t_1 or t_2, and that it is really unnecessary to convert it to a natural number for the standard sort of recursion variant that you have already seen. But to make this idea clearer we shall first go through the unstreamlined reasoning.

We shall define a function `treesize` of type `tree * -> num`, with no pre-conditions, satisfying the properties

`treesize` t_1 < `treesize (Node` t_1 x t_2`)`
`treesize` t_2 < `treesize (Node` t_1 x t_2`)`

Then `treesize` t is a recursion variant for `insertT` n t.

```
treesize::(tree *) -> num
treesize Emptytree = 1
treesize (Node t1 x t2)
    = (treesize t1) + 1 + (treesize t2)
```

But how do we know that `treesize` t always terminates? Well, it does not! You can define infinite trees just as easily as infinite lists (for example, t = `Node` t 0 t), and for them `treesize` does not terminate. So we have only actually shown that `insertT` n t works for *finite* trees t, those for which `treesize` t gives a result. Strictly speaking, we should state the finiteness as a pre-condition for `insertT`, but just as for lists we will leave it implicit.

Now it is important not to see `treesize` as a clever trick cooked up specially for `insertT`. It works equally well for *any* function of trees whose recursive calls are on the left or right subtrees of the main argument, and this is by far the most common pattern.

What is more, the numerical value of `treesize` t is not in itself very important — there are many other functions satisfying the specification of `treesize`, all serving just as well. What you should see in the specification is the idea of the tree itself 'decreasing' to a subtree, and hence serving as a recursion variant:

t_1‘ < '`Node` t_1 x t_2 and
t_2‘ < '`Node` $t_1 x$ t_2

This kind of '<' is explored more mathematically in Appendix A, which in particular looks at what properties of '<' are needed; but for the present it is enough to remember that it gives a more general kind of recursion variant. If you are unsure about this you could always use `treesize`, but we prefer you to use the *structural induction* that is described in the following section.

7.8 Structural induction

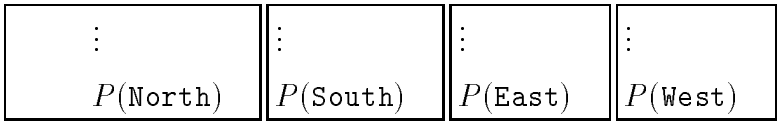
The real purpose of this section is to show how to introduce new induction principles for recursively defined datatypes (such as `tree *`), although we are going to start off with non-recursive types that do not lead to induction. The key idea is to see a direct link between the type definition and the box proof

structure of induction (and also, though we are not going to discuss it so much in this section, function definitions).

Type definition	*Induction proofs*	*Function definitions*
constructors	boxes	cases
arguments of constructor	new constants in box	variables matched in pattern
recursion	induction hypotheses	recursion

This should become clearer with the examples. We start off with a couple of non-inductive ones.

- The first example illustrates the first line only of the above table: it has four constructors (without arguments) and four corresponding boxes.
  `direction ::= North | South | East | West`

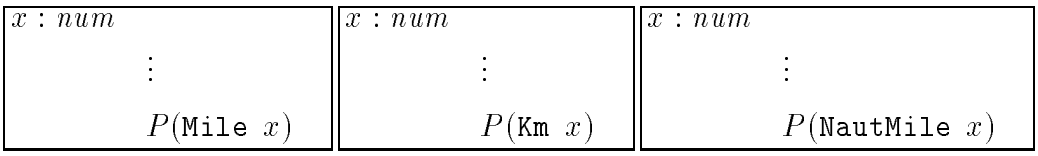


  $\forall d : \texttt{direction}.P(d)$

  This is really nothing more than $\forall$-introduction (see Chapter 17) and $\vee$-elimination (Chapter 16) based on an axiom

$$\forall d : \texttt{direction}.(d = \texttt{North} \vee d = \texttt{South} \vee d = \texttt{East} \vee d = \texttt{West})$$

  The boxes given above are a streamlined version setting out what is needed to complete the proof (exercise — show how this works).
- The second example moves on to the second line of the table, bringing in constructors with arguments:
  `distance ::= Mile num | Km num | NautMile num`



  $\forall d : \mathit{distance}.\ P(d)$

  Again (exercise) this is no more than you would obtain from logic, using $\forall$-introduction, $\vee$- and $\exists$-elimination Chapters 16 and 17, and an axiom

$$\forall d : \texttt{distance}.\ ((\exists x : \texttt{num}.\ d = \texttt{Mile}\ x) \vee (\exists x : \texttt{num}.\ d = \texttt{Km}\ x) \vee (\exists x : \texttt{num}.\ d = \texttt{NautMile}\ x))$$

- Natural numbers (and simple induction):

```
diynat ::= Zero | Suc diynat
```

$\vdots$	N : `diynat` $\quad P(N)$
$P($`Zero`$)$	$\vdots$
	$P($`Suc` $N)$

$\forall n$: `diynat`$.P(n)$

This is exactly the simple induction you know already, but translated into the notation for the do-it-yourself natural numbers.
Now because there is recursion in the definition of `diynat`, we have the inductive hypothesis $P(N)$, and that takes this example beyond mere logic. You could not justify the induction hypothesis solely from an axiom such as

$$\forall n : \texttt{diynat}.\ (n = \texttt{Zero} \vee \exists m : \texttt{diynat}.\ n = \texttt{Suc}\ m)$$

so the induction hypothesis is a free gift. (It is not completely free. The cost is the restriction to *finite* natural numbers, even though Miranda can cope with some infinite ones.)

- Lists:

```
diylist * ::= Emptylist | Cons * (diylist *)
```

$\vdots$	X : $*$
$P($`Emptylist`$)$	XS : `diylist` $*\quad P(XS)$
	$\vdots$
	$P($`Cons` $X\ XS)$

$\forall xs$: `diylist` $*.P(xs)$

Again, this is just a familiar (list) induction translated into the do-it-yourself notation.
Notice how because `cons` has two arguments, there are two new constants X and XS in the proof box. But only its second argument is recursively of type `diylist` *, so there is only one induction hypothesis, $P(XS)$.

- Finally, we come to tree induction:

```
tree * ::= Emptytree | Node (tree *) * (tree *)
```

$\vdots$	t_1 : `tree` $*\quad P(t_1)$
$P($`Emptytree`$)$	x : $*$
	t_2 : `tree` $*\quad P(t_2)$
	$\vdots$
	$P($`Node` $t_1\ x\ t_2)$

$\forall t$: `tree` $*.P(t)$

This is an entirely new induction principle! It says that to prove $\forall t : tree * [P(t)]$, it suffices to prove

- a base case, P(`Emptytree`);
- an induction step, P(`Node` t_1 x t_2), assuming that $P(t_1)$ and $P(t_2)$ both hold (two induction hypotheses).

(All this is subject to the usual proviso, that it only works for finite trees — in Miranda, infinite trees are just as easy to define as infinite lists.)

Is this induction principle really valid? As it happens, it is, and it is justified in Exercise 25. But it is not so important to understand the justification as the pattern of turning a datatype definition into an induction principle.

The following is an application. (The specifications are not given formally, but you can give informal proofs that the definitions satisfy the informal specifications.)

```
flatten :: (tree *) -> [*]
||pre:  none
||post: the elements of flatten t are exactly the node values of t
flatten Emptytree = []
flatten (Node t1 x t2) = (flatten t1) ++ (x:(flatten t2))

revtree :: (tree *) -> (tree *)
||pre: none
||post: revtree t is t "seen in a mirror"
||      (with left and right reversed)
revtree Emptytree = Emptytree
revtree (Node t1 x t2) = Node (revtree t2) x (revtree t1)
```

We can use tree induction to prove that

$\forall t : tree * .$ `flatten (revtree` t`) = reverse(flatten` t`)`

base case: `Emptytree`

```
flatten (revtree Emptytree) =flatten Emptytree
                            = []
                            = reverse []
                            = reverse (flatten Emptytree)
```

induction step: `Node` t_1 x t_2

```
flatten (revtree (Node t1 x t2))
= flatten (Node (revtree t2) x (revtree t1))
= (flatten (revtree t2))++(x : (flatten (revtree t1)))
= (reverse(flatten t2))++[x]++(reverse(flatten t1))    induction
```

$= \texttt{reverse}((\texttt{flatten}\ t_1)\texttt{++}[x]\texttt{++}(\texttt{flatten}\ t_2))$
$= \texttt{reverse}(\texttt{flatten}\ (\texttt{Node}\ t_1\ x\ t_2))$

The pattern works for *any* datatype `newtype` that is defined using constructors. The key points to remember are

- There is a box for each constructor.
- Within a box, there is a new constant introduced for each argument of the corresponding constructor.
- There is an induction hypothesis for each argument whose type is `newtype` used recursively.
- The property proved inductively is proved only for *finite* values of `newtype`.
- *Base cases* are those boxes with no induction hypotheses; *induction steps* are those with at least one induction hypothesis.
- The method can be extended to *mutually* recursive types, each defined using the others. Then you need separate properties for the different types and you prove them all together, using induction hypotheses where there is any kind of recursion.

We will describe the general principles, though to be honest you may see these more clearly from the examples already given.

Each alternative in a type definition corresponds to a box in the proof, so let us concentrate on one alternative:
`thing ::= ... | A s1 ... sn | ...`
`A` is a constructor, it has n arguments, and they are of types $s_1, \ldots, s_n$. *Some* of these types may be `thing` again, using recursion. They will give induction hypotheses:

$x_1 : s_1$
$\vdots$
$x_n : s_n$
$P(x_i)$
$P(x_j)$
$\vdots$
$P(\texttt{A}\ x_1 \cdots x_n)$

$\forall x : \texttt{thing}.\ P(x)$

Recursion variants

Whenever a type `newtype` is defined using constructors, there is a natural format for recursively defined functions on `newtype`, using pattern matching: for each constructor you have a separate case with a pattern to extract the arguments of the constructor, and the arguments of type `newtype` will be used as arguments for the recursive calls of the function.

As long as you keep to this format, and also as long as you restrict yourself to finite elements of `newtype`, the 'circular reasoning' will be valid and you will not need to define a recursion variant.

What is happening in effect is that the argument of type `newtype` is itself being used as a recursion variant, 'decreasing' to one of its components. This can be justified by defining a numerical recursion variant of type `newtype -> num` that counts the number of constructors used for values of `newtype`. It can also be justified using the structural induction just described.

7.9 Summary

- One way of combining types to form new ones is to form a *tuple*-type (for example, a pair, or a triple or a quadruple). Tuple-values are formed by using the constructor `(,...,)`.
- Using tuples, functions can return more than one result by packaging their results into a single tuple.
- A *pattern* serves two purposes. Firstly it specifies the form that arguments must take before the rule can be applied; secondly it decomposes the arguments and names their components.
- Multi-argument functions (also called *curried* functions) are functions which take more than one argument (as opposed to those functions which operate on a single argument such as a tuple).
- An advantage of currying is that a curried function does not have to be applied to all of its arguments at once. Curried functions can be *partially applied*, yielding a function which is of fewer arguments.
- Every expression has a type associated with it and each type has associated with it a set of operations which are meaningful for that type.
- Functional languages are *strongly-typed*, that is, every well-formed expression can be assigned a type that can be deduced from its subexpressions alone. Any expression which cannot be assigned a sensible type (that is, is not well-typed) has no value and is rejected before evaluation.
- *Generic* or *polymorphic* (many-formed) types are represented using *type variables* `*`, `**`, `***` etc., each of which stands for an arbitrary type.

Within a given type expression, all occurrences of the same type variable refer to *the same* unknown type.

- You can define a type by listing the alternative forms of its values (separated by `|`). Each alternative form is a constructor (whose name begins with a capital letter) applied to some number of arguments. It represents 'the arguments packaged together in a wrapper that is clearly marked with the constructor's name'.
- This method subsumes the ideas of enumerated types, union types and recursively defined types (such as trees).
- The type definition determines both a natural format for recursive definitions of functions taking arguments from the type, and an induction principle for proving properties of values of the type.
- If you restrict yourself to using the 'natural format of recursive definitions' then you can use 'circular reasoning' just as though you had a recursion variant.
- Miranda allows infinite values of the new types. The methods here apply only to the finite values.

7.10 Exercises

1. What are the types of `+`, `->`, `-`, `++`, `#`, `!`, `>=`, `=`, `hd` and `tl`?
2. Prove by induction on xs_1 that `zip` satisfies

$$\forall xs_1, xs_2 : [*].\forall ys_1, ys_2 : [**].(\#xs_1 = \#ys_1 \wedge \#xs_2 = \#ys_2 \rightarrow$$
$$\texttt{zip}\ (xs_1\texttt{++}xs_2)\ (ys_1\texttt{++}ys_2) = (\texttt{zip}\ xs_1\ ys_1)\texttt{++}(\texttt{zip}\ xs_2\ ys_2))$$

3. Prove by induction on xs that `unzip` satisfies its specification, namely that

$$\forall xs : [*].\forall ys : [**].(\#xs = \#ys \rightarrow \texttt{unzip}(\texttt{zip}\ xs\ ys) = (xs, ys))$$

4. (a) Explain why the expression `zip (unzip` ps`)` is not well-typed. Can you make it well-typed by redefining `zip`?

   (b) Prove by induction on ps that

$$\forall ps : [(*, **)].\forall xs : [*]\forall ys : [**].$$
$$(\texttt{unzip}\ ps = (xs, ys) \rightarrow \texttt{zip}\ xs\ ys = ps)$$

   (NOTE: box proofs will help you, but you will need to use a little extra thought to deal with the pattern matching.)
5. Let P be a property of elements of type `*`, and consider a function `separate_P` specified as follows. (How it is defined will depend on P.)

```
separate_P :: [*] -> ([*], [*])
||pre:  none
||post: (A)x:* ((isin(x,Ps) -> P(x))
||                  & (isin(x,notPs) -> not P(x)))
||      & Merge(Ps,notPs,zs)
||          where (Ps,notPs) = separate_P zs
```

`separate_P` is supposed to 'demerge' the elements of `zs` into those satisfying P and those not.

Prove that this specification specifies the result uniquely.

6. (a) Recall the function `scrub` of Exercise 10, Chapter 6. Show that scrub satisfies the following specification:

```
scrub :: * -> [*] -> [*]
||pre:  none
||post: (E)xs:[*] (xs, scrub x ys) = separate_P ys
||        where (given x) P(u) is the property u = x.
```

   (b) Specify `count` in a similar way.

   (c) Use the uniqueness property of the specification of `separate_P` to prove some of the properties of `scrub` and `count` given in the exercises in Chapter 6.

7. Suppose that the names of the employees of a Department of Computing are stored as a list of pairs, for example

```
[("Broda","Krysia"),("Eisenbach","Susan"),
("Khoshnevisan","Hessam"),("Vickers","Steve")]
```

   Declare and define a function `display` which, given the current staff list, will return a string in the following format:

   K. Broda

   S. Eisenbach

   H. Khoshnevisan

   S. Vickers

   Assume that everyone has exactly one forename.

8. Define and declare the type of a function that, given any triple whose first component is a pair, returns the second component of that pair.

9. Give an example of an expression (that is, just *one* expression) that contains two occurrences of the empty list, the first occurrence having type `[num]` and the second type `[char]`.

10. Discuss whether the expression `smaller (quotrem (7,3))` is well-formed or not. If not explain why.

11. Given the data type `tree num` write a function `tmax` which finds the maximum element stored in a non-empty tree. (HINT: you may use a

function `largest` which returns the largest of three numbers.)

12. Define a data type `tree2_3` in which a value is either `Empty` or is a node which holds an item and has left and right subtrees, or is a node which holds two items and has a left, middle and a right subtree. All subtrees are of type `tree2_3` and all items stored in the tree have the same type.
13. For the sake of this question, take an *expression* in the variable x to be either

    - a number, for example 1
    - a variable (any character),
    - or the sum, difference or product of two expressions.

    Below is the definition of a type *expression* in Miranda using data constructors. It is recursive in that an expression can contain other expressions:

```
expr::= Number num | Variable char |
        Sum expr expr |
        Difference expr expr |
        Product expr expr
```

    The rules for partial differentiation of simple expressions with respect to x are

$$\frac{\partial n}{\partial x} = 0 \qquad \text{— where } n \text{ is a number}$$
$$\frac{\partial x}{\partial x} = 1$$
$$\frac{\partial y}{\partial x} = 0 \qquad \text{— if } y \text{ is different from } x$$
$$\frac{\partial (E_1+E_2)}{\partial x} = \frac{\partial E_1}{\partial x} + \frac{\partial E_2}{\partial x} \qquad \text{— where } E_1, E_2 \text{ are } \textit{any} \texttt{ exprs}$$
$$\frac{\partial (E_1-E_2)}{\partial x} = \frac{\partial E_1}{\partial x} - \frac{\partial E_2}{\partial x}$$
$$\frac{\partial (E_1\times E_2)}{\partial x} = \frac{\partial E_1}{\partial x} \times E_2 + E_1 \times \frac{\partial E_2}{\partial x}$$

    Define a function `differentiate` of type `char -> expr -> expr` that will perform these differentiation rules, `differentiate x e` representing $\frac{\partial e}{\partial x}$. For example,

```
differentiate x (Sum e1 e2)  =
    Sum (differentiate x e1) (differentiate x e2)


differentiate x (Number n) = Number 0
```

14. Show that any application of your function `differentiate` will terminate.

How might you write a `simplify` function to reduce such expressions to a simpler form? For example, simplifying a multiplication by 0 would result in replacing $0 \times x$ and $x \times 0$ by 0.

15. Give specifications (pre-conditions and post-conditions) in logic for the following programs.

    (a) `last :: [*] -> *` ; returns the last element of a list.

    (b) `front :: num -> [*] -> [*]` ; `front n xs` returns the list of the first `n` elements of `xs` if `n` $\leq$ `#xs`, otherwise it returns `xs`.

    (c) `make_unique :: [*] -> [*]`; `make_unique xs` removes the duplicates in *xs*. The elements need not be in the same order as in `xs`.

16. Define a function

```
sub :: expr -> char -> expr -> expr
||pre:  none
||post: sub e1 v e2 = e2 with e1 substituted for every
||          occurrence of var v
```

    and use structural induction on e_1 to prove

$$\forall e_1, e_2, e_3 : \texttt{expr}\ \forall v : \texttt{char}\ (\texttt{sub}\ e_3\ v\ (\texttt{sub}\ e_2\ v\ e_1) = \texttt{sub}(\texttt{sub}\ e_3\ v\ e_2)\ v\ e_1)$$

17. This exercise requires you to implement a series of Miranda functions which manage `dictionary`s stored as ordered binary trees. We define

```
word == [char]
dictionary ::= Empty | Node dictionary word dictionary
```

    Show that `dictionary` is equivalent to `tree word`. Write the following functions:

    (a) `create_new_dictionary`, which creates an empty dictionary.

    (b) `add_word`, which adds a word to a dictionary.

    (c) `lookup`, which returns whether a word is in the dictionary.

    (d) `count_words`, which returns the number of words in a dictionary.

    (e) `delete_word`, which deletes a word from a dictionary.

    (f) `find_word`, which returns the nth word in a dictionary, or returns an empty word if there is no nth word.

    (g) `list_dictionary`, which produces a list of all the words in a dictionary, one to a line. (Use a function such as `flatten`.)

18. Write coding and decoding functions for translating between *diynat* and ordinary natural numbers:

```
numtonat :: num -> diynat
nattonum :: diynat -> num
```

Prove that

$$\forall x : \texttt{num.nattonum(numtonat } x) = x$$

and

$$\forall n : \texttt{diynat.numtonat(nattonum } n) = n.$$

Also, write equivalents for `diynat` of the ordinary arithmetic operations and prove that they satisfy their specifications, for example,

```
add :: diynat -> diynat -> diynat
||pre:  none
||post: (nattonum (add m n))=(nattonum m)+ (nattonum n)
add Zero n=n || represents 0+n=n
add (Suc m)n= Suc (add m n) || represents (m+1)+n=(m+n)+1
```

19. Do something similar for `diylist *`.
20. Define a Miranda program to test whether a tree is ordered.
21. Specify and define a Miranda function to count how many times a given value occurs in a given ordered tree. Prove (informally but rigourously) that the definition satisfies the specification.
22. Use recursion to define some infinite trees.
23. Use `insertT` to define a function `build` to the following specification:

```
build :: [*] -> (tree *)
||pre: none
||post: build xs is ordered, and its node values are exactly
||      the elements of xs.
```

24. Show (you can use the method of 'trees as recursion variants') that if `t` is an ordered tree then `flatten t` is an ordered list.
Hence show that the following definition satisfies the specification for `sort` (Chapter 6):

```
treesort :: [*] -> [*]
treesort xs = flatten (build xs)
```

25. Suppose $P(t)$ is a property of trees, and consider the following sentences:

$$Q \stackrel{\text{def}}{=} \forall t : (tree\ *).P(t)$$

$$R \stackrel{\text{def}}{=} \forall n : nat.\forall t : (tree\ *).(\texttt{treesize } t = n \rightarrow P(t))$$

Remember, as always, that we are talking only about *finite* trees.

(a) Use a box proof to show that $Q \leftrightarrow R$.

(b) Suppose you have a proof by tree induction of Q. Show how you can use its ingredients to create a proof by course of values induction of R. (Use the specification of `treesize`.)

Chapter 8

Higher-order functions

You have already seen examples of functions delivering functions as results, namely the curried functions. These were easy to understand as functions with more than one argument. Much more subtle are functions that take other functions as arguments — some examples from mathematics are differentiation and integration. These are called *higher-order* functions. The argument and result types of functions are *not* restricted to being values.

Differentiation takes one function, f, say, of type `num -> num`, and returns another, usually written f'. So there is a higher-order function `diff` of type `(num -> num) -> num -> num` such that

`diff` $f\ x = f'(x) =$ derivative of f at x

8.1 Higher-order programming

Consider the definitions in Figure 8.1. Although they define different functions, their *pattern of recursion* is the same. In all definitions a function `f` is applied to every element of a list, where `f` is `f x = x*x`, `f x = factorial x` and `f x = x mod 2 = 0` respectively.

It is possible to express such common patterns of recursion by a few higher-order functions. We begin by defining a higher-order function corresponding to the above three definitions and then discuss other patterns.

8.2 The higher-order function `map`

If `f` is a function of type `* -> **`, then the idea is to define a function `map f` of type `[*] -> [**]` that works by applying `f` one by one to all the elements of a list. This can be specified in an obvious way using indices. Since the first argument of `map` is a function `f`, `map` itself is a higher-order

```
squares         ::  [num] -> [num]
||pre:          none
||post:         #ys = #xs
||              & (A) i:nat.(0 <= i< #xs->ys!i = (xs!i)^2)
||                  where ys = squares xs
squares []      =   []
squares (x:xs)  =   (x * x) :  (squares xs)

factlist        ::  [num] -> [num]
||pre:          none
||post:         #ys = #xs
||              & (A) i:nat.  (0 <= i < #xs -> ys!i
||                  = factorial(xs!i))
||                  where ys = factlist xs
factlist []     =   []
factlist (x:xs) =   (factorial x) :  (factlist xs)

iseven          ::  [num] -> [bool]
||pre:          none
||post:         #ys = #xs
||              & (A) i:nat.  (0 <= i < #xs ->
||                  ys!i = (xs!i mod 2 = 0) )
||                  where ys = iseven xs
iseven []       =   []
iseven (x:xs)   =   (x mod 2 = 0) :  (iseven xs)
```

Figure 8.1 Pattern of recursion

function.

In fact, the pattern of recursion expressed by `map` is so common in list-manipulating programs that `map` is *predefined* in many evaluators or is included in a library, for example as in Miranda.

```
map        :: (* -> **) -> [*] -> [**]
||pre:  none
||post: #ys = #xs
||      & (A) i:nat. (0 <= i < #xs -> ys!i = f(xs!i))
||      where ys = map f xs
map f[]    = []
map f(x:xs) = (f x):(map f  xs)
```

The definitions of Figure 8.1 can now be more concisely defined in terms of `map`:

```
squares  =  map (^2)
factlist =  map factorial
iseven   =  map f
              where f x = (x mod 2 = 0)
```

For example,

```
squares[1,3,2] = map(^2)[1,3,2] = [1,9,4]
```

Note that partial application is especially convenient when used in conjunction with higher-order functions, as can be seen from the new definition for `squares`.

Example

Integration (we mean *definite* integration) takes a function f and two limits, a and b, and returns a number. One way of calculating the definite integral is by cutting the domain of integration into equal-sized *slices*, and guessing the average height of the function in each slice. For example, if the function is to be integrated from 0 to 5 in 10 slices, the slices are: 0 to 0.5, 0.5 to 1, and so on up to 4.5 to 5. The guessed height for each slice is simply the value of the function in the centre of each slice, such as $f(4.75)$ for the last slice in the example above. This assumes that the slices are rectangular-shaped, rather than whatever curved shape the function actually has.

The guessed *area* of a slice is then the width (0.5 each, in the example) times the guessed average height. The final answer is the sum of the areas of all the slices. The type of a function `integrate` which calculates the area under a curve could be declared as follows:

```
function == num -> num

integrate :: function -> num -> num -> num -> num
||args are <function> <start> <finish> <no. of slices>
||pre: nat(n) & n>0
||post: (integrate f start finish n) is an estimate of the
||      integral of f from start to finish
```

This function is higher-order because it takes a function as one of its arguments. The following is a definition of `integrate`:

```
integrate f start finish n
  = sum (map area [1..n])
    where
      width  = (finish - start) / n
      area i = width *  f(start + width * (i-0.5))
```

8.3 The higher-order function `fold`

Consider the following function again:

```
sum :: [num] -> num
||pre:  none
||post: sum xs = xs!0 + ... + xs!(#xs-1)
```

In other words, `sum xs` adds together the elements of `xs`:

```
sum [] = 0
sum (x:xs) = x + (sum xs)
```

You can imagine an exactly similar function for finding the product of the elements, replacing `+` by `*`. You also have to replace the base case result 0 by 1 — otherwise you obtain the wrong answer for singleton lists, and so by the recursion for longer lists.

These are so similar that you could imagine both specification and definition being constructed automatically once you have supplied the operator (`+` or `*` or other possibilities) and the base case result (0 or 1). Higher-order functions allow you to do just that. We shall write `fold` f e for the function that 'folds together' the elements of a list using the operator `f` and base case result e.

The infix notation is very convenient, so in what follows we shall often use the Miranda convention that if `f` is a 2-argument function then `$f` is the same function treated notationally as an infix operator. For example, x `$gcd` y is the same as `gcd` x y.

Let us first look at the type of `fold`. It has three arguments, namely the function `f`, e for the base case and the list xs. We do not care what list type xs has. It is `[*]` for some type `*`, and then `f` must have matching types `* -> * -> *` and e must have the type `*`. (For `sum`, `*` was `num`.)

```
fold :: (* -> * -> *) -> * -> [*] -> *
```

For the post-condition, we require

```
||post: fold f e xs = xs!0 $f ... $f xs!(#xs-1)
```

This is a little imprecise. It does not make it at all clear what should happen when xs is empty, and the '...' is slightly fuzzy. We will look at these issues more closely later. For the moment, what is more important is that certain pre-conditions are implied.

First, we wrote `xs!0 $f ... $f xs!(#xs-1)` without any parentheses to show the evaluation order of the different `$f`s. We could have chosen an evaluation order and put parentheses in, for instance

```
(...(xs!0 $f xs!1)... $f xs!(#xs-1))
```

or

```
(xs!0 $f ... (xs!(#xs-2) $f xs!(#xs-1))...)
```

But rather than make such a choice, let us keep to the simple case where, as with `+` and `*`, parentheses are unnecessary.

A particular case of this is when operating on three elements: we require

$$\forall x, y, z.\ x\ \texttt{\$f}\ (y\ \texttt{\$f}\ z) = (x\ \texttt{\$f}\ \ y)\ \texttt{\$f}\ z$$

In fact, this particular case (the 'associativity' law) is enough to show also that parentheses are unnecessary in longer expressions — we mentioned this with `++` in Chapter 6.

Here, then, is one pre-condition: `$f` must be associative.

The other pre-condition concerns the interaction between `$f` and e. The key properties (they will appear at various points of the reasoning) of 0 and 1 in relation to `+` and `*` are that they are 'identities': $x + 0 = x$, $x * 1 = x$. We shall assume a general identity law for e:

$$\forall x.\ x\ \texttt{\$f}\ e = x = e\ \texttt{\$f}\ x$$

Finally, let us try to improve the post-condition by removing the dots. We shall use the same trick as we did with reverse, namely to give strong and useful properties (not, strictly speaking, a post-condition) of the way `fold` f e works, trying to relate it to `++`:

```
fold :: (* -> * -> *) -> * -> [*] -> *
||pre:  (A) x,y,z:*. x $f (y $f z)
||                    = (x $f y) $f z   ($f is associative)
||      & (A) x:*. x $f e = x = e $f x  (e is an identity for $f)
||post: fold f e [] = e
||      & (A) x:*. fold f e [x] = x
||      & (A) xs,ys:[*]. fold f e (xs++ys)
||                    = (fold f e xs) $f (fold f e ys)
```

Let us note straight away that the specification specifies `fold` uniquely. In other words, if $f1$ and $f2$ both satisfy the specification, `$f` is associative, e is an identity for `$f` and xs is a (finite!) list, then

$$f1\ f\ e\ xs = f2\ f\ e\ xs$$

This is easily proved by induction on xs, the induction step coming from

$$f1\ f\ e\ (x : xs) = (f1\ f\ e\ [x])\ \texttt{\$f}\ (f1\ f\ e\ xs) = x\ \texttt{\$f}\ (f1\ f\ e\ xs)$$

8.4 Applications

We shall implement `fold` later. For the moment, let us look at some applications. `sum` can be defined as `fold (+) 0`. (Notice how a built-in infix operator can be passed as an argument to a higher-order function by placing it in parentheses.) Once you have checked that `+` is associative (that is, x`+`$(y$`+`$z) = (x$`+`$y)$`+`z) and 0 is an identity (x`+0` $= x =$ `0+`x), then you know immediately that `sum` $(xs$`++`$ys)$ = (`sum` xs)`+`(`sum` ys). You do not need to prove it by induction; the induction will be done once and for all when we implement fold and show that the specification is satisfied.

The analogous function product can be defined as `fold (*) 1`. Note that subtraction and division are not associative, and it is less obvious what one would mean by 'the elements of a list folded together by subtraction'. The function `concat` is defined as `fold (++) []`. It takes a list of lists and appends (or concatenates) them all together.

By combining `fold` and `map`, quite a wide range of functions can be defined. For instance, `count` of Exercise 7 in Chapter 6 can be defined by

```
count x xs = fold (+) 0 (map f xs)
             where f y = 1,   if y = x
                       = 0.   otherwise
```

Then we can prove the properties of `count` without using induction. For instance,

$$\begin{aligned}\texttt{count}\ x\ (xs\texttt{++}ys) &= \texttt{fold}\ (\texttt{+})\ 0\ (\texttt{map}\ f\ (xs\texttt{++}ys))\\ &= \texttt{fold}\ (\texttt{+})\ 0\ (\texttt{map}\ f\ xs)\texttt{++}(\texttt{map}\ f\ ys)\\ &= \texttt{fold}\ (\texttt{+})\ 0\ (\texttt{map}\ f\ xs)\texttt{+}\texttt{fold}\ (\texttt{+})\ 0\ (\texttt{map}\ f\ ys)\\ &= (\texttt{count}\ x\ xs)\texttt{+}(\texttt{count}\ x\ ys)\end{aligned}$$

8.5 Implementing `fold` — `foldr`

There are two common implementations of `fold`. They have different names, `foldr` and `foldl`, and this is because they can also be used when `$f` and e do not satisfy the pre-conditions of `fold`, but they give different answers — actually, they correspond to different bracketings. (In fact, they even have more general types than `fold`, as you can see if you ask the Miranda system what it thinks their types are.) `foldr` and `foldl` are rather different. We shall show `foldr` here — it uses the same idea as `sum` — and leave the discussion of `foldl` to Exercise 6. `foldr f e xs` calculates

$$(xs!0\ \texttt{\$f}\ldots\texttt{\$f}\ (xs!(\#xs-1)\ \texttt{\$f}\ e)\ldots)$$

```
foldr f e [] = e
foldr f e (x:xs) = x $f (foldr f e xs)
```

Proposition 8.1 `foldr` satisfies its specification. We fix an associative operator `$f` with an identity e, and prove the three equations of the post-condition. The first is immediate and the second is easy. For the third we use induction on xs to prove $\forall xs : \texttt{[}*\texttt{]}.\ P(xs)$, where

$$P(xs) \stackrel{\text{def}}{=} \forall ys : \texttt{[}*\texttt{]}\ \texttt{foldr}\ f\ e\ (xs\texttt{++}ys) = (\texttt{foldr}\ f\ e\ xs)\ \texttt{\$f}\ (\texttt{foldr}\ f\ e\ ys)$$

base case: $P(\texttt{[]})$

$$\begin{aligned}\text{LHS} &= \texttt{foldr}\ f\ e\ (\texttt{[]++}ys)\\ &= \texttt{foldr}\ f\ e\ ys\\ &= e\ \texttt{\$f(foldr}\ f\ e\ ys) && \text{(identity law)}\\ &= (\texttt{foldr}\ f\ e\ \texttt{[]})\ \texttt{\$f}\ (\texttt{foldr}\ f\ e\ ys) = \text{RHS}\end{aligned}$$

induction step: assume $P(xs)$ and prove $P(x\texttt{:}xs)$:

$$\begin{aligned}\text{LHS} &= \texttt{foldr}\ f\ e\ (x\texttt{:}xs\texttt{++}ys) = x\ \texttt{\$f}\ (\texttt{foldr}\ f\ e\ (xs\texttt{++}ys))\\ &= x\ \texttt{\$f}\ (\texttt{foldr}\ f\ e\ xs)\ \texttt{\$f}\ (\texttt{foldr}\ f\ e\ ys)\\ &= (\texttt{foldr}\ f\ e\ (x\texttt{:}xs))\ \texttt{\$f}\ (\texttt{foldr}\ f\ e\ ys) = \text{RHS}\end{aligned}$$

□

Although the reasoning is more complicated, `foldr` can also be used in more general cases (for example, non-associative); note also that the definition of `foldr` has a more liberal type than `fold`:

```
foldr :: (** -> * -> *) -> * -> [**] -> *
length x  = foldr fun 0 x   where fun a acc = 1 + acc
```

Notice that built-in infix operators can be passed as arguments to higher-order functions by placing them in parentheses. Recall the function for building an ordered tree from a list:

```
build        ::  [*] -> (tree *)
build []      =  Emptytree
build (x:xs)  =  insertT x (build xs)
```

A more concise and preferred definition uses `fold`:

```
build x = foldr insertT Emptytree x
```

The evaluation sequence for an application of the new definition of `build` illustrates the reduction sequence:

```
build[6,2,4]
= foldr insertT Emptytree [ 6, 2, 4 ]
= insertT 6 (insertT 2 (insertT 4 Emptytree) )
= Node (Node Emptytree 2 Emptytree) 4 (Node Emptytree 6 Emptytree)
```

8.6 Summary

- Most list-processing functions can be described using higher-order functions such as `map` and `fold` (which capture the two most common patterns of recursion over lists). The same approach can also be applied to other patterns of recursion and for user-defined types.
- A small suite of higher-order functions to iterate over each data type can be used to avoid writing many explicit recursive functions on that

type. Then an appropriately parameterized higher-order function is used to define the required function.

- The technique can be compared with polymorphism where structures (including functions, of course) of similar shape are described by a single polymorphic definition. Higher-order functions are used to describe other recursive functions with the same overall structure.
- The functional programming 'style' is to use higher-order functions since they lead to concise and abstract programs.
- It is usually easier to understand programs that avoid excessive use of explicit recursion and to use library and higher-order functions whenever possible.
- Induction proofs can be done once and for all on the higher-order functions.

8.7 Exercises

1. Prove that `integrate` terminates, assuming that the supplied function terminates.
2. Define a function `sigma`, which, given a function, say `f`, and two integers corresponding to the lower and the upper limits of a range of integers, say `n` and `m`, will capture the common mathematical notation of

   $$\sum_{x=n}^{m} fx$$

3. In the imperative programming language C there is a library function called `ctoi` which converts a string to an integer. For example, `ctoi "123"` gives `123`. Declare and define `ctoi` in Miranda. Ensure that your definition is not recursive.
4. Give type declarations and definitions of functions `curry` and `uncurry`, for example, `uncurry` f $(x,y) = f\ x\ y$.
5. This question is about writing a function to sort lists using what is called a *merging* algorithm:
   (a) Recall `smerge`, which, given two sorted lists, merged them into a single sorted list. Show that `smerge` is associative and `[]` is an identity for it.
   (b) Write a function `mergesort` which sorts a list by converting it to a list of singletons and then applying `fold` *smerge* `[]`.
6. The other implementation of `fold` is `foldl`, which calculates

   $(\ldots(e$ `$f` $xs!0)$ `$f` $\ldots$ `$f` $xs!(\#xs-1))$

```
foldl f a [] = a
foldl f a (x:xs) = foldl f (a $f x) xs
```

Note that we have replaced e by a. This is because the parameter is passed through the recursive calls of `foldl`, so even if it starts off as an identity for `$f` it will not remain as `$f`'s identity. In general, still assuming that `$f` is associative and e is an identity for it, `foldl` $f\ a\ xs = a$ `$f` (`foldl` $f\ e\ xs$). This can be proved easily by induction on xs; but since we would still need another induction to prove the equations of the specification, it is possible to combine both induction proofs.

(a) Use induction on xs to prove that

$$\forall a :: *.\ \forall xs, ys :: [*].\ \texttt{foldl}\ f\ a\ (xs \texttt{++} ys)$$
$$= \texttt{foldl}\ f\ (a\ \texttt{\$f}\ (\texttt{foldl}\ f\ e\ xs))\ ys$$

(HINT: In the induction step you use the induction hypothesis *twice*, with different values substituted for a and ys. The unexpected one has $ys = []$. To avoid confusion, introduce new constants for your $\forall$-introductions.)

(b) Deduce from (a) that

$$\forall a :: *.\ \forall xs :: [*].\ \texttt{foldl}\ f\ a\ xs\ =\ a\ \texttt{\$f}\ (\texttt{foldl}\ f\ e\ xs)$$

(c) Deduce from (a) and (b) that

$$\forall xs, ys :: [*].\ \texttt{foldl}\ f\ e\ (xs \texttt{++} ys)$$
$$= (\texttt{foldl}\ f\ e\ xs)\ \texttt{\$f}\ (\texttt{foldl}\ f\ e\ ys)$$

and hence that `foldl` implements the specification for `fold`.

(d) Deduce that

$$\texttt{foldr}\ f\ e\ xs\ =\ \texttt{foldl}\ f\ e\ xs$$

provided that `$f` is associative, e is an identity for it (and xs is finite).

(e) Give examples to show that `foldr` and `foldl` can compute different results is `$f` is not associative or e is not an identity for it.

7. Consider the following specification:

```
filter :: (*->bool)->[*]->[*]
||filter p xs is the list xs except that the
||elements x for which p x is False have all been removed.
||pre:  none
||post: (A)x:*. (Isin(x,ys) -> p x)
||      & (E)ws:[*]. (Merge(ys,ws,xs)
||      & (A)x:*. (Isin(x,ws) -> ~(p x)))
||              where ys = filter p xs
||      (ws contains the elements that were filtered out)
```