Chapter 9

Infinite lists

We have already met infinite lists in Chapter 4 and even given an induction principle for reasoning about them in Chapter 6. But we haven't really appreciated what can be done with them. In this chapter we want to explain in more detail exactly what an infinite list is, and how they can be represented by *cyclic* structures. We also describe another useful method for reasoning about infinite lists, and discuss a number of intriguing examples in which infinite and cyclic lists can be used to good effect.

9.1 Review

Recall that `[m..]` denotes the infinite list of all integers from `m` onwards:

```
ghci> [1..]
[1,2,3,4,5,6,7,{Interrupted}
ghci> zip [1..] "hallo"
[(1,'h'),(2,'a'),(3,'l'),(4,'l'),(5,'o')]
```

It would take forever to print `[1..]`, so we interrupt the first computation. The second example illustrates a simple but typical use of infinite lists in finite computations.

In Haskell, the arithmetic expression `[m..]` is translated into `enumFrom m`, where `enumFrom` is a method in the `Enum` class, and defined by

```
enumFrom :: Integer -> [Integer]
enumFrom m = m:enumFrom (m+1)
```

Thus `[m..]` is defined as an instance of a recursively defined function. The computation makes progress because `(:)` is non-strict in its second argument.

It is important to bear in mind that infinite lists in computing do not have the same properties as infinite sets do in mathematics. For example, in set theory

$$\{x \mid x \in \{1,2,3,\ldots\}, x^2 < 10\}$$

denotes the set $\{1,2,3\}$, but

```
ghci> [x | x <- [1..], x*x < 10]
[1,2,3
```

After printing the first three values the computer gets stuck in an infinite loop looking for the next number after 3 whose square is less than 10. The value of the expression above is the partial list `1:2:3:undefined`.

It is possible to have an infinite list of infinite lists. For example,

```
multiples = [map (n*) [1..] | n <- [2..]]
```

defines an infinite list of infinite lists of numbers, the first three being

```
[2,4,6,8,...]  [3,6,9,12,...]  [4,8,12,16,...]
```

Suppose we ask whether the above list of lists can be merged back into a single list, namely `[2..]`. We can certainly merge two infinite lists:

```
merge :: Ord a => [a] -> [a] -> [a]
merge (x:xs) (y:ys) | x<y  = x:merge xs (y:ys)
                    | x==y = x:merge xs ys
                    | x>y  = y:merge (x:xs) ys
```

This version of `merge` removes duplicates: if the two arguments are in strictly increasing order, so is the result. Note the absence of any clauses of `merge` mentioning the empty list. Now it seems that, if we define

```
mergeAll = foldr1 merge
```

then `mergeAll multiples` will return the infinite list `[2..]`. But it doesn't. What happens is that the computer gets stuck in an infinite loop attempting to compute the first element of the result, namely

```
minimum (map head multiples)
```

It is simply not possible to compute the minimum element in an infinite list. Instead we have to make use of the fact that `map head multiples` is in strictly increasing order, and define

```
mergeAll = foldr1 xmerge
xmerge (x:xs) ys = x:merge xs ys
```

With this definition, `mergeAll multiples` does indeed return `[2..]`.

Finally, recall the induction principle described in Chapter 6 for proving facts about infinite lists. Provided P is a chain-complete assertion, we can prove that $P(\texttt{xs})$ holds for all infinite lists `xs` by showing that: (i) $P(\texttt{undefined})$ holds; and (ii) $P(\texttt{xs})$ implies $P(\texttt{x:xs})$ for all `x` and `xs`. Using this principle, we proved in Chapter 6 that `xs++ys` $=$ `xs` for all infinite lists `xs`. But it's not immediately clear how induction can be used to prove, for example,

```
map fact [0..] = scanl (*) 1 [1..]
```

The obvious result to prove is

```
map fact [0..n] = scanl (*) 1 [1..n]
```

for all `n`, but can one then assert the first identity holds?

9.2 Cyclic lists

Data structures, like functions, can be defined recursively. For instance

```
ones :: [Int]
ones = 1:ones
```

This is an example of a *cyclic* list, a list whose definition is recursive. Contrast this definition with `ones = repeat 1`, where

```
repeat x = x:repeat x
```

This definition of `ones` creates an infinite, not a cyclic list. We could define

```
repeat x = xs  where xs = x:xs
```

Now the function `repeat` is defined in terms of a cyclic list. The second definition (call it `repeat2`) is faster to evaluate than the first (call it `repeat1`) because there is less overhead:

```
ghci> last $ take 10000000 $ repeat1 1
1
(2.95 secs, 800443676 bytes)
ghci> last $ take 10000000 $ repeat2 1
1
```

```
(0.11 secs, 280465164 bytes)
```

As another example, consider the following three definitions of the standard prelude function `iterate`:

```
iterate1 f x = x:iterate1 f (f x)
iterate2 f x = xs  where xs = x:map f xs
iterate3 f x = x:map f (iterate3 f x)
```

All three functions have type `(a -> a) -> a -> [a]` and produce an infinite list of the iterates of `f` applied to `x`. The three functions are equal, but the induction principle reviewed earlier doesn't seem to be applicable in proving this assertion because there is no obvious argument on which to perform the induction. More on this later. The first definition is the one used in the standard prelude, but it does not create a cyclic list. The second definition does, and the third is obtained from the second by eliminating the `where` clause. Assuming `f x` can be computed in constant time, the first definition takes $\Theta(n)$ steps to compute the first n elements of the result, but the third takes $\Theta(n^2)$ steps:

```
  iterate3 (2*) 1
= 1:map (2*) (iterate3 (2*1))
= 1:2:map (2*) (map (2*) (iterate3 (2*1)))
= 1:2:4:map (2*) (map (2*) (map (2*) (iterate3 (2*1))))
```

Evaluating the nth element requires n applications of `(2*)`, so it takes $\Theta(n^2)$ to produce the first n elements.

That leaves the second definition. Does it take linear or quadratic time? The evaluation of `iterate2 (2*) 1` proceeds as follows:

```
  xs       where xs = 1:map (2*) xs
= 1:ys     where ys = map (2*) (1:ys)
= 1:2:zs   where zs = map (2*) (2:zs)
= 1:2:4:ts where ts = map (2*) (4:ts)
```

Each element of the result is produced in constant time, so `iterate2 (2*) 1` takes $\Theta(n)$ steps to produce n elements.

Let us now develop a cyclic list to generate an infinite list of all the primes. To start with we define

```
primes     = [2..] \\ composites
composites = mergeAll multiples
multiples  = [map (n*) [n..] | n <- [2..]]
```

where (\\) subtracts one strictly increasing list from another:

```
(x:xs) \\ (y:ys) | x<y   = x:(xs \\ (y:ys))
                 | x==y  = xs \\ ys
                 | x>y   = (x:xs) \\ ys
```

Here, multiples consists of the list of all multiples of 2 from 4 onwards, all multiples of 3 from 9 onwards, all multiples of 4 from 16 onwards, and so on. Merging the list gives the infinite list of all the composite numbers, and taking its complement with respect to [2..] gives the primes. We saw the definition of mergeAll in the previous section.

So far, so good. But the algorithm can be made many times faster by observing that too many multiples are being merged. For instance, having constructed the multiples of 2 there is no need to construct the multiples of 4, or of 6, and so on. What we really would like to do is just to construct the multiples of the primes. That leads to the idea of 'tying the recursive knot' and defining

```
primes = [2..] \\ composites
 where
 composites = mergeAll [map (p*) [p..] | p <- primes]
```

What we have here is a cyclic definition of primes. It looks great, but does it work? Unfortunately, it doesn't: primes produces the undefined list. In order to determine the first element of primes the computation requires the first element of composites, which in turn requires the first element of primes. The computation gets stuck in an infinite loop. To solve the problem we have to pump-prime (!) the computation by giving the computation the first prime explicitly. We have to rewrite the definition as

```
primes = 2:([3..] \\ composites)
 where
 composites = mergeAll [map (p*) [p..] | p <- primes]
```

But this still doesn't produce the primes! The reason is a subtle one and is quite hard to spot. It has to do with the definition

```
mergeAll = foldr1 xmerge
```

The culprit is the function foldr1. Recall the Haskell definition:

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f [x]    = x
foldr1 f (x:xs) = f x (foldr1 xs)
```

The order of the two defining equations is significant. In particular,

```
foldr1 f (x:undefined) = undefined
```

because the list argument is first matched against `x:[]`, causing the result to be undefined. That means

```
mergeAll [map (p*) [p..] | p <- 2:undefined] = undefined
```

What we wanted was

```
mergeAll [map (p*) [p..] | p <- 2:undefined] = 4:undefined
```

To effect this change we have to define `mergeAll` differently:

```
mergeAll (xs:xss) = xmerge xs (mergeAll xss)
```

Now we have

```
  mergeAll [map (p*) [p..] | p <- 2:undefined]
= xmerge (map (2*) [2..]) undefined
= xmerge (4:map (2*) [3..]) undefined
= 4:merge (map (2*) [3..]) undefined
= 4:undefined
```

This version of `mergeAll` behaves differently on finite lists from the previous one. Why?

With this final change we claim that `primes` does indeed get into gear and produces the primes. But how can the claim be proved? To answer this question we need to know something about the semantics of recursively defined functions and other values in Haskell, and how infinite lists are defined as limits of their partial approximations.

9.3 Infinite lists as limits

In mathematics, certain values are defined as *limits* of infinite sequences of approximations of simpler values. For example, the irrational number

$$\pi = 3.14159265358979323846\cdots$$

can be defined as the limit of the infinite sequence of rational approximations

$$3,\ 3.1,\ 3.14,\ 3.141,\ 3.1415,\ \ldots$$

The first element of the sequence, 3, is a fairly crude approximation to π. The next element, 3.1, is a little better; 3.14 is better still, and so on.

Similarly, an infinite list can also be regarded as the limit of a sequence of approximations. For example, the infinite list [1..] is the limit of the infinite sequence of partial lists

$$\bot,\ 1:\bot,\ 1:2:\bot,\ 1:2:3:\bot,\ \ldots$$

Again, the sequence consists of better and better approximations to the intended limit. The first term, $\bot$, is the undefined element, and thus a very crude approximation: it tells us nothing about the limit. The next term, $1:\bot$, is a slightly better approximation: it tells us that the limit is a list whose first element is 1, but says nothing about the rest of the list. The following term, $1:2:\bot$, is a little better still, and so on. Each successively better approximation is derived by replacing $\bot$ with a more defined value, and thus gives more information about the limit.

Here is another sequence of approximations whose limit is [1..]:

$$\bot,\ 1:2:\bot,\ 1:2:3:4:\bot,\ 1:2:3:4:5:6:\bot,\ \ldots$$

This sequence is a subsequence of the one above but it converges to the same limit.

Here is a sequence of approximations that does not converge to a limit:

$$\bot,\ 1:\bot,\ 2:1:\bot,\ 3:2:1:\bot,\ \ldots$$

The problem with this sequence is that it gives conflicting information: the second term says that the limit begins with 1. However, the third term says that the limit begins with 2, and the fourth term says that it begins with 3, and so on. No approximation tells us anything about the intended limit and the sequence does not converge.

It should not be thought that the limit of a sequence of lists is necessarily infinite. For example, the sequence

$$\bot,\quad 1:\bot,\quad 1:[\,],\quad 1:[\,],\quad \ldots$$

in which every element after the first two is [1], is a perfectly valid sequence with limit [1]. Similarly,

$$\bot,\quad 1:\bot,\quad 1:2:\bot,\quad 1:2:\bot,\quad \ldots$$

is a sequence with limit $1:2:\bot$. Finite and partial lists are limits of sequences possessing only a finite number of distinct elements.

The way to formalise the property that an infinite sequence of partial lists converges to a limit is to introduce the notion of an *approximation ordering* $\sqsubseteq$ on the

elements of each type. The assertion $x \sqsubseteq y$ means that x is an approximation to y. The ordering $\sqsubseteq$ will be reflexive ($x \sqsubseteq x$), transitive ($x \sqsubseteq y$ and $y \sqsubseteq z$ implies $x \sqsubseteq z$), and anti-symmetric ($x \sqsubseteq y$ and $y \sqsubseteq x$ implies $x = y$). However, it is not the case that every pair of elements have to be comparable by $\sqsubseteq$. Thus $\sqsubseteq$ is what is known as a *partial* ordering. Note that $\sqsubseteq$ is a mathematical operator (like $=$), and not a Haskell operator returning boolean results.

The approximation ordering for numbers, booleans, characters and any other enumerated type, is defined by

$$x \sqsubseteq y \equiv (x = \bot) \vee (x = y).$$

The first clause says that $\bot$ is an approximation to everything. In other words, $\bot$ is the *bottom* element of the ordering. This explains why $\bot$ is pronounced 'bottom'. The value $\bot$ is the bottom element of $\sqsubseteq$ for every type. The above ordering is *flat*. With a flat ordering one either knows everything there is to know about a value, or one knows absolutely nothing.

The approximation ordering on the type (a,b) is defined by $\bot \sqsubseteq (x,y)$ and

$$(x,y) \sqsubseteq (x',y') \equiv (x \sqsubseteq x') \wedge (y \sqsubseteq y').$$

The occurrences of $\sqsubseteq$ on the right refer to the orderings on the types a and b, respectively. The ordering $\sqsubseteq$ on (a,b) is not flat, even when the component orderings are. For example, in `(Bool,Bool)` we have the following chain of distinct elements:

$$\bot \sqsubseteq (\bot,\bot) \sqsubseteq (\bot,\texttt{False}) \sqsubseteq (\texttt{True},\texttt{False}).$$

Note that in Haskell the pair $(\bot,\bot)$ is distinct from $\bot$:

```
ghci> let f (a,b) = 1
ghci> f (undefined,undefined)
1
ghci> f undefined
*** Exception: Prelude.undefined
```

The ordering $\sqsubseteq$ on `[a]` is defined by $\bot \sqsubseteq xs$ and `(x:xs)` $\not\sqsubseteq$ `[]` and

$$\texttt{[]} \sqsubseteq \texttt{xs} \equiv \texttt{xs} = \texttt{[]},$$
$$\texttt{(x:xs)} \sqsubseteq \texttt{(y:ys)} \equiv (\texttt{x} \sqsubseteq \texttt{y}) \wedge (\texttt{xs} \sqsubseteq \texttt{ys}).$$

These equations should be read as an inductive definition of a mathematical assertion, not as a Haskell definition. The second condition says that `[]` approximates only itself, and the third condition says that `(x:xs)` is an approximation to `(y:ys)` if and only if `x` is an approximation to `y` and `xs` is an approximation to `ys`. The first

occurrence of $\sqsubseteq$ on the right-hand side refers to the approximation ordering on the type `a`.

As two examples, we have

$$[1,\bot,3] \sqsubseteq [1,2,3] \quad \text{and} \quad 1:2:\bot \sqsubseteq [1,2,3].$$

However, $1:2:\bot$ and $[1,\bot,3]$ are not related by $\sqsubseteq$.

The approximation ordering for each type T is assumed to have another property in addition to those described above: each *chain* of approximations $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$ has to possess a limit which is also a member of T. The limit, which we denote by $\lim_{n\to\infty} x_n$, is defined by two conditions:

1. $x_n \sqsubseteq \lim_{n\to\infty} x_n$ for all n. This condition states that the limit is an *upper bound* on the sequence of approximations.
2. If $x_n \sqsubseteq y$ for all n, then $\lim_{n\to\infty} x_n \sqsubseteq y$. This condition states that the limit is the *least* upper bound.

The definition of the limit of a chain of approximations applies to every type. Partial orderings possessing this property are called *complete*, and every Haskell type is a complete partial ordering (CPO for short). In particular, the property, introduced in Chapter 6, of a mathematical assertion P being chain complete can now be formalised as

$$(\forall n : P(x_n)) \Rightarrow P(\lim_{n\to\infty} x_n).$$

In words, P holds in the limit if it holds for each approximation to the limit.

For lists there is a useful Haskell function `approx`, which produces approximations to a given list. The definition is

```
approx :: Integer -> [a] -> [a]
approx n []     | n>0 = []
approx n (x:xs) | n>0 = x:approx (n-1) xs
```

The definition of `approx` is very similar to that of `take` except that, by case exhaustion, we have `approx 0 xs = undefined` for all `xs`. For example,

```
approx 0 [1] = undefined
approx 1 [1] = 1:undefined
approx 2 [1] = 1:[]
```

The crucial property of `approx` is that

$$\lim_{n\to\infty} \texttt{approx n xs} = \texttt{xs}$$

for all lists `xs`, finite, partial or infinite. The proof, an induction on `xs`, is left as an exercise.

It follows that if `approx n xs` = `approx n ys` for all natural numbers `n`, then `xs` = `ys`. Thus we can prove that

```
iterate f x = x:map f (iterate f x)
```

by showing

```
approx n (iterate f x) = approx n (x:map f (iterate f x))
```

for all natural numbers `n`. And, of course, we can use induction over the natural numbers to establish this fact. The details are left as an easy exercise.

As another example, consider the value `primes` defined in the previous section. Suppose we define

```
prs n = approx n primes
```

We would like to show that `prs n` $= p_1 : p_2 : \cdots p_n : \bot$, where p_j is the jth prime. We claim that

```
prs n = approx n (2:([3..] \\ crs n))
crs n = mergeAll [map (p*) [p..] | p <- prs n]
```

Given this, it is sufficient to show that `crs n` $= c_1 : c_2 : \cdots c_m : \bot$, where c_j is the jth composite number (so $c_1 = 4$) and $m = p_n^2$. Then the proof is completed by using the fact that $p_{n+1} < p_n^2$, which is a non-trivial result in Number Theory. Details are in the exercises.

Computable functions and recursive definitions

One can describe many functions in mathematics, but only some of them are computable. There are two properties of computable functions not shared by arbitrary functions. Firstly, a computable function f is *monotonic* with respect to the approximation ordering. In symbols,

$$x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$$

for all x and y. Roughly speaking, monotonicity states that the more information you supply about the argument, the more information you get as a result. Secondly, a computable function f is *continuous*, which means that

$$f\,(\lim_{n\to\infty} x_n) = \lim_{n\to\infty} f(x_n)$$

for all chains of approximations $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$. Roughly speaking, continuity states that there are no surprises on passing to the limit.

Continuity appears similar to chain completeness but differs in two respects. One is that the chain completeness of P does not imply the converse property that if P is false for all approximations, then P is false for the limit. In other words, it does not imply that $\neg P$ is chain complete. Secondly, P is a mathematical assertion, not a Haskell function returning a boolean value.

Although we won't prove it, every monotonic and continuous function f has a *least fixed point*. A fixed point of a function f is a value x such that $f(x) = x$. And x is a least fixed point if $x \sqsubseteq y$ for any other fixed point y. The least fixed point of a monotonic and continuous function f is given by $\lim_{n\to\infty} x_n$ where $x_0 = \bot$ and $x_{n+1} = f(x_n)$. In functional programming, recursive definitions are interpreted as least fixed points.

Here are three examples. Consider the definition `ones = 1:ones`. This definition asserts that `ones` is a fixed point of the function `(1:)`. Haskell interprets it as the least fixed point, so $\texttt{ones} = \lim_{n\to\infty} \texttt{ones}_n$, where $\texttt{ones}_0 = \bot$ and $\texttt{ones}_{n+1} = \texttt{1:ones}_n$. It is easy to see that $\texttt{ones}_n$ is the partial list consisting of n ones, so the limit is indeed an infinite list of ones.

Second, consider the factorial function

```
fact n = if n==0 then 1 else n*fact (n-1)
```

We can rewrite this definition in the equivalent form

```
fact = (\f n -> if n==0 then 1 else n*f(n-1)) fact
```

Again, this definition asserts that `fact` is a fixed point of a function. Here we have

$$\begin{aligned}
&\texttt{fact}_0\ \texttt{n} = \bot \\
&\texttt{fact}_1\ \texttt{n} = \texttt{if n==0 then 1 else } \bot \\
&\texttt{fact}_2\ \texttt{n} = \texttt{if n<=1 then 1 else } \bot
\end{aligned}$$

and so on. The value of $\texttt{fact}_k$ `n` is the factorial of `n` if `n` is less than k, and $\bot$ otherwise.

Finally, consider the list `primes` once again. Here we have

$$\begin{aligned}
&\texttt{primes}_0 \quad = \quad \bot \\
&\texttt{primes}_{n+1} \quad = \quad \texttt{2:([3..] \textbackslash\textbackslash} \\
&\qquad\qquad\quad\ \ \texttt{mergeAll [map (p*) [p..] | p <- primes}_n\texttt{]})
\end{aligned}$$

It is not the case that $\texttt{primes}_n = \texttt{approx n primes}$. In fact,

$$\begin{aligned}\texttt{primes}_1 &= 2 : \bot\\ \texttt{primes}_2 &= 2 : 3 : \bot\\ \texttt{primes}_3 &= 2 : 3 : 5 : 7 : \bot\\ \texttt{primes}_4 &= 2 : 3 : 5 : 7 : \cdots : 47 : \bot\end{aligned}$$

The partial list $\texttt{primes}_2$ produces all the primes less than 4, $\texttt{primes}_3$ all the primes less than 9, and $\texttt{primes}_4$ all the primes less than 49. And so on.

9.4 Paper–rock–scissors

Our next example of infinite lists is entertaining as well as instructive. Not only does it introduce the idea of using potentially infinite lists to model a sequence of interactions between processes, it also provides another concrete illustration of the necessity for formal analysis.

The paper–rock–scissors game is a familiar one to children, though it is known by different names in different places. The game is played by two people facing one another. Behind their backs, each player forms a hand in the shape of either a rock (a clenched fist), a piece of paper (a flat palm) or a pair of scissors (two fingers extended). At a given instant, both players bring their hidden hand forward. The winner is determined by the rule 'paper wraps rock, rock blunts scissors, and scissors cut paper'. Thus, if player 1 produces a rock and player 2 produces a pair of scissors, then player 1 wins because rock blunts scissors. If both players produce the same object, then the game is a tie and neither wins. The game continues in this fashion for a fixed number of rounds agreed in advance.

Our objective in this section is to write a program to play and score the game. We begin by introducing the types

```
data Move  = Paper | Rock | Scissors
type Round = (Move,Move)
```

To score a round we define

```
score :: Round -> (Int,Int)
score (x,y) | x `beats` y  = (1,0)
            | y `beats` x  = (0,1)
            | otherwise    = (0,0)
```

where

```
Paper `beats` Rock     = True
Rock `beats` Scissors  = True
Scissors `beats` Paper = True
_ `beats` _            = False
```

Each player in the game will be represented by a certain strategy. For instance, one simple strategy is, after the first round, always to produce what the opposing player showed in the previous round. This strategy will be called `copy`. Another strategy, which we will call `smart`, is to determine a move by analysing the number of times the opponent has produced each of the three possible objects, and calculating an appropriate response based on probabilities.

We will consider the details of particular strategies, and how they can be represented, in a moment. For now, suppose the type `Strategy` is given in some way. The function

```
rounds :: (Strategy,Strategy) -> [Round]
```

takes a pair of strategies and returns the infinite list of rounds that ensue when each player follows his or her assigned strategy. The function

```
match :: Int -> (Strategy,Strategy) -> (Int,Int)
match n = total . map score . take n . rounds
  where total rs = (sum (map fst rs),sum (map snd rs))
```

determines the total score after playing a given number of rounds.

The instructive aspect of the game is how to represent strategies. We are going to consider two ways, calling them `Strategy1` and `Strategy2`. The obvious idea is to take

```
type Strategy1 = [Move] -> Move
```

Here, a strategy is a function which takes the (finite) list of moves made by the opponent so far and returns an appropriate move for the subsequent round. For efficiency in processing lists, we suppose that the list of moves is given in reverse order, with the last move first.

For example, the `copy1` strategy is implemented by

```
copy1 :: Strategy1
copy1 ms = if null ms then Rock else head ms
```

The first move is an arbitrary choice of `Rock`, The second strategy `smart1` is implemented by

```
smart1 :: Strategy1
smart1 ms = if null ms then Rock
            else pick (foldr count (0,0,0) ms)

count :: Move -> (Int,Int,Int) -> (Int,Int,Int)
count Paper (p,r,s)    = (p+1,r,s)
count Rock  (p,r,s)    = (p,r+1,s)
count Scissors (p,r,s) = (p,r,s+1)

pick :: (Int,Int,Int) -> Move
pick (p,r,s)
     | m < p     = Scissors
     | m < p+r   = Paper
     | otherwise = Rock
     where m = rand (p+r+s)
```

This strategy counts the number of times each move has been made, and uses the results to pick a move. The value of `rand` applied to n is some integer m in the range $0 \leq m < n$. (Note that `rand` is never applied to the same integer.) Thus the choice of move depends on whether m falls in one of the three ranges

$$0 \leq m < p \quad \text{or} \quad p \leq m < p+r \quad \text{or} \quad p+r \leq m < p+r+s.$$

For example, if p is large, then `Scissors` will be chosen with high probability (because scissors cuts paper); and if r is large, then `Paper` will be chosen with high probability (because paper wraps rock); and so on.

To define `rand` we can make use of two functions in the library `System.Random`:

```
rand :: Int -> Int
rand n = fst $ randomR (0,n-1) (mkStdGen n)
```

The function `mkStdGen` takes an integer and returns a random number generator, likely to be different for different integers. The choice of argument to `mkStdGen` is arbitrary, and we have simply chosen `n`. The function `randomR` takes a range (a,b) and a random number generator, and returns a pseudo-random integer r in the range $a \leq r \leq b$ and a new random number generator.

We can now define `rounds1`:

```
rounds1 :: (Strategy1,Strategy1) -> [Round]
rounds1 (p1,p2)
         = map head $ tail $ iterate (extend (p1,p2)) []
extend (p1,p2) rs = (p1 (map snd rs),p2 (map fst rs)):rs
```

The function `extend` adds a new pair of moves to the front of the list of existing rounds, and `rounds1` generates the infinite list of rounds by repeatedly applying `extend` to the initially empty list. It is more efficient to add something to the front of a list than to the end, which is why we keep the list of moves in reverse order.

Nevertheless `rounds1` is inefficient. Suppose a strategy takes time proportional to the length of its input to compute the next move. It follows that `extend` takes $\Theta(n)$ steps to update a game of n rounds with a new round. Therefore, it takes $\Theta(N^2)$ steps to compute a game of N rounds.

For comparison, let's consider another way we might reasonably represent strategies. This time we take

```
type Strategy2 = [Move] -> [Move]
```

In the new representation, a strategy is a function that takes the potentially infinite list of moves made by the opponent and returns the potentially infinite list of replies. For example, the copy strategy is now implemented by

```
copy2 :: Strategy2
copy2 ms = Rock:ms
```

This strategy returns `Rock` the first time, and thereafter returns just the move made by the opponent in the previous round. The smart strategy is reprogrammed as

```
smart2 :: Strategy2
smart2 ms = Rock:map pick (stats ms)
 where stats = tail . scanl (flip count) (0,0,0)
```

The function `stats` computes the running counts of the three possible moves. This strategy, like `copy2`, is also efficient in that it produces each successive output with constant delay.

With this new model of strategies we can redefine the function `rounds`:

```
rounds2 :: (Strategy2,Strategy2) -> [Round]
rounds2 (p1,p2) = zip xs ys
                  where xs = p1 ys
                        ys = p2 xs
```

Here, `xs` is the list of replies computed by the first player in response to the list `ys` which, in turn, is the list of replies made by the second player in response to the list of moves `xs`. Thus `rounds2` is defined by two cyclic lists and we are obliged to show that it does indeed generate an infinite list of well-defined moves. More on this below. If the two players do encapsulate legitimate strategies, then

`rounds2` computes the first n moves of the game in $\Theta(n)$ steps, assuming that both players compute each new move with constant delay. Thus the second method for modelling strategies leads to a more efficient program than the earlier one.

Unfortunately, there is a crucial flaw with the second representation of strategies: it offers no protection against someone who cheats! Consider the strategy

```
cheat ms = map trump ms

trump Paper    = Scissors
trump Rock     = Paper
trump Scissors = Rock
```

The first reply of `cheat` is the move guaranteed to beat the opponent's first move; similarly for subsequent moves. To see that `cheat` cannot be prevented from subverting the game, consider a match in which it is played against `copy2`, and let `xs = cheat ys` and `ys = copy2 xs`. The lists `xs` and `ys` are the limits of the two chains $\{\texttt{xs}_n \mid 0 \leq n\}$ and $\{\texttt{ys}_n \mid 0 \leq n\}$, where $\texttt{xs}_0$ = `⊥` and $\texttt{xs}_{n+1}$ = `cheat` $\texttt{ys}_n$, and $\texttt{ys}_0$ = `⊥` and $\texttt{ys}_{n+1}$ = `copy2` $\texttt{xs}_n$. Now, we have

```
xs1 = cheat ⊥           = ⊥
ys1 = copy2 ⊥           = Rock: ⊥
xs2 = cheat (Rock: ⊥)   = Paper: ⊥
ys2 = copy2 ⊥           = Rock: ⊥
xs3 = cheat (Rock: ⊥)   = Paper: ⊥
ys3 = copy2 (Paper: ⊥)  = Rock:Paper: ⊥
```

Continuing in this way, we see that the limits of these sequences are indeed infinite lists of well-defined moves. Moreover, `cheat` always triumphs. Another cheating strategy is given by

```
devious :: Int -> Strategy2
devious n ms = take n (copy2 ms) ++ cheat (drop n ms)
```

This strategy behaves like copy for `n` moves then starts to cheat.

Can we find a way to protect against cheats? To answer this question, we need to take a closer look at what constitutes an honest strategy. Informally speaking, a strategy is honest if its first move is computed in the absence of any information about the opponent's first move, the second move is computed without any information about the opponent's second move, and so on. Moreover, each of these moves should be well-defined, given that the opponent's moves are well-defined. More precisely, let $wdf(n, ms)$ denote the assertion that the first n elements in the

(possibly partial) list of moves *ms* are well-defined. Then a strategy f is *honest* if

$$wdf(n, ms) \Rightarrow wdf(n+1, f(ms))$$

for all n and *ms*. It is easy to show that `copy2` is honest. On the other hand, `cheat` is not honest because $wdf(0, \bot)$ is true but $wdf(1, \texttt{cheat}\ \bot)$ is false. The strategy `dozy`, where

```
dozy ms = repeat undefined
```

is also dishonest according to this definition although it doesn't actually cheat.

Having identified the source of criminal or lackadaisical behaviour, can we ensure that only honest strategies are admitted to the game? The answer is a qualified yes: although it is not possible for a mechanical evaluator to recognise cheating (in the same way that it is not possible to recognise $\bot$, or strategies that do not return well-defined moves), it is possible to define a function `police` so that if `p` is an honest player and `ms` is an infinite sequence of well-defined moves, then `police p ms = p ms`. On the other hand, if `p` is dishonest at some point, then the game ends at that point in $\bot$. Operationally speaking, `police` works by forcing `p` to return the first (well-defined!) element of its output before it gives `p` the first element of its input. Similarly for the other elements. The definition is

```
police p ms = ms'  where ms' = p (synch ms ms')
synch (x:xs) (y:ys) = (y `seq` x):synch xs ys
```

Recall from Chapter 7 that `x `seq` y` evaluates `x` before returning the value of `y`. The proof that this implementation meets its specification is rather involved, so we are not going into details. It follows from the above analysis that to prevent cheating we must rewrite the definition of `rounds2` to read

```
rounds2 (p1,p2) = zip xs ys
                  where xs = police p1 ys
                        ys = police p2 xs
```

9.5 Stream-based interaction

In the paper–rock–scissors game we modelled interaction by a function that took an infinite list of moves and returned a similar list. The same idea can be used to provide a simple model of input–output interaction. It's called *stream-based* interaction because infinite lists are also called streams. Haskell provides a function

```
interact :: ([Char] -> [Char]) -> IO ()
```

for interacting with the world. The argument to `interact` is a function that takes a potentially infinite list of characters from the standard input channel, and returns a potentially infinite list of characters to be typed on the standard output channel.

For example,

```
ghci> import Data.Char
ghci> interact (map toUpper)
hello world!
HELLO WORLD!
Goodbye, cruel world!
GOODBYE, CRUEL WORLD!
{Interrupted}
```

We imported the library `Data.Char` to make `toUpper` available, and then created an interaction that capitalised each letter. Each time a line of input was typed (and echoed) the interaction produced the same line in capital letters. The process continues until we interrupt it.

We can also design an interactive program that terminates. For example,

```
interact (map toUpper . takeWhile (/= '.'))
```

will interact as above but terminate as soon as a line containing a period is typed:

```
ghci> interact (map toUpper . takeWhile (/= '.'))
Goodbye. Forever
GOODBYE
```

Finally, here is a stand-alone program that takes a literate Haskell file as input and returns a file in which all nonempty lines not beginning with `>` are removed. The remaining lines are modified by removing the `>` character, so the result is a legitimate `.hs` file (a Haskell script not using the literate style):

```
main    = interact replace
replace = unlines . map cleanup . filter code . lines
code xs = null xs || head xs == '>'
cleanup xs = if null xs then [] else tail xs
```

The program is the computation associated with the identifier `main`, and there always has to be a definition associated with this name if we want to compile a program. The function `lines` splits a text into lines, and `unlines` reassembles the text by putting a single newline between lines. If we store the program in `lhs2hs.lhs`, we can compile it and then run it:

```
$ ghc lhs2hs.lhs
$ lhs2hs <myscript.lhs >myscript.hs
```

In the second line, the input is taken from `myscript.lhs` and the output is directed to `myscript.hs`.

Stream-based interaction was the main method for interacting with the outside world in early versions of Haskell. However, the model presented above is too simple for most practical purposes. In a serious application one wants to do other things than reading and printing characters to a screen. For example, one also wants to open and read files, to write to or delete files, and in general to interact with all the mechanisms that are available in the world outside the confines of a functional programming language. Interaction takes place in time, and the order in which events occur has to be managed correctly by the programmer. In the stream-based approach, this ordering of events is represented by the order of the elements in a list; in other words, it is represented in the data and not reflected primarily in the way the program is composed. In the following chapter we will consider another approach to interaction, indeed, a general method for writing programs that have to control an orderly sequence of events. In this approach, the order is made explicit in the way the program is composed.

9.6 Doubly-linked lists

We end with another application of cyclic lists. Imagine reading a book consisting of a nonempty list of pages. To navigate around the book we need some way of moving on to the next page and moving back to the previous page. Other navigation tools would be useful, but we'll stick with these two. Here is an interactive session with a particularly boring book `book` consisting of three pages:

```
ghci> start book
"Page 1"
ghci> next it
"Page 2"
ghci> prev it
"Page 1"
ghci> next it
"Page 2"
ghci> next it
"Page 3"
```

In GHCi the variable `it` is bound to the expression just typed at the prompt. We started a book and what was printed was the first page. We turned to the next page, and then returned to the previous one. The interesting question is what should happen when we turn to the next page after the last one. Should the navigation report an error, just deliver the last page again or go to the first page? Suppose we decide on the last alternative, namely that the next page after the last one should be the first page, and the previous page before the first one should be the last page. In other words, our book is an instance of a *cyclic doubly-linked list*.

Here is the relevant datatype declaration:

```
data DList a = Cons a (DList a) (DList a)

elem :: DList a -> a
elem (Cons a p n) = a

prev,next :: DList a -> DList a
prev (Cons a p n) = p
next (Cons a p n) = n
```

We print a doubly-linked list by displaying the current entry:

```
instance Show a => Show (DList a)
  where show d = show (elem d)
```

Our book is then a list `[p1,p2,p3]` of three pages, where

```
p1 = Cons "Page 1" p3 p2
p2 = Cons "Page 2" p1 p3
p3 = Cons "Page 3" p2 p1
```

This example suggests that the function `mkCDList :: [a] -> DList a` for converting a (nonempty) list `as` into a doubly-linked list can be specified as the first element in a finite list `xs` of doubly-linked lists satisfying the following three properties:

```
map elem xs = as
map prev xs = rotr xs
map next xs = rotl xs
```

Here, `rotr` and `rotl` (short for *rotate right* and *rotate left*), are defined by

```
rotr xs = [last xs] ++ init xs
rotl xs = tail xs ++ [head xs]
```

Observe now that for any list `xs` of doubly-linked lists we have

```
xs = zipWith3 Cons
     (map elem xs) (map prev xs) (map next xs)
```

where `zipWith3` is like `zipWith` except that it takes three lists instead of two. The standard prelude definition is:

```
zipWith3 f (x:xs) (y:ys) (z:zs)
    = f x y z : zipWith3 f xs ys zs
zipWith3 _ _ _ _ = []
```

We will see another definition in a moment. We can prove the claim above by induction. It clearly holds for the undefined and empty lists. For the inductive case we reason:

```
  x:xs
=   {since xs is a doubly-linked list}
  Cons (elem x) (prev x) (next x):xs
=   {induction}
  Cons (elem x) (prev x) (next x):
  (zipWith3 Cons
  (map elem xs) (map prev xs) (map next xs))
=   {definition of zipWith3 and map}
  zipWith3 Cons
  (map elem (x:xs)) (map prev (x:xs)) (map next (x:xs)
```

Putting this result together with our specification of doubly-linked lists, we arrive at

```
mkCDList as = head xs
  where xs = zipWith3 Cons as (rotr xs) (rotl xs)
```

This definition involves a cyclic list `xs`. Does it work? The answer is: No, it doesn't. The reason is that `zipWith3` as defined above is too eager. We need to make it lazier by not demanding the values of the second two lists until they are really needed:

```
zipWith3 f (x:xs) ys zs
    = f x (head ys) (head zs):
      zipWith3 f xs (tail ys) (tail zs)
zipWith3 _ _ _ _ = []
```

An equivalent way to define this function is to make use of Haskell's *irrefutable patterns*:

```
zipWith3 f (x:xs) ~(y:ys) ~(z:zs)
    = f x y z : zipWith3 f xs ys zs
zipWith3 _ _ _ _ = []
```

An irrefutable pattern is introduced using a tilde, and `~(x:xs)` is matched lazily, meaning that no matching is actually performed until either `x` or `xs` is needed.

Just to convince ourselves that the above definition of `mkCDList` with the revised definition of `zipWith3` does make progress, let $\texttt{xs}_0 = \bot$ and

$$\texttt{xs}_{n+1} = \texttt{zipWith3 Cons "A" (rotr xs}_n\texttt{) (rotl xs}_n\texttt{)}$$

Then $\texttt{xs}_1$ is given by

$$\begin{aligned} &\texttt{zipWith3 Cons "A"} \ \bot \ \bot \\ =\ &\texttt{[Cons 'A'} \ \bot \ \bot\texttt{]} \end{aligned}$$

and $\texttt{xs}_2$ by

$$\begin{aligned} &\texttt{zipWith3 Cons "A"} \\ &\texttt{[Cons 'A'} \ \bot \ \bot\texttt{] [Cons 'A'} \ \bot \ \bot\texttt{]} \\ =\ &\texttt{[Cons 'A' (Cons 'A'} \ \bot \ \bot\texttt{) (Cons 'A'} \ \bot \ \bot\texttt{)]} \end{aligned}$$

and so on.

9.7 Exercises

Exercise A

Given three lists `xs`, `ys` and `zs` in strictly increasing order, we have

```
merge (merge xs ys) zs} = merge xs (merge ys zs)
```

Thus `merge` is associative. Assuming in addition that the first elements of `xs`, `ys` and `zs` are in strictly increasing order, we also have

```
xmerge (xmerge xs ys) zs = xmerge xs (xmerge ys zs)
```

Does it follow that in the expression `foldr1 xmerge multiples` we could replace `foldr1` by `foldl1`?

Exercise B

The standard prelude function `cycle :: [a] -> [a]` takes a list `xs` and returns a list consisting of an infinite number of repetitions of the elements of `xs`. If `xs` is the empty list, then `cycle []` returns an error message. For instance

```
cycle "hallo" = "hallohallohallo...
```

Define `cycle` using a cyclic list. Ensure that your definition works on empty, finite and infinite lists.

Exercise C

The fibonacci function is defined by

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Write down a one-line definition of the list `fibs` that produces the infinite list of Fibonacci numbers.

Exercise D

A well-known problem, due to the mathematician W.R. Hamming, is to write a program that produces an infinite list of numbers with the following properties: (i) the list is in strictly increasing order; (ii) the list begins with the number 1; (iii) if the list contains the number x, then it also contains the numbers $2x$, $3x$ and $5x$; (iv) the list contains no other numbers. Thus, the required list begins with the numbers

$$1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, \ldots$$

Write a definition of `hamming` that produces this list.

Exercise E

Prove that `approx n xs` $\sqsubseteq$ `xs` for all `n`. Now prove that if `approx n xs` $\sqsubseteq$ `ys` for all `n`, then `xs` $\sqsubseteq$ `ys`. Hence conclude that

$$\lim_{n \to \infty} \texttt{approx n xs} = \texttt{xs}.$$

Exercise F

Give a counter-example to the claim that `xs=ys` if `xs!!n=ys!!n` for all `n`.

Exercise G

Prove that `iterate f x = x: map f (iterate f x)`.

Exercise H

In the definition of `primes` as a cyclic list, could we have defined

```
mergeAll = foldr xmerge []
```

as an alternative to the definition in the text?

Exercise I

Recall that

```
prs n = approx n (2:([3..] \\ crs n))
crs n = mergeAll [map (p*) [p..] | p <- prs n]
```

Given that `prs n`=$p_1 : p_2 : \cdots p_n : \bot$, where p_j is the jth prime, sketch how to show that `crs n`=$c_1 : c_2 : \cdots c_m : \bot$, where c_j is the jth composite number (so c_1=4) and m=p_n^2. Hence show that `primes` does produce the infinite list of primes.

We said in the text that it is not the case that the nth approximation $\texttt{primes}_n$ of `primes` is equal to `approx n primes`. In fact

$$\texttt{primes}_4 = 2 : 3 : 5 : 7 : \cdots : 47 : \bot$$

What list does $\texttt{primes}_5$ produce?

Exercise J

Another way of generating the primes is known as the *Sieve of Sundaram*, after its discoverer S.P. Sundaram in 1934:

```
primes   = 2:[2*n+1 | n <- [1..] \\ sundaram]
sundaram = mergeAll [[i+j+2*i*j | j <- [i..]] | i <- [1..]]
```

To show that the list comprehension in the definition of `primes` generates exactly the odd primes, it is sufficient to prove that the term `2*n+1` is never composite, which is to say that it never factorises into `(2*i+1)*(2*j+1)` for positive integers `i` and `j`. Why is this so?

Exercise K

Is the function f, defined by $f(\bot) = 0$ and $f(x) = 1$ for $x \neq \bot$, computable? How about the function that returns $\bot$ on all finite or partial lists, and 1 on all infinite lists?

Exercise L

By definition, a *torus* is a doubly-cyclic, doubly-doubly-linked list. It is a cyclic doubly-linked list in the left/right direction, and also in the up/down direction. Given a matrix represented as a list of length m of lists, all of length n, construct a definition of

```
mkTorus :: Matrix a -> Torus a
```

where

```
data Torus a = Cell a (Torus a) (Torus a)
                      (Torus a) (Torus a)
elem  (Cell a u d l r) = a
up    (Cell a u d l r) = u
down  (Cell a u d l r) = d
left  (Cell a u d l r) = l
right (Cell a u d l r) = r
```

That looks tricky, but the answer is short enough to be tweeted.

9.8 Answers

Answer to Exercise A

No, since `foldl1 f xs = undefined` for any infinite list `xs`.

Answer to Exercise B

The definition is

```
cycle [] = error "empty list"
cycle xs = ys  where ys = xs ++ ys
```

Note that if `xs` is infinite, then `xs ++ ys` = `xs`, so `cycle` is the identity function on infinite lists.

Answer to Exercise C

The one-liner is:

```
fibs :: [Integer]
fibs = 0:1:zipWith (+) fibs (tail fibs)
```

Answer to Exercise D

```
hamming :: [Integer]
hamming = 1: merge (map (2*) hamming)
               (merge (map (3*) hamming)
                      (map (5*) hamming))
```

Answer to Exercise E

The proof of `approx n xs` $\sqsubseteq$ `xs` is by induction on `n`. The base case is easy but the induction step involves a sub-induction over `xs`. The base cases (the empty list and the undefined list) of the sub-induction are easy and the inductive case is

$$\begin{aligned} & \texttt{approx (n+1) (x:xs)} \\ = \quad & \{\text{definition}\} \\ & \texttt{x:approx n xs} \\ \sqsubseteq \quad & \{\text{induction and monotonicity of (x:)}\} \\ & \texttt{x:xs}. \end{aligned}$$

The proof of

$$(\forall \texttt{n} : \texttt{approx n xs} \sqsubseteq \texttt{ys}) \Rightarrow \texttt{xs} \sqsubseteq \texttt{ys}$$

is by induction on `xs`. The claim is immediate for the undefined and empty lists, and for the inductive case we have

$$\begin{aligned} & (\forall \texttt{n} : \texttt{approx n (x:xs)} \sqsubseteq \texttt{ys}) \\ \Rightarrow\ & \texttt{xs} \sqsubseteq \texttt{head ys} \wedge (\forall \texttt{n} : \texttt{approx n xs} \sqsubseteq \texttt{tail ys}) \end{aligned}$$

by the definitions of `approx` and the approximation ordering on lists. By induction we therefore have

$$\texttt{x:xs} \sqsubseteq \texttt{head ys:tail ys} = \texttt{ys}.$$

It follows that

$$\lim_{n \to \infty} \texttt{approx n xs} = \texttt{xs}$$

by the definition of limit.

Answer to Exercise F

The two lists `repeat undefined` and `undefined` are not equal, but

$$\texttt{(repeat undefined)!!n} = \texttt{undefined!!n}$$

for all `n` because both sides are $\bot$.

Answer to Exercise G

We have to show that

```
approx n (iterate f x) = approx n (x:map f (iterate f x))
```

for all natural numbers `n`. This claim follows from

```
approx n (iterate f (f x))
                  = approx n (map f (iterate f x))
```

which we establish by induction on `n`. For the inductive step we simplify each side. For the left-hand side:

```
  approx (n+1) (iterate f (f x))
=   {definition of iterate}
  approx (n+1) (f x:iterate f (f (f x)))
=   {definition of approx}
  f x: approx n (iterate f (f (f x)))
=   {induction}
  f x: approx n (map f (iterate f (f x)))
```

For the right-hand side:

```
  approx (n+1) (map f (iterate f x))
=   {definition of iterate and map}
  approx (n+1) (f x:map f (iterate f (f x)))
=   {definition of approx}
  f x: approx n (map f (iterate f (f x)))
```

Answer to Exercise H

Yes, since

```
foldr xmerge [] (xs:undefined) = xmerge xs undefined
```

and the right-hand side begins with the first element of `xs`.

Answer to Exercise I

The proof is by induction. We have first to show that `crs (n+1)` is the result of merging $c_1 : c_2 : \cdots c_m : \perp$, where $m = p_n^2$ with the infinite list of multiples

$p_{n+1}p_{n+1}, p_{n+1}(p_{n+1}+1), \ldots$ of p_{n+1}. That gives the partial list of all composite numbers up to p_{n+1}^2. Finally, we need the result that $p_{n+2} < p_{n+1}^2$.

The partial list $\texttt{primes}_5$ produces all the primes smaller than $2209 = 47 \times 47$.

Answer to Exercise J

Because an odd integer is excluded from the final list if it takes the form $2n+1$ where n is of the form $i+j+2ij$. But

$$2(i+j+2ij)+1 = (2i+1)(2j+1).$$

Answer to Exercise K

No, f is not monotonic: $\bot \sqsubseteq 1$ but $f(\bot) \not\sqsubseteq f(1)$. For the second function (call it g) we have $\texttt{xs} \sqsubseteq \texttt{ys}$ implies $g(\texttt{xs}) \sqsubseteq g(\texttt{ys})$, so g is monotonic. But g is not continuous, so it's not computable.

Answer to Exercise L

The definition is

```
mkTorus ass = head (head xss)
 where xss = zipWith5 (zipWith5 Cell)
             ass (rotr xss) (rotl xss)
             (map rotr xss) (map rotl xss)
```

Whereas `rotr` and `rotl` rotate the rows of a matrix, `map rotr` and `map rotl` rotate the columns. The definition of `zipWith5` has to be made non-strict in its last four arguments.

9.9 Chapter notes

Melissa O'Neill has written a nice pearl on sieve methods for generating primes; see 'The genuine sieve of Eratosthenes', *Journal of Functional Programming* 19 (1), 95–106, 2009. Ben Sijtsma's thesis *Verification and derivation of infinite-list programs* (University of Groningen, the Netherlands, 1988) studies various aspects of infinite-list programs and gives a number of techniques for reasoning about them. One chapter is devoted to the proof of fairness in the paper–rock–scissors game.

My paper, 'On building cyclic and shared data structures in Haskell', *Formal Aspects of Computing* 24(4–6), 609–621, July 2012, contains more examples of the uses of infinite and cyclic lists. See also the article on 'Tying the knot' at

```
haskell.org/haskellwiki/Tying_the_Knot
```

Hamming's problem has been used as an illustration of cyclic programs since the early days of functional programming.