tree *) * (other_tree *)
```

What would be the consequences for the abstract type implementation?

7.3.2 Arrays as abstract types—alternative implementations

The following example demonstrates the usefulness of concealing a particular representation of an abstract type from the programmer. The first implementation of `array` could be replaced by an implementation as a list of tuples, without altering its public interface.

An array as a list of lists

This implementation shows an abstract type representation of a two-dimensional array, where an array may be defined as a fixed size aggregate data structure whose elements may be changed or retrieved by reference to an index.

A two-dimensional array data structure can very easily be represented as a list of lists (where the innermost lists represent the rows of the array). The function definitions are fairly straightforward but rely upon the existence of some of the list handling functions introduced in Chapter 3 and some of the higher order functions of Chapter 4. As recommended for larger programs, the implementation is presented as a literate script:

```
>|| List of lists array representation.  (Page 1 of 2)

> abstype array *
> with
>   num_rows    ::  array * -> num
>   num_cols    ::  array * -> num
>   init_array  ::  num -> num -> * -> array *
>   change_item ::  array * -> num -> num -> * -> array *
>   get_item    ::  array * -> num -> num -> *

> array * == [[*]]

> num_rows anarray = # anarray

> num_cols [] = 0
> num_cols anarray = # (hd anarray)

replace applies a function ff to a item in a list
giving an identical list except that the item is
replaced by the result of the function application

> replace ::  num -> (*->*) -> [*] -> [*]
> replace pos ff anylist
>   = error "replace:  illegal position",
>       if (pos > # anylist) \/ (pos <= 0)
>   = (take (pos - 1) anylist)
>      ++ [ff (get_nth pos anylist)]
>      ++ (drop pos anylist), otherwise

>get_nth ::  num -> [*] -> *
>get_nth n anylist
>     = anylist !  (n - 1)
```

```
>|| List of lists array representation continued (Page 2)

init_array produces an initial array from:
  i.  how many rows and columns the array should have
  ii. and a starting value for all the elements.

repeat creates a list of the right length to represent
a row, and also constructs an array of many such rows.

> init_array nrows ncols first_val
>   = error "init_array:  negative rows",
>     if nrows < 0
>   = error "init_array:  negative columns",
>     if ncols < 0
>   = hd [a | b<-[take nrows (repeat x)];
>         a<-[take ncols (repeat b)]], otherwise


change_item returns the input array with an altered item.
replace's outer application finds the appropriate row and
its inner application is applied to that row to replace
the item in the column position with a new value using the
built-in combinator const (cf cancel as shown in Chapter 4).
The inner use of replace is a partial application, it is
not fully evaluated until the appropriate row has been
selected.

> change_item anarray row column newvalue
>   = error "change_item:  illegal position",
>     if (row <= 0) \/ (column <= 0)
>         \/ (row > num_rows anarray)
>         \/ (column > num_cols anarray)
>   = replace row (replace column (const newvalue)) anarray,
>      otherwise

> get_item anarray row column
>   = error "get_item:  illegal position",
>      if (row <= 0) \/ (column <= 0)
>         \/ (row > num_rows anarray)
>         \/ (column > num_cols anarray)
>   = get_nth column (get_nth row anarray), otherwise
```

The `array` primitives can now be used directly in programs and also to build more complex utilities:

```
> get_col ::  array * -> num -> [*]
> get_col anarray column
>   = error "get_col:  illegal column number",
>      if (column <= 0) \/ (column > num_cols anarray)
>   = map (converse (get_item anarray) column)
>         [1..(num_rows x)], otherwise

> get_row ::  array * -> num -> [*]
> get_row anarray row
>   = error "get_row:  illegal row number",
>      if (row <= 0) \/ (row > num_rows anarray)
>   = map (get_item anarray row)
>         [1..(num_cols x)], otherwise
```

An array as a list of tuples

The first implementation of the *array* can now be replaced with an implementation as a list of tuples, without changing the interface function signatures and therefore without the need to change any part of any program that uses such an *array* type.

```
>|| list of tuples array representation.  (Page 1 of 2)

> abstype array *
> with
>   num_rows    ::  array * -> num
>   num_cols    ::  array * -> num
>   init_array  ::  num -> num -> * -> array *
>   change_item ::  array * -> num -> num -> * -> array *
>   get_item    ::  array * -> num -> num -> *

> array * == (num,num,*,[(num,num,*)])

The array is represented by its dimensions (nrows * ncols),
a value (default) for all items that have not been updated,
together with a list of tuples representing the row and
column position and new value of any updated item "changes"

> num_rows (nrows, ncols, default, changes) = nrows

> num_cols (nrows, ncols, default, changes) = ncols
```

```
>|| list of tuples array representation.  (Page 2)

> init_array nrows ncols default
>    = error "init_array:  inappropriate dimensions",
>       if (nrows < 0) \/ (ncols < 0)
>          \/ anyfractional [nrows,ncols]
>    = (nrows, ncols, default, []), otherwise

Changing an item involves removing the changes list entry and
appending the new entry.  There is no check whether the new
value differs from the existing entry or the default value

> change_item (nrows,ncols,default,changes) row col newvalue
>   = error "change_item:  inappropriate dimensions",
>      if invalid nrows ncols row col
>   = (nrows,ncols,default,newchanges), otherwise
>     where
>      newchanges
>         = (filter ((~) .  (compare row col)) changes)
>           ++ [(row,col,newvalue)]

Getting an item involves filtering it from the changes list;
if there is no entry then the default entry becomes the head
of the items list otherwise it is ignored.

> get_item (nrows,ncols,default,changes) row col
>     = error "get_item:  inappropriate dimensions",
>       if invalid nrows ncols row col
>     = (third .  hd) items, otherwise
>       where
>        third (row, col,value) = value
>        items = (filter (compare row col) changes)
>                ++ [(row,col,default)]

> compare row1 col1 (row2, col2, item)
>    = (row1 = row2) & (col1 = col2)

> anyfractional nlist = and (map ((~) .  integer) nlist)

> invalid nrows ncols row col
>    = (row > nrows) \/ (row <= 0) \/ (col > ncols) \/
>      (col <= 0) \/ anyfractional [nrows,nrows,col,ncols]
```

The functions `get_col` and `get_row` defined using the first (list of lists) version of `array` can now be substituted with the second (list of tuples) version *without modification of the existing code*; all that is necessary is to recompile the program, to ensure that the new representation is used throughout (recompilation will be done automatically on exit from the editor).

Exercise 7.7

A *queue* aggregate data structure (Standish, 1980) can be defined as either being empty or as consisting of a queue followed by an element; operations include creating a new queue, inserting an element at the end of a queue and removing the first element in a queue. The following declares a set of primitives for a polymorphic abstract type `queue`:

```
abstype queue *
with
   qisempty     = queue * -> bool
   qtop         = queue * -> *
   qinsert      = queue * -> * -> queue *
   qcreate      = queue *
```

Provide an implementation for this abstract type.

7.4 Guidelines for abstract type primitive selection

Apart from the criterion that an **abstype** must provide all the operations that are necessary and sufficient to manipulate the underlying type, there are no other rigid rules to determine which functions should be provided to make the abstract type easy to use. Two of the necessary primitives for all abstract types are *creation* and *inspection*. Many abstract types will also have *initialization* and *modification* primitives, the former sometimes being incorporated into the *creation* primitive (as with the `abstree` definition). A useful set is an initialization primitive to create an "empty" object, together with a modification primitive to change the underlying value and an inspection primitive to view that value—this works particularly well when the underlying type is a recursive algebraic type. Furthermore, it is often useful to provide the equivalents of `map` and `reduce` for the abstract type. It is however unwise to provide too large a set of primitive functions, because this will tend to reinforce the current underlying representation and make it difficult to make future changes to this representation.

For example, the above `array` implementation provided primitives for creating a new array, together with those for inspecting and changing one element. It can be shown that these three are sufficient for array manipulation, but in practice an

equality primitive and functions to extract the contents of a row or column would probably be provided. Indeed, for many abstract types it will often be the case that additional functions, which build on the basic primitives, should be offered by the abstract type implementor. The rationale for this is threefold:

1. To eliminate programmer effort and potential error, by writing these "non-essential" primitives once only.
2. To provide efficient implementations based on a knowledge of the underlying algebraic type representation.
3. There is sometimes a choice of which primitives comprise a necessary and sufficient set of operations. For example, an abstract type for binary Boolean algebra could be represented by a unary *not* operator, together with a dyadic *and* operator; or by *not*, together with the dyadic *or* operator. In this case, it is obvious that both dyadic operators should be provided to reflect different programmers' views of the data.

Notice that it is always safe to change an abstract type's underlying representation in order to *extend* the functionality of the type; however it is *not* safe to decrease the number of interface functions without potentially affecting parts of the program already written.

Finally, there are many instances of general purpose types (such as the `coords` algebraic type) for which it is not possible to predict an adequate and easy to use set of primitives. For these types it does not make sense to force them into an abstract type.[5]

7.5 Summary

This chapter has extended the principle of strong typing introduced in Chapter 1 and emphasized throughout this book. The use of *abstract types* helps to ensure that values of a certain type (a basic type or a new type) are only operated on by appropriately defined functions. Not only does this help to detect errors, it also serves to document the program and helps to keep the program structure clear.

This mechanism also extends the principle of structured programming discussed in Chapter 5 to show how a programmer can have safer and more reusable code. The **abstype** code is generally code that will be used by many programmers for different applications and can be provided in a library with the underlying implementation hidden from the application programmers.

Large-scale programming benefits enormously from the rigorous application of the concepts of closure, modularity, encapsulation and self-documentation, as described and recommended throughout this book.

[5]However, it may be reasonable to encapsulate a "library" of related operations in a file, as shown in Chapter 9