

5

Functions and Infinite Data

The most powerful techniques of functional programming are those that treat functions as data. Most functional languages give function values full rights, free of arbitrary restrictions. Like other values, functions may be arguments and results of other functions and may belong to pairs, lists and trees.

Procedural languages like Fortran and Pascal accept this idea as far as is convenient for the compiler writer. Functions may be arguments: say, the comparison to be used in sorting or a numerical function to be integrated. Even this restricted case is important.

A function is *higher-order* (or a *functional*) if it operates on other functions. For instance, the functional *map* applies a function to every element of a list, creating a new list. A sufficiently rich collection of functionals can express all functions without using variables. Functionals can be designed to construct parsers (see Chapter 9) and theorem proving strategies (see Chapter 10).

Infinite lists, whose elements are evaluated upon demand, can be implemented using functions as data. The tail of a lazy list is a function that, if called, produces another lazy list. A lazy list can be infinitely long and any finite number of its elements can be evaluated.

Chapter outline

The first half presents the essential programming techniques involving functions as data. The second half serves as an extended, practical example. Lazy lists can be represented in ML (despite its strict evaluation rule) by means of function values.

The chapter contains the following sections:

Functions as values. The `fn` notation can express a function without giving it a name. Any function of two arguments can be expressed as a ‘curried’ function of one argument, whose result is another function. Simple examples of higher-order functions include polymorphic sorting functions and numerical operators.

General-purpose functionals. Higher-order functional programming largely

consists of using certain well-known functionals, which operate on lists or other recursive datatypes.

Sequences, or infinite lists. The basic mechanism for obtaining laziness in ML is demonstrated using standard examples. A harder problem is to combine a list of lists of integers into a single list of integers — if the input lists are infinite, they must be combined fairly such that no integers are lost.

Search strategies and infinite lists. The possibly infinite set of solutions to a search problem can be generated as a lazy list. The consumer of the solutions can be designed independently of the producer, which may employ any suitable search strategy.

Functions as values

Functions in ML are abstract values: they can be created; they can be applied to an argument; they can belong to other data structures. Nothing else is allowed. A function is given by patterns and expressions but taken as a ‘black box’ that transforms arguments to results.

5.1 Anonymous functions with *fn* notation

An ML function need not have a name. If x is a variable (of type σ) and E is an expression (of type τ) then the expression

$$\text{fn } x \Rightarrow E$$

denotes a function of type $\sigma \rightarrow \tau$. Its argument is x and its body is E . Pattern-matching is allowed: the expression

$$\text{fn } P_1 \Rightarrow E_1 \mid \dots \mid P_n \Rightarrow E_n$$

denotes the function defined by the patterns P_1, \dots, P_n . It has the same meaning as the `let` expression

$$\text{let fun } f(P_1) = E_1 \mid \dots \mid f(P_n) = E_n \text{ in } f \text{ end}$$

provided f does not appear in the expressions E_1, \dots, E_n . The `fn` syntax cannot express recursion.

For example, `fn n=>n*2` is a function that doubles an integer. It can be applied to an argument; it can be given a name by a `val` declaration.

```
(fn n=>n*2) (9);
> 18 : int
val double = fn n=>n*2;
```

```
> val double = fn : int -> int
```

Many ML constructs are defined in terms of the `fn` notation. The conditional expression

```
if E then E1 else E2
```

abbreviates the function application

```
(fn true => E1 | false => E2) (E)
```

The `case` expression is translated similarly.

Exercise 5.1 Express these functions using `fn` notation.

```
fun square (x) : real = x*x;
fun cons (x, y) = x::y;
fun null [] = true
  | null (_::_) = false;
```

Exercise 5.2 Modify these function declarations to use `val` instead of `fun`:

```
fun area (r) = pi*r*r;
fun title(name) = "The Duke of " ^ name;
fun lengthvec (x, y) = Math.sqrt(x*x + y*y);
```

5.2 Curried functions

A function can have only one argument. Hitherto, functions with multiple arguments have taken them as a tuple. Multiple arguments can also be realized by a function that returns another function as its result. This device is called **currying** after the logician H. B. Curry.¹ Consider the function

```
fun prefix pre =
  let fun cat post = pre ^ post
      in cat end;
> val prefix = fn : string -> (string -> string)
```

Using `fn` notation, `prefix` is the function

```
fn pre => (fn post => pre ^ post)
```

Given a string `pre`, the result of `prefix` is a function that concatenates `pre` to the front of its argument. For instance, `prefix "Sir "` is the function

¹ It has been credited to Schönfinkel, but *Schönfinkeling* has never caught on.

```
fn post => "Sir " ^ post
```

It may be applied to a string:

```
prefix "Sir ";
> fn : string -> string
it "James Tyrrell";
> "Sir James Tyrrell" : string
```

Dispensing with *it*, both function applications may be done at once:

```
(prefix "Sir ") "James Tyrrell";
> "Sir James Tyrrell" : string
```

This is a function call where the function is computed by an expression, namely `prefix "Sir "`.

Note that *prefix* behaves like a function of two arguments. It is a **curried function**. We now have two ways of declaring a function with arguments of types σ_1 and σ_2 and result of type τ . A function over pairs has type $(\sigma_1 \times \sigma_2) \rightarrow \tau$. A curried function has type $\sigma_1 \rightarrow (\sigma_2 \rightarrow \tau)$.

A curried function permits **partial application**. Applied to its first argument (of type σ_1) its result is a function of type $\sigma_2 \rightarrow \tau$. This function may have a general use: say, for addressing Knights.

```
val knightify = prefix "Sir ";
> val knightify = fn : string -> string
knightify "William Catesby";
> "Sir William Catesby" : string
knightify "Richard Ratcliff";
> "Sir Richard Ratcliff" : string
```

Other illustrious personages can be addressed similarly:

```
val dukify = prefix "The Duke of ";
> val dukify = fn : string -> string
dukify "Clarence";
> "The Duke of Clarence" : string
val lordify = prefix "Lord ";
> val lordify = fn : string -> string
lordify "Stanley";
> "Lord Stanley" : string
```

Syntax for curried functions. The functions above are declared by `val`, not `fun`. A `fun` declaration must have explicit arguments. There may be several arguments, separated by spaces, for a curried function. Here is an equivalent declaration of *prefix*:

```
fun prefix pre post = pre ^ post;
> val prefix = fn : string -> (string -> string)
```

A function call has the form $E E_1$, where E is an expression that denotes a function. Since

$$E E_1 E_2 \dots E_n \text{ abbreviates } (\dots((E E_1) E_2) \dots) E_n$$

we may write `prefix "Sir " "James Tyrrell"` without parentheses. The expressions are evaluated from left to right.

The type of `prefix`, namely $string \rightarrow (string \rightarrow string)$, may be written without parentheses: the symbol \rightarrow associates to the right.

Recursion. Curried functions may be recursive. Calling `replist n x` makes the list consisting of n copies of x :

```
fun replist n x = if n=0 then [] else x :: replist (n-1) x;
> val replist = fn : int -> 'a -> 'a list
replist 3 true;
> [true, true, true] : bool list
```

Recursion works by the usual evaluation rules, even with currying. The result of `replist 3` is the function

```
fn x => if 3 = 0 then [] else x :: replist (3 - 1) x
```

Applying this to `true` produces the expression

$$true :: replist\ 2\ true$$

As evaluation continues, two further recursive calls yield

$$true :: true :: true :: replist\ 0\ true$$

The final call returns `nil` and the overall result is `[true, true, true]`.



An analogy with arrays. The choice between pairing and currying is analogous to the choice, in Pascal, between a 2-dimensional array and nested arrays.

```
A: array [1..20, 1..30] of integer
B: array [1..20] of array [1..30] of integer
```

The former array is subscripted $A[i, j]$, the latter as $B[i][j]$. Nested arrays permit partial subscripting: $B[i]$ is a 1-dimensional array.

Exercise 5.3 What functions result from partial application of the following curried functions? (Do not try them at the machine.)

```
fun plus i j : int = i+j;
fun lesser a b : real = if a<b then a else b;
fun pair x y = (x,y);
fun equals x y = (x=y);
```

Exercise 5.4 Is there any practical difference between the following two declarations of the function f ? Assume that the function g and the curried function h are given.

```
fun f x y = h (g x) y;
fun f x = h (g x);
```

5.3 Functions in data structures

Functions and concrete datatypes play complementary rôles in a data structure. Lists and trees provide the outer framework and organize the information, while functions hold potential computations. Although functions are represented by finite programs in the computer, we can often treat them as infinite objects.

Pairs and lists may contain functions as their components:²

```
(concat, Math.sin);
> (fn, fn) : (string list -> string) * (real -> real)
[op+, op-, op*, op div, op mod, Int.max, Int.min];
> [fn, fn, fn, fn, fn] : (int * int -> int) list
```

Functions stored in a data structure can be extracted and applied.

```
val titlefns = [dukify, lordify, knightify];
> val titlefns = [fn, fn, fn] : (string -> string) list
hd titlefns "Gloucester";
> "The Duke of Gloucester" : string
```

This is a curried function call: `hd titlefns` returns the function `dukify`. The polymorphic function `hd` has, in this example, the type

$$(string \rightarrow string)list \rightarrow (string \rightarrow string).$$

A binary search tree containing functions might be useful in a desk calculator program. The functions are addressed by name.

² Recall that the keyword `op` yields the value of an infix operator, as a function.

```

val funtree = Dict.insert (Dict.insert (Dict.insert (Lf, "sin", Math.sin),
                                         "cos", Math.cos),
                          "atan", Math.atan);
> val funtree =
> Br (("sin", fn),
>     Br ("cos", fn), Br ("atan", fn), Lf, Lf), Lf),
>     Lf) : (real -> real) Dict.t
Dict.lookup (funtree, "cos") 0.0;
> 1.0 : real

```

The functions stored in the tree must have the same type, here $real \rightarrow real$. Although different types can be combined into one datatype, this can be inconvenient. As mentioned at the end of Section 4.6, type exn can be regarded as including all types. A more flexible type for the functions is $exn\ list \rightarrow exn$.

Exercise 5.5 What type does the polymorphic function `Dict.lookup` have in the example above?

5.4 Functions as arguments and results

The sorting functions of Chapter 3 are coded to sort real numbers. They can be generalized to an arbitrary ordered type by passing the ordering predicate (\leq) as an argument. Here is a polymorphic function for insertion sort:

```

fun insort lessequal =
  let fun ins (x, []) = [x]
      | ins (x, y::ys) =
          if lessequal(x, y) then x::y::ys
          else y :: ins (x, ys)
      fun sort [] = []
      | sort (x::xs) = ins (x, sort xs)
  in sort end;
> val insort = fn
> : ('a * 'a -> bool) -> 'a list -> 'a list

```

Functions `ins` and `sort` are declared locally, referring to `lessequal`. Though it may not be obvious, `insort` is a curried function. Given an argument of type $\tau \times \tau \rightarrow bool$ it returns the function `sort`, which has type $\tau\ list \rightarrow \tau\ list$. The types of the ordering and the list elements must agree.

Integers can now be sorted. (Although the operator `<=` is overloaded, its type is constrained by the list of integers.)

```

insort (op<=) [5,3,7,5,9,8];
> [3, 5, 5, 7, 8, 9] : int list

```

Passing the relation \geq for `lessequal` gives a decreasing sort:

```

insort (op>=) [5,3,7,5,9,8];
> [9, 8, 7, 5, 5, 3] : int list

```

Pairs of strings can be sorted using lexicographic ordering:

```

fun leq_stringpair ((a,b), (c,d) : string*string) =
  a<c orelse (a=c andalso b<=d);
> val leq_stringpair = fn
> : (string * string) * (string * string) -> bool

```

We sort a list of (family name, forename) pairs:

```

insort leq_stringpair
  [("Herbert","Walter"),      ("Plantagenet","Richard"),
   ("Plantagenet","Edward"), ("Brandon","William"),
   ("Tyrrell","James"),      ("Herbert","John") ];
> [("Brandon", "William"), ("Herbert", "John"),
>  ("Herbert", "Walter"), ("Plantagenet", "Edward"),
>  ("Plantagenet", "Richard"), ("Tyrrell", "James")]
> : (string * string) list

```

Functions are frequently passed as arguments in numerical computing. The following functional computes the summation $\sum_{i=0}^{m-1} f(i)$. For efficiency, it uses an iterative function that refers to the arguments f and m :

```

fun summation f m =
  let fun sum (i,z) : real =
        if i=m then z else sum (i+1, z+(f i))
      in sum(0, 0.0) end;
> val summation = fn : (int -> real) -> int -> real

```

The `fn` notation works well with functionals. Here it eliminates the need to declare a squaring function prior to computing the sum $\sum_{k=0}^9 k^2$:

```

summation (fn k => real(k*k)) 10;
> 285.0 : real

```

The double sum $\sum_{i=0}^{m-1} \sum_{j=0}^{n-1} g(i, j)$ is computed by

```

summation (fn i => summation (fn j => g(i,j)) n) m;

```

This serves as a translation of the \sum -notation into ML; the index variables i and j are bound by `fn`. The inner summation, $\sum_{j=0}^{n-1} g(i, j)$, is a function of i . The function over j is the partial application of g to i .

The partial application can be simplified by summing over a curried function h instead of g . The double sum $\sum_{i=0}^{m-1} \sum_{j=0}^{n-1} h i j$ is computed by


```
summation (fn i => summation (h i) n) m;
```

Observe that $\text{summation } f$ has the same type as f , namely $\text{int} \rightarrow \text{real}$, and that $\sum_{i=0}^{m-1} \sum_{j=0}^{i-1} f(j)$ may be computed by $\text{summation } (\text{summation } f) m$.



Polymorphic val declarations. Because a function can be the result of a computation, you might expect declarations such as the following to be legal:

```
val list5 = replist 5;
> val list5 = fn : 'a -> 'a list
val f = hd [hd];
> val f = fn : 'a list -> 'a
```

They were indeed legal in earlier versions of ML, but now trigger a message such as ‘Non-value in polymorphic declaration.’ This restriction has to do with references; Section 8.3 explains the details. Changing the function declaration $\text{val } f = E$ to

```
fun f x = E x
```

renders it acceptable. The change is harmless unless computing E has side-effects, or is expensive.

The restriction affects all polymorphic `val` declarations, not just those of functions. Recall that typing E at top level abbreviates typing $\text{val } it = E$. For instance, we may not type $\text{hd } [[]]$.

Exercise 5.6 Write a polymorphic function for top-down merge sort, passing the ordering predicate (\leq) as an argument.

Exercise 5.7 Write a functional to compute the minimum value $\min_{i=0}^{m-1} f(i)$ of a function f , where m is any given positive integer. Use the functional to express the two-dimensional minimum $\min_{i=0}^{m-1} \min_{j=0}^{n-1} g(i, j)$, for positive integers m and n .

General-purpose functionals

Functional programmers often use higher-order functions to express programs clearly and concisely. Functionals to process lists have been popular since the early days of Lisp, appearing in infinite variety and under many names. They express operations that otherwise would require separate recursive function declarations. Similar recursive functionals can be defined for trees.

A comprehensive set of functionals provides an abstract language for expressing other functions. After reading this section, you may find it instructive to review previous chapters and simplify the function definitions using functionals.

5.5 Sections

Imagine applying an infix operator to only one operand, either left or right, leaving the other operand unspecified. This defines a function of one argument, called a *section*. Here are some examples in the notation of Bird and Wadler (1988):

- ("Sir ") is the function *knightify*
- (/2.0) is the function 'divide by 2'

Sections can be added to ML (rather crudely) by the functionals *secl* and *secr*:

```
fun secl x f y = f(x, y);
> val secl = fn : 'a -> ('a * 'b -> 'c) -> 'b -> 'c
fun secr f y x = f(x, y);
> val secr = fn : ('a * 'b -> 'c) -> 'b -> 'a -> 'c
```

These functionals are typically used with infix functions and `op`, but may be applied to any function of suitable type. Here are some left sections:

```
val knightify = (secl "Sir " op^);
> val knightify = fn : string -> string
knightify "Geoffrey";
> "Sir Geoffrey" : string
val recip = (secl 1.0 op/);
> val recip = fn : real -> real
recip 5.0;
> 0.2 : real
```

Here is a right section for division by 2:

```
val halve = (secr op/ 2.0);
> val halve = fn : real -> real
halve 7.0;
> 3.5 : real
```

Exercise 5.8 Is there any similarity between sections and curried functions?

Exercise 5.9 What functions do the following sections yield? Recall that *take* removes elements from the head of a list (Section 3.4) while *inter* forms the intersection of two lists (Section 3.15).

```
secl op@ ["Richard"]
secl ["heed", "of", "yonder", "dog!"] List.take
secr List.take 3
secl ["his", "venom", "tooth"] inter
```

5.6 Combinators

The theory of the λ -calculus is in part concerned with expressions known as *combinators*. Many combinators can be coded in ML as higher-order functions, and have practical applications.

Composition. The infix *o* (yes, the letter ‘o’) denotes function composition. The standard library declares it as follows:

```
infix o;
fun (f o g) x = f (g x);
> val o = fn : ('b -> 'c) * ('a -> 'b) -> 'a -> 'c
```

Composition is familiar to mathematicians; $f \circ g$ is the function that applies g , then f , to its argument. Composition can express many functions, especially using sections. For instance, the functions

```
fn x => ~ (Math.sqrt x)
fn a => "beginning" ^ a ^ "end"
fn x => 2.0 / (x-1.0)
```

can be expressed without mentioning their argument:

```
~ o Math.sqrt
(secl "beginning" op^) o (secl op^ "end")
(secl 2.0 op/) o (secl op- 1.0)
```

To compute the sum $\sum_{k=0}^9 \sqrt{k}$, the functions *Math.sqrt* and *real* (which converts integers to reals) are composed. Composition is more readable than *fn* notation:

```
summation (Math.sqrt o real) 10;
```

The combinators S, K and I. The identity combinator, *I*, simply returns its argument:

```
fun I x = x;
> val I = fn : 'a -> 'a
```

Composition of a function with *I* has no effect:

```
knightify o I o (prefix "William ") o I;
> fn : string -> string
it "Catesby";
> "Sir William Catesby" : string
```

The combinator *K* makes constant functions. Given x it makes the function that always returns x :

```
fun K x y = x;
> val K = fn : 'a -> 'b -> 'a
```

For a contrived demonstration of constant functions, let us compute the product $m \times z$ by the repeated addition $\sum_{i=0}^{m-1} z$:

```
summation (K 7.0) 5;
> 35.0 : real
```

The combinator S is a general form of composition:

```
fun S x y z = x z (y z);
> val S = fn : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c
```

Every function in the λ -calculus can be expressed using just S and K — with no variables! David Turner (1979) has exploited this celebrated fact to obtain lazy evaluation: since no variables are involved, no mechanism is required for binding their values. Virtually all lazy functional compilers employ some refinement of this technique.

Here is a remarkable example of the expressiveness of S and K . The identity function I can be defined as $S K K$:

```
S K K 17;
> 17 : int
```

Exercise 5.10 Write the computation steps of $S K K 17$.

Exercise 5.11 Suppose we are given an expression E consisting of infix operators, constants and variables, with one occurrence of the variable x . Describe a method for expressing the function $\text{fn } x \Rightarrow E$ using I , sections and composition instead of fn .

5.7 The list functionals *map* and *filter*

The functional *map* applies a function to every element of a list, returning a list of the function's results:

$$\text{map } f [x_1, \dots, x_n] = [f x_1, \dots, f x_n]$$

The ML library declares *map* as follows:

```

fun map f [] = []
  | map f (x::xs) = (f x) :: map f xs;
> val map = fn : ('a -> 'b) -> 'a list -> 'b list
map recip [0.1, 1.0, 5.0, 10.0];
> [10.0, 1.0, 0.2, 0.1] : real list
map size ["York", "Clarence", "Gloucester"];
> [4, 8, 10] : int list

```

The functional *filter* applies a predicate — a boolean-valued function — to a list. It returns a list of all elements satisfying the predicate, in their original order.

```

fun filter pred [] = []
  | filter pred (x::xs) =
      if pred x then x :: filter pred xs
      else filter pred xs;
> val filter = fn : ('a -> bool) -> 'a list -> 'a list
filter (fn a => size a = 4)
  ["Hie", "thee", "to", "Hell", "thou", "cacodemon"];
> ["thee", "Hell", "thou"] : string list

```

Pattern-matching in curried functions works exactly as if the arguments were given as a tuple. Both functionals are curried: *map* takes a function of type $\sigma \rightarrow \tau$ to one of type $\sigma \text{ list} \rightarrow \tau \text{ list}$, while *filter* takes a function of type $\tau \rightarrow \text{bool}$ to one of type $\tau \text{ list} \rightarrow \tau \text{ list}$.

Thanks to currying, these functionals work together for lists of lists. Observe that *map(map f)[l_1, l_2, \dots, l_n]* applies *map f* to each list l_1, l_2, \dots .

```

map (map double) [[1], [2,3], [4,5,6]];
> [[2], [4, 6], [8, 10, 12]] : int list list
map (map (implode o rev o explode))
  ["When", "he", "shall", "split"],
  ["thy", "very", "heart", "with", "sorrow"]];
> [[ "nehW", "eh", "llahs", "tilps"],
  ["yht", "yrev", "traeh", "htiw", "worros"]]
> : string list list

```

Similarly, *map(filter pred)[l_1, l_2, \dots, l_n]* applies *filter pred* to each of the lists l_1, l_2, \dots . It returns a list of lists of elements satisfying the predicate *pred*.

```

map (filter (secr op< "m"))
  ["my", "hair", "doth", "stand", "on", "end"],
  ["to", "hear", "her", "curses"]];
> [[ "hair", "doth", "end"], ["hear", "her", "curses"]]
> : string list list

```

Many list functions can be coded trivially using *map* and *filter*. Our matrix transpose function (Section 3.9) becomes

```

fun transp ([]:.-) = []
  | transp rows = map hd rows :: transp (map tl rows);
> val transp = fn : 'a list list -> 'a list list
transp [["have", "done", "thy", "charm"],
        ["thou", "hateful", "withered", "hag!"]];
> [["have", "thou"], ["done", "hateful"],
   ["thy", "withered"], ["charm", "hag!"]]
> : string list list

```

Recall how we defined the intersection of two ‘sets’ in terms of the membership relation, in Section 3.15. That declaration can be reduced to a single line:

```

fun inter(xs, ys) = filter (secc (op mem) ys) xs;
> val inter = fn : 'a list * 'a list -> 'a list

```

Exercise 5.12 Show how to replace any expression of the form

$$\text{map } f (\text{map } g \text{ } xs),$$

by an equivalent expression that calls *map* only once.

Exercise 5.13 Declare the infix operator *andf* such that

$$\text{filter } (\text{pred1 andf pred2}) \text{ } xs$$

returns the same value as *filter pred1 (filter pred2 xs)*.

5.8 The list functionals *takewhile* and *dropwhile*

These functionals chop an initial segment from a list using a predicate:

$$\underbrace{[x_0, \dots, x_{i-1}]}_{\text{takewhile}} \underbrace{[x_i, \dots, x_{n-1}]}_{\text{dropwhile}}$$

The initial segment, which consists of elements satisfying the predicate, is returned by *takewhile*:

```

fun takewhile pred [] = []
  | takewhile pred (x::xs) =
    if pred x then x :: takewhile pred xs
    else [];
> val takewhile = fn : ('a -> bool) -> 'a list -> 'a list

```

The remaining elements (if any) begin with the first one to falsify the predicate. This list is returned by *dropwhile*:

```

fun dropwhile pred [] = []
  | dropwhile pred (x::xs) =
      if pred x then dropwhile pred xs
      else x::xs;
> val dropwhile = fn : ('a -> bool) -> 'a list -> 'a list

```

These two functionals can process text in the form of character lists. The predicate `Char.isAlpha` recognizes letters. Given this predicate, `takewhile` returns the first word from a sentence and `dropwhile` returns the remaining characters.

```

takewhile Char.isAlpha (explode "that deadly eye of thine");
> [#"t", #"h", #"a", #"t"] : char list
dropwhile Char.isAlpha (explode "that deadly eye of thine");
> [#" ", #"d", #"e", #"a", #"d", #"l", ...] : char list

```

Since they are curried, `takewhile` and `dropwhile` combine with other functionals. For instance, `map(takewhile pred)` returns a list of initial segments.

5.9 The list functionals exists and all

These functionals report whether some (or every) element of a list satisfies some predicate. They can be viewed as quantifiers over lists:

```

fun exists pred [] = false
  | exists pred (x::xs) = (pred x) orelse exists pred xs;
> val exists = fn : ('a -> bool) -> 'a list -> bool

fun all pred [] = true
  | all pred (x::xs) = (pred x) andalso all pred xs;
> val all = fn : ('a -> bool) -> 'a list -> bool

```

By currying, these functionals convert a predicate over type τ to a predicate over type τ list. The membership test `x mem xs` can be expressed in one line:

```

fun x mem xs = exists (secr op= x) xs;
> val mem = fn : 'a * 'a list -> bool

```

The function `disjoint` tests whether two lists have no elements in common:

```

fun disjoint (xs, ys) = all (fn x => all (fn y => x<>y) ys) xs;
> val disjoint = fn : 'a list * 'a list -> bool

```

Because of their argument order, `exists` and `all` are hard to read as quantifiers when nested; it is hard to see that `disjoint` tests ‘for all x in xs and all y in ys , $x \neq y$.’ However, `exists` and `all` combine well with the other functionals.

Useful combinations for lists of lists include

```
exists(exists pred)
filter(exists pred)
takewhile(all pred)
```

5.10 The list functionals *foldl* and *foldr*

These functionals are unusually general. They apply a 2-argument function over the elements of a list:

$$\text{foldl } f \ e \ [x_1, \dots, x_n] = f(x_n, \dots, f(x_1, e) \dots)$$

$$\text{foldr } f \ e \ [x_1, \dots, x_n] = f(x_1, \dots, f(x_n, e) \dots)$$

Since expressions are evaluated from the inside out, the *foldl* call applies *f* to the list elements from left to right, while the *foldr* call applies it to them from right to left. The functionals are declared by

```
fun foldl f e []      = e
  | foldl f e (x::xs) = foldl f (f(x, e)) xs;
> val foldl = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

fun foldr f e []      = e
  | foldr f e (x::xs) = f(x, foldr f e xs);
> val foldr = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

Numerous functions can be expressed using *foldl* and *foldr*. The sum of a list of numbers is computed by repeated addition starting from 0:

```
val sum = foldl op+ 0;
> val sum = fn : int list -> int
sum [1,2,3,4];
> 10 : int
```

The product is computed by repeated multiplication from 1. Binding the function to an identifier is not necessary:

```
foldl op* 1 [1,2,3,4];
> 24 : int
```

These definitions work because 0 and 1 are the *identity elements* of + and ×, respectively; in other words, $0 + k = k$ and $1 \times k = k$ for all *k*. Many applications of *foldl* and *foldr* are of this sort.

Both functionals take as their first argument a function of type $\sigma \times \tau \rightarrow \tau$. This function may itself be expressed using functionals. A nested application of *foldl* adds a list of lists:


```
foldl (fn (ns, n) => foldl op+ n ns) 0 [[1], [2,3], [4,5,6]];
> 21 : int
```

This is more direct than `sum(map sum [[1], [2, 3], [4, 5, 6]])`, which forms the intermediate list of sums `[1, 5, 15]`.

List construction (the operator `::`) has a type of the required form. Supplying it to *foldl* yields an efficient reverse function:

```
foldl op:: [] (explode "Richard");
> [#"d", #"r", #"a", #"h", #"c", #"i", #"R"] : char list
```

An iterative length computation is equally simple:

```
foldl (fn (_, n) => n+1) 0 (explode "Margaret");
> 8 : int
```

To append *xs* and *ys*, apply `::` through *foldr* to each element of *xs*, starting with *ys*:

```
foldr op:: ["out", "thee?"] ["And", "leave"];
> ["And", "leave", "out", "thee?"] : string list
```

Applying append through *foldr* joins a list of lists, like the function `List.concat`; note that `[]` is the identity element of append:

```
foldr op@ [] [[1], [2,3], [4,5,6]];
> [1, 2, 3, 4, 5, 6] : int list
```

Recall that *newmem* adds a member, if not already present, to a list (Section 3.15). Applying that function through *foldr* builds a ‘set’ of distinct elements:

```
foldr newmem [] (explode "Margaret");
> [#"M", #"g", #"a", #"r", #"e", #"t"] : char list
```

To express *map f*, apply a function based on `::` and *f*:

```
fun map f = foldr (fn(x,l)=> f x :: l) [];
> val map = fn : ('a -> 'b) -> 'a list -> 'b list
```

Two calls to *foldr* compute the Cartesian product of two lists:

```
fun cartprod (xs, ys) =
  foldr (fn (x, pairs) =>
    foldr (fn (y, l) => (x,y)::l) pairs ys)
    [] xs;
> val cartprod = fn : 'a list * 'b list -> ('a * 'b) list
```

Cartesian products can be computed more clearly using *map* and *List.concat*, at the expense of creating an intermediate list. Declare a curried pairing function:

```
fun pair x y = (x,y);
> val pair = fn : 'a -> 'b -> 'a * 'b
```


A list of lists of pairs is created ...

```
map (fn a => map (pair a) ["Hastings","Stanley"])
  ["Lord","Lady"];
> [(["Lord", "Hastings"), ("Lord", "Stanley")],
>  [(["Lady", "Hastings"), ("Lady", "Stanley")]]
> : (string * string) list list
```

... then concatenated to form the Cartesian product:

```
List.concat it;
> [(["Lord", "Hastings"), ("Lord", "Stanley"),
>  ("Lady", "Hastings"), ("Lady", "Stanley")]
> : (string * string) list
```

Both algorithms for Cartesian products can be generalized, replacing (x, y) by other functions of x and y , to express sets of the form $\{f(x, y) \mid x \in xs, y \in ys\}$.

 *Functionals and the standard library.* The infix *o*, for function composition, is available at top level. Also at top level are the list functionals *map*, *foldl* and *foldr*; they are separately available as components of structure *List*, as are *filter*, *exists* and *all*. Structure *ListPair* provides variants of *map*, *exists* and *all* that take a 2-argument function and operate on a pair of lists. For example, *ListPair.map* applies a function to pairs of corresponding elements of two lists:

$$ListPair.map f ([x_1, \dots, x_n], [y_1, \dots, y_n]) = [f(x_1, y_1), \dots, f(x_n, y_n)]$$

If the lists have unequal lengths, the unmatched elements are ignored. The same result can be obtained using *List.map* and *ListPair.zip*, but this builds an intermediate list.

Exercise 5.14 Express the function *union* (Section 3.15) using functionals.

Exercise 5.15 Simplify matrix multiplication (Section 3.10) using functionals.

Exercise 5.16 Express *exists* using *foldl* or *foldr*.

Exercise 5.17 Using functionals, express the conditional set expression

$$\{x - y \mid x \in xs, y \in ys, y < x\}.$$

5.11 More examples of recursive functionals

Binary trees and similar recursive types can be processed using recursive functionals. Even the natural numbers 0, 1, 2, ... can be viewed as a recursive type: their constructors are 0 and the successor function.

Powers of a function. If f is a function and $n \geq 0$ then f^n is the function such that

$$f^n(x) = \underbrace{f(\dots f(f(x))\dots)}_{n \text{ times}}$$

This is the function *repeat f n*:

```
fun repeat f n x =
  if n>0 then repeat f (n-1) (f x)
  else x;
> val repeat = fn : ('a -> 'a) -> int -> 'a -> 'a
```

Surprisingly many functions have this form. Examples include *drop* and *replist* (declared in Sections 3.4 and 5.2, respectively):

```
repeat tl 5 (explode "I'll drown you in the malmsey-butt...");
> [#"d", #"r", #"o", #"w", #"n", #" ", ...] : char list
repeat (secl "Ha!" op::) 5 [];
> ["Ha!", "Ha!", "Ha!", "Ha!", "Ha!"] : string list
```

Complete binary trees with a constant label are created by

```
repeat (fn t=>Br("No",t,t)) 3 Lf;
> Br ("No", Br ("No", Br ("No", Lf, Lf),
>                               Br ("No", Lf, Lf)),
>      Br ("No", Br ("No", Lf, Lf),
>      Br ("No", Lf, Lf)))
> : string tree
```

A suitable function on pairs, when repeated, computes factorials:

```
fun factaux (k,p) = (k+1, k*p);
> val factaux = fn : int * int -> int * int
repeat factaux 5 (1,1);
> (6, 120) : int * int
```

Tree recursion. The functional *treerec*, for binary trees, is analogous to *foldr*. Calling *foldr f e xs*, figuratively speaking, replaces *::* by *f* and *nil* by *e* in a list. Given a tree, *treefold* replaces each leaf by some value *e* and each branch by the application of a 3-argument function *f*.

```

fun treefold f e Lf          = e
  | treefold f e (Br (u, t1, t2)) = f (u, treefold f e t1, treefold f e t2);
> val treefold = fn
> : ('a * 'b * 'b -> 'b) -> 'b -> 'a tree -> 'b

```

This functional can express many of the tree functions of the last chapter. The function *size* replaces each leaf by 0 and each branch by a function to add 1 to the sizes of the subtrees:

```
treefold (fn (_, c1, c2) => 1+c1+c2) 0
```

The function *depth* computes a maximum at each branch:

```
treefold (fn (_, d1, d2) => 1 + Int.max (d1, d2)) 0
```

Tree recursion over a reversed version of *Br* defines *reflect*:

```
treefold (fn (u, t1, t2) => Br (u, t2, t1)) Lf
```

To compute a preorder list, each branch joins its label to the lists for the subtrees:

```
treefold (fn (u, l1, l2) => [u] @ l1 @ l2) []
```

Operations on terms. The set of terms $x, f(x), g(x, f(x)), \dots$, which is generated by variables and function applications, corresponds to the ML datatype

```

datatype term = Var of string
              | Fun of string * term list;

```

The term $(x + u) - (y \times x)$ could be declared by

```

val tm = Fun("-", [Fun("+", [Var "x", Var "u"]),
                  Fun("*", [Var "y", Var "x"])]);

```

Though it is natural to represent a function's arguments as an ML list, the types *term* and *term list* must be regarded as mutually recursive. A typical function on terms will make use of a companion function on term lists. Fortunately, the companion function need not be declared separately; in most instances it can be expressed using list functionals.

If the ML function $f : \text{string} \rightarrow \text{term}$ defines a substitution from variables to terms, then *subst f* extends this over terms. Observe how *map* applies the substitution to term lists.

```

fun subst f (Var a)          = f a
  | subst f (Fun (a, args)) = Fun (a, map (subst f) args);
> val subst = fn : (string -> term) -> term -> term

```

The list of variables in a term could also be computed using *map*:

```

fun vars (Var a)          = [a]
  | vars (Fun (_, args)) = List.concat (map vars args);
> val vars = fn : term -> string list
vars tm;
> ["x", "u", "y", "x"] : string list

```

This is wasteful because `List.concat` copies lists repeatedly. Instead, declare a function `accumVars` with an argument to accumulate a list of variables. It can be extended to term lists using `foldr`:

```

fun accumVars (Var a, bs)      = a::bs
  | accumVars (Fun (_, args), bs) = foldr accumVars bs args;
> val accumVars = fn : term * string list -> string list
accumVars (tm, []);
> ["x", "u", "y", "x"] : string list

```

Here is a demonstration. A trivial substitution, `replace t a` replaces the variable `a` by `t` while leaving other variables unchanged:

```

fun replace t a b = if a=b then t else Var b;
> val replace = fn : term -> string -> string -> term

```

Thus, `subst (replace t a) u` replaces `a` by `t` throughout the term `u`. Substituting `-z` for `x` in `tm` yields the term $(-z + u) - (y \times -z)$:

```

subst (replace (Fun("-", [Var "z"])) "x") tm;
> Fun ("-",
>      [Fun ("+", [Fun ("-", [Var "z"]), Var "u"]),
>      Fun ("*", [Var "y", Fun ("-", [Var "z"])])])
> : term

```

Now the list of variables contains `z` in place of `x`:

```

accumVars (it, []);
> ["z", "u", "y", "z"] : string list

```

Exercise 5.18 Declare the functional `prefold` such that `prefold f e t` is equivalent to `foldr f e (preorder t)`.

Exercise 5.19 Write a function `nf` such that `repeat nf` computes Fibonacci numbers.

Exercise 5.20 What is this function good for?

```

fun funny f 0 = I
  | funny f n = if n mod 2 = 0
                 then funny (f o f) (n div 2)
                 else funny (f o f) (n div 2) o f;

```

Exercise 5.21 What function is *treefold* F I , where F is declared as follows?

```
fun F (v, f1, f2) vs = v :: f1 (f2 vs);
```

Exercise 5.22 Consider counting the *Fun* nodes in a term. Express this as a function modelled on *vars*, then as a function modelled on *accumVars* and finally without using functionals.

Exercise 5.23 Note that the result of *vars* tm mentions x twice. Write a function to compute the list of variables in a term without repetitions. Can you find a simple solution using functionals?

Sequences, or infinite lists

Lazy lists are one of the most celebrated features of functional programming. The elements of a lazy list are not evaluated until their values are required by the rest of the program; thus a lazy list may be infinite. In lazy languages like Haskell, all data structures are lazy and infinite lists are commonplace in programs. In ML, which is not lazy, infinite lists are rare. This section describes how to express infinite lists in ML, representing the tail of a list by a function in order to delay its evaluation.

It is important to recognize the hazards of programming with lazy lists. Hitherto we have expected every function, from the greatest common divisor to priority queues, to deliver its result in finite time. Recursion was used to reduce a problem to simpler subproblems. Every recursive function included a base case where it would terminate.

Now we shall be dealing with potentially infinite results. We may view any finite part of an infinite list, but never the whole. We may add two infinite lists element by element to form a list of sums, but may not reverse an infinite list or find its smallest element. We shall define recursions that go on forever, with no base case. Instead of asking whether the program terminates, we can only ask whether the program generates each finite part of its result in finite time.

ML functions on infinite lists are more complicated than their counterparts in a lazy language. By laying the mechanism bare, however, they may help us avoid some pitfalls. Mechanistic thinking should not be our only tool; computations over infinite values may exceed our powers of imagination. Domain theory gives a deeper view of such computations (Gunter, 1992; Winskel, 1993).

5.12 A type of sequences

Infinite lists are traditionally called *streams*, but let us call them *sequences*. (A ‘stream’ in ML is an input/output channel.) Like a list, a sequence either is empty or contains a head and tail. The empty sequence is *Nil* and a non-empty sequence has the form $Cons(x, xf)$, where x is the head and xf is a function to compute the tail:³

```
datatype 'a seq = Nil
                | Cons of 'a * (unit -> 'a seq);
```

Starting from this declaration, we shall interactively develop a set of sequence primitives, by analogy with lists. Later, to avoid name clashes, we shall group them into an appropriate structure.

Functions to return the head and tail of a sequence are easily declared. As with lists, inspecting the empty sequence should raise an exception:

```
exception Empty;
fun hd (Cons(x, xf)) = x
  | hd Nil           = raise Empty;
> val hd = fn : 'a seq -> 'a
```

To inspect the tail, apply the function xf to (). The argument, the sole value of type *unit*, conveys no information; it merely forces evaluation of the tail.

```
fun tl (Cons(x, xf)) = xf()
  | tl Nil           = raise Empty;
> val tl = fn : 'a seq -> 'a seq
```

Calling $cons(x, xq)$ combines a head x and tail sequence xq to form a longer sequence:

```
fun cons(x, xq) = Cons(x, fn()=>xq);
> val cons = fn : 'a * 'a seq -> 'a seq
```

Note that $cons(x, E)$ is not evaluated lazily. ML evaluates the expression E , yielding say xq , and returns $Cons(x, fn()=>xq)$. So the fn inside $cons$ does not delay the evaluation of the tail. Only use $cons$ where lazy evaluation is not required, say to convert a list into a sequence:

```
fun fromList l = List.foldr cons Nil l;
> val fromList = fn : 'a list -> 'a seq
```

To delay the evaluation of E , write $Cons(x, fn()=>E)$ instead of $cons(x, E)$. Let us define the increasing sequence of integers starting from k :

³ Type *unit* was described in Section 2.8.

```

fun from k = Cons(k, fn()=> from(k+1));
> val from = fn : int -> int seq
from 1;
> Cons (1, fn) : int seq

```

The sequence starts with 1; here are some more elements:

```

tl it;
> Cons (2, fn) : int seq
tl it;
> Cons (3, fn) : int seq

```

Calling $take(xq, n)$ returns the first n elements of the sequence xq as a list:

```

fun take (xq, 0)          = []
  | take (Nil, n)         = raise Subscript
  | take (Cons(x,xf), n) = x :: take (xf(), n-1);
> val take = fn : 'a seq * int -> 'a list
take (from 30, 7);
> [30, 31, 32, 33, 34, 35, 36] : int list

```

How does it work? The computation of $take(from\ 30, 2)$ goes as follows:

```

take(from 30, 2)
⇒ take(Cons(30, fn()=>from(30+1)), 2)
⇒ 30 :: take(from(30+1), 1)
⇒ 30 :: take(Cons(31, fn()=>from(31+1)), 1)
⇒ 30 :: 31 :: take(from(31+1), 0)
⇒ 30 :: 31 :: take(Cons(32, fn()=>from(32+1)), 0)
⇒ 30 :: 31 :: []
⇒ [30, 31]

```

Observe that the element 32 is computed but never used. Type $\alpha\ seq$ is not really lazy; the head of a non-empty sequence is always computed. What is worse, inspecting the tail repeatedly evaluates it repeatedly; we do not have call-by-need, only call-by-name. Such defects can be cured at the cost of considerable extra complication (see Section 8.4).

Exercise 5.24 Explain what is wrong with this version of $from$, describing the computation steps of $take(badfrom\ 30, 2)$.

```

fun badfrom k = cons(k, badfrom(k+1));

```


Exercise 5.25 This variant of type α *seq* represents every non-empty sequence by a function, preventing premature evaluation of the first element (Reade, 1989, page 324). Code the functions *from* and *take* for this type of sequences:

```
datatype 'a seq = Nil
              | Cons of unit -> 'a * 'a seq;
```

Exercise 5.26 This variant of α *seq*, declared using mutual recursion, is even lazier than the one above. Every sequence is a function, delaying even the computation needed to tell if a sequence is non-empty. Code the functions *from* and *take* for this type of sequences:

```
datatype 'a seqnode = Nil
                  | Cons of 'a * 'a seq
and 'a seq = Seq of unit -> 'a seqnode;
```

5.13 Elementary sequence processing

For a function on sequences to be computable, each finite part of the output must depend on at most a finite part of the input. Consider squaring a sequence of integers one by one. The tail of the output, when evaluated, applies *squares* to the tail of the input.

```
fun squares Nil : int seq = Nil
  | squares (Cons(x,xf)) = Cons(x*x, fn()=> squares(xf()));
> val squares = fn : int seq -> int seq
squares (from 1);
> Cons(1, fn) : int seq
take(it, 10);
> [1, 4, 9, 16, 25, 36, 49, 64, 81, 100] : int list
```

Adding corresponding elements of two sequences is similar. Evaluating the tail of the output evaluates the tails of the two inputs. If either input sequence becomes empty, then so does the output.

```
fun add (Cons(x,xf), Cons(y,yf)) = Cons(x+y,
                                       fn()=> add(xf(), yf()))
  | add _ : int seq = Nil;
> val add = fn : int seq * int seq -> int seq
add (from 10000, squares (from 1));
> Cons(10001, fn) : int seq
take(it, 5);
> [10001, 10005, 10011, 10019, 10029] : int list
```

The *append* function for sequences works like the one for lists. The elements of $xq @ yq$ are first taken from xq ; when xq becomes empty, elements are taken from yq .

```

fun Nil @ yq = yq
  | (Cons(x,xf)) @ yq = Cons(x, fn()=> (xf()) @ yq);
> val @ = fn : 'a seq * 'a seq -> 'a seq

```

For a simple demonstration, let us build a finite sequence using *fromList*.

```

val finite = fromList [25,10];
> Cons (25, fn) : int seq
finite @ from 1415;
> Cons (25, fn) : int seq
take(it, 3);
> [25, 10, 1415] : int list

```

If *xq* is infinite then *xq @ yq* equals *xq*. A variant of append combines infinite sequences fairly. The elements of two sequences can be *interleaved*:

```

fun interleave (Nil, yq) = yq
  | interleave (Cons(x,xf), yq) =
    Cons(x, fn()=> interleave(yq, xf()));
> val interleave = fn : 'a seq * 'a seq -> 'a seq
take(interleave(from 0, from 50), 10);
> [0, 50, 1, 51, 2, 52, 3, 53, 4, 54] : int list

```

In its recursive call, *interleave* exchanges the two sequences so that neither can exclude the other.

Functionals for sequences. List functionals like *map* and *filter* can be generalized to sequences. The function *squares* is an instance of the functional *map*, which applies a function to every element of a sequence:

```

fun map f Nil = Nil
  | map f (Cons(x,xf)) = Cons(f x, fn()=> map f (xf()));
> val map = fn : ('a -> 'b) -> 'a seq -> 'b seq

```

To filter a sequence, successive tail functions are called until an element is found to satisfy the given predicate. If no such element exists, the computation will never terminate.

```

fun filter pred Nil = Nil
  | filter pred (Cons(x,xf)) =
    if pred x then Cons(x, fn()=> filter pred (xf()))
    else filter pred (xf());
> val filter = fn : ('a -> bool) -> 'a seq -> 'a seq
filter (fn n => n mod 10 = 7) (from 50);
> Cons (57, fn) : int seq
take(it, 8);
> [57, 67, 77, 87, 97, 107, 117, 127] : int list

```

The function *from* is an instance of the functional *iterates*, which generates sequences of the form $[x, f(x), f(f(x)), \dots, f^k(x), \dots]$:

```
fun iterates f x = Cons(x, fn () => iterates f (f x));
> val iterates = fn : ('a -> 'a) -> 'a -> 'a seq
iterates (secr op/ 2.0) 1.0;
> Cons (1.0, fn) : real seq
take(it, 5);
> [1.0, 0.5, 0.25, 0.125, 0.0625] : real list
```

A structure for sequences. Let us again gather up the functions we have explored, making a structure. As in the binary tree structure (Section 4.13), we leave the datatype declaration outside to allow direct reference to the constructors. Imagine that the other sequence primitives have been declared not at top level but in a structure *Seq* satisfying the following signature:

```
signature SEQUENCE =
sig
  exception Empty
  val cons      : 'a * 'a seq -> 'a seq
  val null     : 'a seq -> bool
  val hd       : 'a seq -> 'a
  val tl       : 'a seq -> 'a seq
  val fromList : 'a list -> 'a seq
  val toList   : 'a seq -> 'a list
  val take     : 'a seq * int -> 'a list
  val drop     : 'a seq * int -> 'a seq
  val @        : 'a seq * 'a seq -> 'a seq
  val interleave : 'a seq * 'a seq -> 'a seq
  val map      : ('a -> 'b) -> 'a seq -> 'b seq
  val filter   : ('a -> bool) -> 'a seq -> 'a seq
  val iterates : ('a -> 'a) -> 'a -> 'a seq
  val from     : int -> int seq
end;
```

Exercise 5.27 Declare the missing functions *null* and *drop* by analogy with the list versions. Also declare *toList*, which converts a finite sequence to a list.

Exercise 5.28 Show the computation steps of *add(from 5, squares(from 9))*.

Exercise 5.29 Declare a function that, given a positive integer *k*, transforms a sequence $[x_1, x_2, \dots]$ into a new sequence by repeating each element *k* times:

$$[\underbrace{x_1, \dots, x_1}_{k \text{ times}}, \underbrace{x_2, \dots, x_2}_{k \text{ times}}, \dots]$$

Exercise 5.30 Declare a function to add adjacent elements of a sequence, transforming $[x_1, x_2, x_3, x_4, \dots]$ to $[x_1 + x_2, x_3 + x_4, \dots]$.

Exercise 5.31 Which of the list functionals *takewhile*, *dropwhile*, *exists* and *all* can sensibly be generalized to infinite sequences? Code those that can be, and explain what goes wrong with the others.

5.14 Elementary applications of sequences

We can use structure *Seq* for making change, to express an infinite sequence of random numbers and to enumerate the prime numbers. These examples especially illustrate the sequence functionals.

Making change, revisited. The function *allChange* (Section 3.7) computes all possible ways of making change. It is not terribly practical: using British coin values, there are 4366 different ways of making change for 99 pence!

If the function returned a sequence, it could compute solutions upon demand, saving time and storage. Getting the desired effect in ML requires care. Replacing the list operations by sequence operations in *allChange* would achieve little. The new function would contain two recursive calls, with nothing to delay the second call's execution. The resulting sequence would be fully evaluated.

```
Seq.@ (allChange (c::coins, c::coinvals, amount-c),
      allChange (coins, coinvals, amount))
```

Better is to start with the solution of Exercise 3.14, where the append is replaced by an argument to accumulate solutions. An accumulator argument is usually a list. Should we change it to a sequence?

```
fun seqChange (coins, coinvals, 0, coinsf)           = Cons (coins, coinsf)
  | seqChange (coins, [], amount, coinsf)          = coinsf ()
  | seqChange (coins, c::coinvals, amount, coinsf) =
    if amount < 0 then coinsf ()
    else seqChange (c::coins, c::coinvals, amount-c,
                   fn () => seqChange (coins, coinvals, amount, coinsf));
> val seqChange = fn : int list * int list * int *
>                  (unit -> int list seq) -> int list seq
```

Instead of a sequence there is a tail function *coinsf* of type $unit \rightarrow int\ list\ seq$. This allows us to use *Cons* in the first line, instead of the eager *Seq.cons*. And it requires a *fn* around the inner recursive call, delaying it. This sort of thing is easier in Haskell.

We can now enumerate solutions, getting each one instantly:

```
seqChange ([], gbcoins, 99, fn () => Nil);
```

```

> Cons ([2, 2, 5, 20, 20, 50], fn) : int list seq
Seq.tl it;
> Cons ([1, 1, 2, 5, 20, 20, 50], fn) : int list seq
Seq.tl it;
> Cons ([1, 1, 1, 1, 5, 20, 20, 50], fn) : int list seq

```

The overheads are modest. Computing all solutions takes 354 msec, which is about 1/3 slower than the list version of the function and twice as fast as the original *allChange*.

Random numbers. In Section 3.18 we generated a list of 10,000 random numbers for the sorting examples. However, we seldom know in advance how many random numbers are required. Conventionally, a random number generator is a procedure that stores the ‘seed’ in a local variable. In a functional language, we can define an infinite sequence of random numbers. This hides the implementation details and generates the numbers as they are required.

```

local val a = 16807.0 and m = 2147483647.0
      fun nextRand seed =
          let val t = a*seed
              in t - m * real(floor(t/m)) end
      in
          fun randseq s = Seq.map (secl op/ m)
                                (Seq.iterates nextRand (real s))
          end;
      > val randseq = fn : int -> real seq

```

Observe how *Seq.iterates* generates a sequence of numbers, which *Seq.map* divides by *m*. The random numbers are reals between 0 and 1, exclusive. Using *Seq.map* we convert them to integers from 0 to 9:

```

Seq.map (floor o secl(10.0) op* ) (randseq 1);
> Cons (0, fn) : int seq
Seq.take (it, 12);
> [0, 0, 1, 7, 4, 5, 2, 0, 6, 6, 9, 3] : int list

```

Prime numbers. The sequence of prime numbers can be computed by the Sieve of Eratosthenes.

- Start with the sequence [2, 3, 4, 5, 6, ...].
- Take 2 as a prime. Delete all multiples of 2, since they cannot be prime. This leaves the sequence [3, 5, 7, 9, 11, ...].
- Take 3 as a prime and delete its multiples. This leaves the sequence [5, 7, 11, 13, 17, ...].
- Take 5 as a prime

At each stage, the sequence contains those numbers not divisible by any of the primes generated so far. Therefore its head is prime, and the process can continue indefinitely.

The function *sift* deletes multiples from a sequence, while *sieve* repeatedly sifts a sequence:

```
fun sift p = Seq.filter (fn n => n mod p <> 0);
> val sift = fn : int -> int seq -> int seq
fun sieve (Cons(p, nf)) = Cons(p, fn()=> sieve (sift p (nf())));
> val sieve = fn : int seq -> int seq
```

The sequence *primes* results from *sieve* [2, 3, 4, 5, ...]. No primes beyond the first are generated until the sequence is inspected.

```
val primes = sieve (Seq.from 2);
> val primes = Cons(2, fn) : int seq
Seq.take (primes, 25);
> [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,
> 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97] : int list
```

When we write programs such as these, ML types help to prevent confusion between sequences and tail functions. A sequence has type $\tau \text{ seq}$ while a tail function has type $\text{unit} \rightarrow \tau \text{ seq}$. We can insert a function call $\dots()$ or a function abstraction $\text{fn}() \Rightarrow \dots$ in response to type error messages.

5.15 Numerical computing

Sequences have applications in numerical analysis. This may seem surprising at first, but, after all, many numerical methods are based on infinite series. Why not express them literally?

Square roots are a simple example. Recall the Newton-Raphson method for computing the square root of some number a . Start with a positive approximation x_0 . Compute further approximations by the rule

$$x_{k+1} = \left(\frac{a}{x_k} + x_k \right) / 2,$$

stopping when two successive approximations are sufficiently close. With sequences we can perform this computation directly.

The function *nextApprox* computes x_{k+1} from x_k . Iterating it computes the series of approximations.

```
fun nextApprox a x = (a/x + x) / 2.0;
> val nextApprox = fn : real -> real -> real
Seq.take (Seq.iterates (nextApprox 9.0) 1.0, 7);
> [1.0, 5.0, 3.4, 3.023529412, 3.000091554,
```

```
> 3.000000001, 3.0] : real list
```

The simplest termination test is to stop when the absolute difference between two approximations is smaller than a given tolerance $\epsilon > 0$ (written *eps* below).⁴

```
fun within (eps:real) (Cons(x,xf)) =
  let val Cons(y,yf) = xf()
  in if Real.abs(x-y) < eps then y
     else within eps (Cons(y,yf))
  end;
> val within = fn : real -> real seq -> real
```

Putting 10^{-6} for the tolerance and 1 for the initial approximation yields a square root function:

```
fun qroot a = within 1E-6 (Seq.iterates (nextApprox a) 1.0);
> val qroot = fn : real -> real
qroot 5.0;
> 2.236067977 : real
it*it;
> 5.0 : real
```

Would not a Fortran program be better? This example follows Hughes (1989) and Halfant and Sussman (1988), who show how interchangeable parts involving sequences can be assembled into numerical algorithms. Each algorithm is tailor made to suit its application.

For instance, there are many termination tests to choose from. The absolute difference ($|x - y| < \epsilon$) tested by *within* is too strict for large numbers. We could test relative difference ($|x/y - 1| < \epsilon$) or something fancier:

$$\frac{|x - y|}{(|x| + |y|)/2 + 1} < \epsilon$$

Sometimes it is prudent to test that three or more approximations are sufficiently close.

Each termination test can be packaged as a function from sequences to reals. Techniques like Richardson extrapolation (for accelerating the convergence of a series) can be packaged as functions from sequences to sequences. These functions can be combined to perform numerical differentiation, integration and so on.

⁴ The recursive call passes *Cons(y, yf)* rather than *xf()*, which denotes the same value, to avoid calling *xf()* twice. Recall that our sequences are not truly lazy, but employ a call-by-name rule.

Exercise 5.32 Compute the exponential function e^x by generating a sequence for the infinite sum

$$e^x = \frac{1}{0!} + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^k}{k!} + \cdots$$

Exercise 5.33 Write an ML function to take a value from a sequence using one of the other termination tests mentioned above. Declare a square root (or exponential) function using it.

5.16 Interleaving and sequences of sequences

Given infinite sequences xq and yq , consider forming the sequence of all pairs (x, y) with x from xq and y from yq . This problem illustrates the subtleties of computing with infinities.

As remarked above in Section 5.10, a list of lists can be generated using *map* with the curried pairing function *pair*. A sequence of sequences can be generated similarly:

```
fun makeqq (xq, yq) = Seq.map (fn x => Seq.map (pair x) yq) xq;
> val makeqq = fn : 'a seq * 'b seq -> ('a * 'b) seq seq
```

A sequence of sequences can be viewed using *takeqq*($xqq, (m, n)$). This list of lists is the $m \times n$ upper left rectangle of xqq .

```
fun takeqq (xqq, (m, n)) = map (secl Seq.take n) (Seq.take(xqq, m));
> val takeqq = fn
> : 'a seq seq * (int * int) -> 'a list list
makeqq (Seq.from 30, primes);
> Cons (Cons ((30, 2), fn), fn) : (int * int) seq seq
takeqq (it, (3, 5));
> [[(30, 2), (30, 3), (30, 5), (30, 7), (30, 11)],
> [(31, 2), (31, 3), (31, 5), (31, 7), (31, 11)],
> [(32, 2), (32, 3), (32, 5), (32, 7), (32, 11)]]
> : (int * int) list list
```

The function *List.concat* appends the members of a list of lists, forming one list. Let us declare an analogous function *enumerate* to combine a sequence of sequences. Because the sequences may be infinite, we must use *interleave* instead of *append*.

Here is the idea. If the input sequence has head xq and tail xqq , recursively enumerate xqq and interleave the result with xq . If we take *List.concat* as a model we end up with faulty code:


```

fun enumerate Nil = Nil
  | enumerate (Cons(xq, xqf)) = Seq.interleave(xq, enumerate(xqf()));
> val enumerate = fn : 'a seq seq -> 'a seq

```

If the input to this function is infinite, ML will make an infinite series of recursive calls, generating no output. This version would work in a lazy functional language, but with ML we must explicitly terminate the recursive calls as soon as some output can be produced. This requires a more complex case analysis. If the input sequence is non-empty, examine its head; if that is also non-empty then it contains an element for the output.

```

fun enumerate Nil = Nil
  | enumerate (Cons(Nil, xqf)) = enumerate(xqf())
  | enumerate (Cons(Cons(x, xf), xqf)) =
    Cons(x, fn()=> Seq.interleave(enumerate(xqf()), xf()));
> val enumerate = fn : 'a seq seq -> 'a seq

```

The second and third cases simulate the incorrect version's use of *interleave*, but the enclosing `fn()=>...` terminates the recursive calls.

Here is the sequence of all pairs of positive integers.

```

val pairqq = makeqq (Seq.from 1, Seq.from 1);
> val pairqq = Cons (Cons ((1, 1), fn), fn)
> : (int * int) seq seq
Seq.take(enumerate pairqq, 18);
> [(1, 1), (2, 1), (1, 2), (3, 1), (1, 3), (2, 2), (1, 4),
>  (4, 1), (1, 5), (2, 3), (1, 6), (3, 2), (1, 7), (2, 4),
>  (1, 8), (5, 1), (1, 9), (2, 5)] : (int * int) list

```

We can be more precise about the order of enumeration. Consider the following declarations:

```

fun powof2 n = repeat double n 1;
> val powof2 = fn : int -> int
fun pack(i, j) = powof2(i-1) * (2*j - 1);
> val pack = fn : int * int -> int

```

This function, $pack(i, j) = 2^{i-1}(2j - 1)$, establishes a one-to-one correspondence between positive integers and pairs (i, j) of positive integers. Thus, the Cartesian product of two countable sets is a countable set. Here is a small table of this function:

```

val nqq = Seq.map (Seq.map pack) pairqq;
> val nqq = Cons (Cons (1, fn), fn) : int seq seq
takeqq(nqq, (4, 6));
> [[1, 3, 5, 7, 9, 11],
>  [2, 6, 10, 14, 18, 22],
>  [4, 12, 20, 28, 36, 44],

```

```
> [8, 24, 40, 56, 72, 88]] : int list list
```

Our enumeration decodes the packing function, returning the sequence of positive integers in their natural order:

```
Seq.take (enumerate nqq, 12);
> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12] : int list
```

It is not hard to see why this is so. Each interleaving takes half its elements from one sequence and half from another. Repeated interleaving distributes the places in the output sequence by powers of two, as in the packing function.

Exercise 5.34 Predict, or at least explain, ML's response to the following:

```
enumerate (Seq.iterates I Nil);
```

Exercise 5.35 Generate the sequence of all finite lists of positive integers. (Hint: first, declare a function to generate the sequence of lists having a given length.)

Exercise 5.36 Show that for every positive integer k there are unique positive integers i and j such that $k = \text{pack}(i, j)$. What is $\text{pack}(i, j)$ in binary notation?

Exercise 5.37 Adapt the definition of type $\alpha \text{ seq}$ to declare a type of infinite binary trees. Write a function itr that, applied to an integer n , constructs the tree whose root has the label n and the two subtrees $\text{itr}(2n)$ and $\text{itr}(2n + 1)$.

Exercise 5.38 (Continuing the previous exercise.) Write a function to build a sequence consisting of all the labels in a given infinite binary tree. In what order are the labels enumerated? Then write an inverse function that constructs an infinite binary tree whose labels are given by a sequence.

Search strategies and infinite lists

Theorem proving, planning and other Artificial Intelligence applications require search. There are many search strategies:

- Depth-first search is cheap, but it may follow a blind alley and run forever without finding any solutions.
- Breadth-first search is *complete* — certain to find all the solutions — but it requires a huge amount of space.
- Depth-first iterative deepening is complete and requires little space, but can be slow.

- Best-first search must be guided by a function to estimate the distance from a solution.

By representing the set of solutions as a lazy list, the search strategy can be chosen independently from the process that consumes the solutions. The lazy list serves as a communication channel: the producer generates its elements and the consumer removes them. Because the list is lazy, its elements are not produced until the consumer requires them.

Figures 5.1 and 5.2 contrast the depth-first and breadth-first strategies, applying both to the same tree. The tree is portrayed at some point during the search, with subtrees not yet visited as wedges. Throughout this section, no tree node may have an infinite number of branches. Trees may have infinite depth.

In *depth-first search*, each subtree is fully searched before its brother to the right is considered. The numbers in the figure show the order of the visits. Node 5 is reached because node 4 is a leaf, while four subtrees remain to be visited. If the subtree below node 5 is infinite, the other subtrees will never be reached: the strategy is incomplete. Depth-first search is familiarly called backtracking.

Breadth-first search visits all nodes at the current depth before moving on to the next depth. In Figure 5.2 it has explored the tree to three levels. Because of finite branching, all nodes will be reached: the strategy is complete. But it is seldom practical, except in trivial cases. To reach a given depth, it visits an exponential number of nodes and uses an exponential amount of storage.

5.17 Search strategies in ML

Infinite trees could be represented rather like infinite lists, namely as an ML datatype containing functions to delay evaluation. For the search trees of this section, however, a node's subtrees can be computed from its label. Trees over type τ (with finite branching) are represented by a function $next : \tau \rightarrow \tau list$, where $next x$ is the list of the subtrees of x .

Depth-first search can be implemented efficiently using a stack to hold the nodes to visit next. At each stage, the head y is removed from the stack and replaced by its subtrees, $next y$, which will be visited before other nodes in the stack. Nodes are included in the output in the order visited.

```
fun depthFirst next x =
  let fun dfs [] = Nil
      | dfs (y::ys) = Cons(y, fn()=> dfs(next y @ ys))
      in dfs [x] end;
  > val depthFirst = fn : ('a -> 'a list) -> 'a -> 'a seq
```

Figure 5.1 A *depth-first search tree*

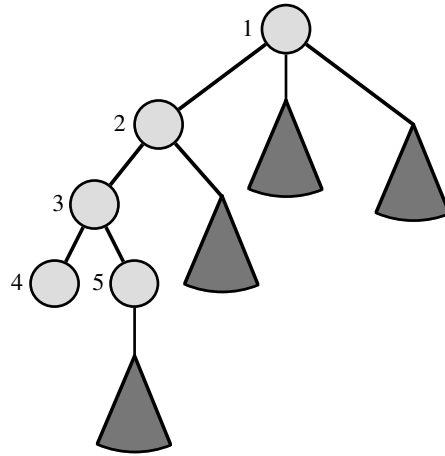
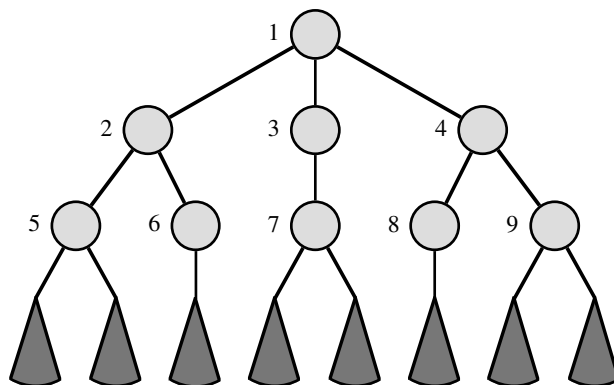


Figure 5.2 A *breadth-first search tree*



Breadth-first search stores the pending nodes on a queue, not on a stack. When y is visited, its successors in $next\ y$ are put at the end of the queue.⁵

```
fun breadthFirst next x =
  let fun bfs [] = Nil
      | bfs (y::ys) = Cons(y, fn()=> bfs (ys @ next y))
      in bfs [x] end;
  > val breadthFirst = fn : ('a -> 'a list) -> 'a -> 'a seq
```

Both strategies simply enumerate all nodes in some order. Solutions are identified using the functional *Seq.filter* with a suitable predicate on nodes. Other search strategies can be obtained by modifying these functions.



Best-first search. Searches in Artificial Intelligence frequently employ a heuristic distance function, which estimates the distance to a solution from any given node. The estimate is added to the known distance from that node to the root, thereby estimating the distance from the root to a solution via that node. These estimates impose an order on the pending nodes, which are stored in a priority queue. The node with the least estimated total distance is visited next.

If the distance function is reasonably accurate, best-first search converges rapidly to a solution. If it is a constant function, then best-first search degenerates to breadth-first search. If it overestimates the true distance, then best-first search may never find any solutions. The strategy takes many forms, the simplest of which is the A* algorithm. See Rich and Knight (1991) for more information.

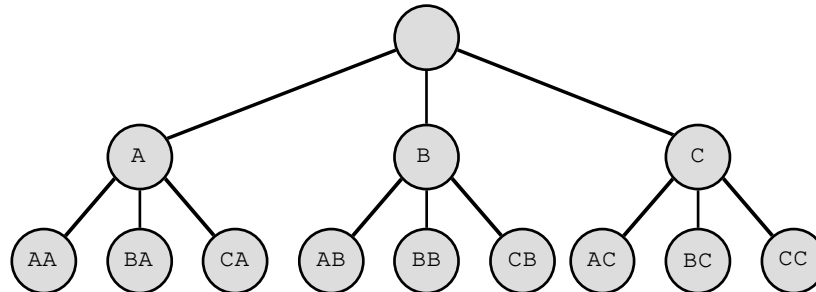
Exercise 5.39 Write versions of *depthFirst* and *breadthFirst* with an additional argument: a predicate to recognize solutions. This is slightly more efficient than the approach used in the text, as it avoids calling *Seq.filter* and copying the sequence of outputs.

Exercise 5.40 Implement best-first search, as described above. Your function must keep track of each node's distance from the root in order to add this to the estimated distance to a solution.

5.18 Generating palindromes

Let us generate the sequence of palindromes over the alphabet $\{A, B, C\}$. Each node of the search tree will be a list l of these letters, with 3 branches to nodes $\#"A"::l$, $\#"B"::l$ and $\#"C"::l$.

⁵ Stacks and queues are represented here by lists. Lists make efficient stacks but poor queues. Section 7.3 presents efficient queues.



Function `nextChar` generates this tree.

```

fun nextChar l = [#"A"::l, #"B"::l, #"C"::l];
> val nextChar = fn : char list -> char list list

```

A **palindrome** is a list that equals its own reverse. Let us declare the corresponding predicate:

```

fun isPalin l = (l = rev l);
> val isPalin = fn : 'a list -> bool

```

There are, of course, more efficient ways of generating palindromes. Our approach highlights the differences between different search strategies. Let us declare a function to help us examine sequences of nodes (`implode` joins a list of characters to form a string):

```

fun show n csq = map implode (Seq.take(csq, n));
> val show = fn : int -> char list seq -> string list

```

Breadth-first search is complete and generates all the palindromes. Let us inspect the sequences before and after filtering:

```

show 8 (breadthFirst nextChar []);
> ["", "A", "B", "C", "AA", "BA", "CA", "AB"] : string list
show 8 (Seq.filter isPalin (breadthFirst nextChar []));
> ["", "A", "B", "C", "AA", "BB", "CC", "AAA"] : string list

```

Depth-first search fails to find all solutions. Since the tree's leftmost branch is infinite, the search never leaves it. We need not bother calling `Seq.filter`:

```

show 8 (depthFirst nextChar []);
> ["", "A", "AA", "AAA", "AAAA", "AAAAA", "AAAAAA", "AAAAAAA",
> "AAAAAAA"] : string list

```

If there is no solution on an infinite branch then depth-first search finds nothing at all. Let us start the search at the label `B`. There is only one palindrome of the form `AA...AB`:

```
show 5 (depthFirst nextChar ["B"]);
> ["B", "AB", "AAB", "AAAB", "AAAAAB"] : string list
```

The attempt to find more than one palindrome in this sequence ...

```
show 2 (Seq.filter isPalin (depthFirst nextChar ["B"]));
```

... runs forever.

On the other hand, breadth-first search explores the entire subtree below B . Filtering yields the sequence of all palindromes ending in B :

```
show 6 (breadthFirst nextChar ["B"]);
> ["B", "AB", "BB", "CB", "AAB", "BAB"] : string list
show 6 (Seq.filter isPalin (breadthFirst nextChar ["B"]));
> ["B", "BB", "BAB", "BBB", "BCB", "BAAB"] : string list
```

Again, we see the importance of a complete search strategy.

5.19 The Eight Queens problem

A classic problem is to place 8 Queens on a chess board so that no Queen may attack another. No two Queens may share a row, column or diagonal. Solutions may be found by examining all safe ways of placing new Queens on successive columns. The root of the search tree contains an empty board. There are 8 positions for a Queen in the first column, so there are 8 branches from the root to boards holding one Queen. Once a Queen has been placed in the first column, there are fewer than 8 safe positions for a Queen in the second column; branching decreases with depth in the tree. A board containing 8 Queens must be a leaf node.

Since the tree is finite, depth-first search finds all solutions. Most published solutions to the problem, whether procedural or functional, encode depth-first search directly. A procedural program, recording the occupation of rows and diagonals using boolean arrays, can find all solutions quickly. Here, the Eight Queens problem simply serves to demonstrate the different search strategies.

We can represent a board position by a list of row numbers. The list $[q_1, \dots, q_k]$ stands for the board having Queens in row q_i of column i for $i = 1, \dots, k$. Function *safeQueen* tests whether a queen can safely be placed in row *newq* of the next column, forming the board $[newq, q_1, \dots, q_k]$. (The other columns are essentially shifted to the left.) The new Queen must not be on the same row or diagonal as another Queen. Note that $|newq - q_i| = i$ exactly when *newq* and q_i share a diagonal.

```
fun safeQueen oldqs newq =
  let fun nodiag (i, []) = true
```

```

| nodiaq (i, q::qs) =
  Int.abs(newq-q) <> i andalso nodiaq (i+1, qs)
in not (newq mem oldqs) andalso nodiaq (1, oldqs) end;

```

To generate the search tree, function *nextQueen* takes a board and returns the list of the safe board positions having a new Queen. Observe the use of the list functionals, *map* with a section and *filter* with a curried function. The Eight Queens problem is generalized to the *n* Queens problem, which is to place *n* Queens safely on an $n \times n$ board. Calling *upto* (declared in Section 3.1) generates the list $[1, \dots, n]$ of candidate Queens.

```

fun nextQueen n qs =
  map (secr op:: qs) (List.filter (safeQueen qs) (upto(1,n)));
> val nextQueen = fn : int -> int list -> int list list

```

Let us declare a predicate to recognize solutions. Since only safe board positions are considered, a solution is any board having *n* Queens.

```

fun isFull n qs = (length qs=n);
> val isFull = fn : int -> 'a list -> bool

```

Function *depthFirst* finds all 92 solutions for 8 Queens. This takes 130 msec:

```

fun depthQueen n = Seq.filter (isFull n) (depthFirst (nextQueen n) []);
> val depthQueen = fn : int -> int list seq
Seq.toList (depthQueen 8);
> [[4, 2, 7, 3, 6, 8, 5, 1], [5, 2, 4, 7, 3, 8, 6, 1],
> [3, 5, 2, 8, 6, 4, 7, 1], [3, 6, 4, 2, 8, 5, 7, 1],
> [5, 7, 1, 3, 8, 6, 4, 2], [4, 6, 8, 3, 1, 7, 5, 2],
> ...] : int list list

```

Since sequences are lazy, solutions can be demanded one by one. Depth-first search finds the first solution quickly (6.6 msec). This is not so important for the Eight Queens problem, but the 15 Queens problem has over two million solutions. We can compute a few of them in one second:

```

Seq.take(depthQueen 15, 3);
> [[8, 11, 7, 15, 6, 9, 13, 4, 14, 12, 10, 2, 5, 3, 1],
> [11, 13, 10, 4, 6, 8, 15, 2, 12, 14, 9, 7, 5, 3, 1],
> [13, 11, 8, 6, 2, 9, 14, 4, 15, 10, 12, 7, 5, 3, 1]]
> : int list list

```

Imagine the design of a procedural program that could generate solutions upon demand. It would probably involve coroutines or communicating processes.

Function *breadthFirst* finds the solutions slowly.⁶ Finding one solution takes

⁶ It takes 310 msec. A version using efficient queues takes 160 msec.

nearly as long as finding all! The solutions reside at the same depth in the search tree; finding the first solution requires searching virtually the entire tree.

5.20 Iterative deepening

Depth-first iterative deepening combines the best properties of the other search procedures. Like depth-first search, it uses little space; like breadth-first search, it is complete. The strategy is to search the tree repeatedly, to finite but increasing depths. First it performs depth-first search down to some depth d , returning all solutions found. It then searches down to depth $2d$, returning all solutions found between depths d and $2d$. It then searches to depth $3d$, and so on. Since each search is finite, the strategy will eventually reach any depth.

The repeated searching is less wasteful than it may appear. Iterative deepening increases the time required to reach a given depth by no more than a constant factor, unless the tree branches very little. There are more nodes between depths kd and $(k + 1)d$ than above kd (Korf, 1985).

For simplicity, let us implement iterative deepening with $d = 1$. It yields the same result as breadth-first search, requiring more time but much less space.

Function `depthFirst` is not easily modified to perform iterative deepening because its stack contains nodes from various depths in the tree. The following search function has no stack; it visits each subtree in a separate recursive call. Argument `sf` of `dfs` accumulates the (possibly infinite!) sequence of solutions.

```
fun depthIter next x =
  let fun dfs k (y, sf) =
        if k=0 then fn()=> Cons(y, sf)
          else foldr (dfs (k-1)) sf (next y)
            fun deepen k = dfs k (x, fn()=> deepen (k+1)) ()
              in deepen 0 end;
    > val depthIter = fn : ('a -> 'a list) -> 'a -> 'a seq
```

Let us examine this declaration in detail. Tail functions (of type $unit \rightarrow \alpha seq$) rather than sequences must be used in order to delay evaluation. The function call `dfs k (y, sf)` constructs the sequence of all solutions found at depth k below node y , followed by the sequence `sf()`. There are two cases to consider.

- 1 If $k = 0$ then y is included in the output.
- 2 If $k > 0$ then let $next\ y = [y_1, \dots, y_n]$. These nodes, the subtrees of y , are processed recursively via `foldr`. The resulting sequence contains all solutions found at depth $k - 1$ below y_1, \dots, y_n :

$$dfs(k - 1)(y_1, \dots, dfs(k - 1)(y_n, sf) \dots) ()$$

Calling *deepen k* creates a tail function to compute *deepen(k + 1)* and passes it to *dfs*, which inserts the solutions found at depth *k*.

Let us try it on the previous examples. Iterative deepening generates the same sequence of palindromes as breadth-first search:

```
show 8 (Seq.filter isPalin (depthIter nextChar []));
> ["", "A", "B", "C", "AA", "BB", "CC", "AAA"] : string list
```

It can also solve the Eight Queens problem, quite slowly (340 msec). With a larger depth interval *d*, iterative deepening recovers some of the efficiency of depth-first search, while remaining complete.

Exercise 5.41 A flaw of *depthIter* is that it explores ever greater depths even if the search space is finite. It can run forever, seeking the 93rd solution to the Eight Queens problem. Correct this flaw; is your version as fast as *depthIter*?

Exercise 5.42 Generalize function *depthIter* to take the depth interval *d* as a parameter. Generate palindromes using *d = 5*. How does the result differ from those obtained by other strategies?

Exercise 5.43 Declare a datatype of finite-branching search trees of possibly infinite depth, using a representation like that of sequences. Write a function to construct the tree generated by a parameter *next* : $\alpha \rightarrow \alpha$ list. Give an example of a tree that cannot be constructed in this way.

Summary of main points

- An ML expression can evaluate to a function.
- A curried function acts like a function of several arguments.
- Higher-order functions encapsulate common forms of computation, reducing the need for separate function declarations.
- A lazy list can contain an infinite number of elements, but only a finite number are ever evaluated.
- A lazy list connects a consumer to a producer, such that items are produced only when they have to be consumed.