DRAFT: CHARITABLE THOUGHTS
(class notes)

Robin Cockett
Department of Computer Science
University of Calgary

September 23, 1996

Chapter 1

Preface

These notes are a preliminary collection describing charity from the point of view of a programmer.

Charity is a categorical programming language based on strong inductive and coinductive datatypes. It is a categorical typed polymorphic language in which it is (in a technical sense) not possible to write a non–terminating program. All programming is done using folding, unfolding, case matching, record formation, or mapping.

The language is does include higher order datatypes. However, in order to obtain higher order constructs it is necessary to declare them. Thus, the syntax does not support application as a built-in operation. This gives it a rather different flavour when compared to functional languages. Charity is directly comparable in expressive power to what David Turner has called “strong functional” languages. However, Charity uses directly the combinators provided by initial and final datatypes rather than providing a specification style front end, thus, the programming style is rather different.

The language was developed as a platform for verification. The semantics of the language is the initial category with all inductive and coinductive datatypes. This category has an enumerable axiomatization and thus a semi-decision procedure for equality. The proof system, which uses primitive induction and coinduction is illustrated.

Chapter 2

Getting Started

The purpose of this chapter is to get you started on the charity system. This you can best accomplish by running the charity system and playing! The charity system is an intepreter so that one can declare and define functions directly to the system. This makes it easy to run experimental programs.

The syntax of charity is simple and you will quickly pick it up. In fact it is likely that your most frequent error will not be syntax errors but rather typing errors. As has been mentioned charity is a strongly typed, polymorphic, language and its type checking may be viewed as a first level of verification that your program does what is expected of it.

2.1 Starting charity ...

The very first task you must accomplish is to check that you have access to **charity**. This you can find out by typing `charity`. The computer should respond:

```
[baggins] charity
The Charity Interpreter: Version X.X.

Charity>
```

You can get out by typing `ctl-d`.

In general, the way to use charity is to create a file containing your charity program and to read it in using:

```
Charity> rf "test.ch".
```

This causes the charity system to switch control to your file and to interpret the statements in that file. When the file ends, control reverts the terminal.

The next sections describe some elementary programs which you might try in order to get a feel for programming in charity .

2.1.1 The natural numbers

When you enter charity it has virtually no built-in computational ability. You introduce such ability by declaring datatypes. For example, you may want to program using the natural numbers $\mathcal{N} = \{0, 1, 2, ...\}$. You can introduce them into the system by the following declaration:

```
>data nat -> C = zero: 1 -> C
>              | succ: C -> C.
```

This says roughly:

> "Hey I want to introduce an object called nat. The maps (or functions) from nat to another object C are determined by choosing a point from C and giving an endomorphism (or self map) of C. Furthermore, nat itself has a chosen point which I will call zero and an endomorphism which I will call succ."

From a formal point of view this declaration states exactly what the natural numbers are!

The maps zero and succ are called the **constructors** of nat whose types are determined by substituting nat for C in the data definition:

```
zero: 1 -> nat
succ: nat -> nat
```

The elements of this natural number object are then:

$$\{zero, succ(zero), succ(succ(zero)), ...\}.$$

That is the natural numbers represented in a unary notation. These are now understood by charity and so can be entered as values:

```
>succ(zero).
succ(zero): nat
>
```

Now there may appear to be very little computational power behind this definition. However, in fact, we have just introduced sufficient power to write *any* primitive recursive function. So before proceeding let us explore some of this power!

2.1.2 Folds from the natural numbers

Probably the first thing we might want to know how to do is to add:

```
charity> def add: nat * nat -> nat
           = (x,y) => {| zero: () => y
                       | succ: n => succ n
                       |} x.
charity> add(succ(succ(succ(zero))), succ(succ(zero))).
succ(succ(succ(succ(succ(zero))))) : nat
```

Not blindingly fast but got the job done! Notice how we specify the type of the function introduced as nat * nat -> nat. It is surprisingly helpful to write down the types of functions you define: if the type is wrong the function cannot have been written correctly!

The brackets {| ... |} are called barbed wire. They contain the phrases of the **fold** operation. Each phrase is separated from the next by a | and is tagged (this is the significance of the colon) by a constructor name. A phrase is an abstract map: the zero phrase above, for example, consists of a **variable base** (in this case `()`) separated by a `=>` from a term (in this case `y`). This phrase indicates the how the folding should start. The variable base `()`, as it contains no variables, is called the **empty variable base**: it is the only variable base of type `1`. The `succ` phrase of the fold indicates that one should apply `succ` the number of times given by `x`. In this case the variable base is a single variable `n`, which is of type `nat`: this is a local variable with scope the remainder of the phrase. These two phrases, one quickly realizes, define addition.

Notice also that to apply a function to a single argument $f(x)$ we may omit the parentheses to give fx. In fact, given a series of applications $f(g(h(x)))$ we may avoid the parentheses by simply writing $fghx$ which has the same meaning. Please note: this is not the association used in functional languages where this would mean $((fg)h)x$.

Exactly how does the add function work? Below is a sample trace of add(succ(zero), succ(zero)):

```
add(succ succ zero, succ zero)

   => {| zero: () => succ zero
       | succ:  n => succ n
       |} succ succ zero
```

First the `x` and `y` are substituted with the actual arguments. The fold is now applied to `succ(succ(zero))`: the `succ` phrase is selected and its abstract map is applied to the fold which is applied to the number with one `succ` removed ...this is where recursion takes place:

```
   => { n => succ n } {| zero: () => succ zero
                       | succ:  n => succ n
                       |} succ zero

   => { n => succ n } { n => succ n } {| zero: () => succ zero
                                       | succ:  n => succ n
                                       |} zero
```

Finally, the zero constructor is encountered and the `zero` phase is selected and the variable `n` of the previous abstracted map can be replaced by a value:

```
   => { n => succ n } { n => succ n } succ zero


   => { n => succ(n) } succ succ zero

   => succ succ succ zero
```

Once you have inspected this trace you will realize that the fold over the natural numbers acts just like a “for loop” in an imperative language. It allows one to iterate a function a given number of times on some starting state.

In the above we have shown a “by value” evaluation of the term. In theory the order of evaluation in charity does not actually matter: all programs terminate (in a definable sense) under all evaluation strategies. However, in practice the order of evaluation can be quite important: it may significantly affect the execution speed of a program. To write efficient programs one must ultimately know how charity will do the evaluation. At this stage we shall not concern ourselves with these issues and take a näive approach to evaluation.

In general a fold operation is a sequence of phrases enclosed in barbed wire, labelled by the constructors of the datatype, and separated by vertical bars. The fold operation can be applied to any element of the datatype. Every (inductive) datatype has an associated fold operation: we shall meet more examples below.

Well with such success (i.e. defining add) under our belt we might consider multiplication and even exponentiation:

```
charity> def mult: nat * nat -> nat
             = (x,y) =>  {| zero: () => zero
                          | succ: n => add(x,n)
                          |} y.
charity> mult(succ succ succ zero, succ succ zero).
succ(succ(succ(succ(succ(succ(zero)))))) : nat
charity> def exp: nat * nat -> nat
             = (n,m) =>  {| zero: () => succ zero
                          | succ: x => mult(n,x)
                          |} m.
charity>exp(succ succ succ zero, succ succ zero).
succ(succ(succ(succ(succ(succ(succ(succ(succ(zero))))))))) : nat
charity> def expexp: nat * nat -> nat
             = (n,m) =>  {| zero: () => succ zero
                          | succ: x => exp(n,x)
                          |} m.
charity> expexp(succ succ succ zero, succ succ zero).
succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(
succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(succ(
succ(zero))))))))))))))))))))))))))) : nat
```

What is expexp??

2.1.3 Case statement

Sometimes one wants to do different things depending on whether the number you are holding is zero or succ of something (i.e. whether it is non-zero or not). The **case** operation allows you to do this. For example to program the predecessor function the case construct is rather useful:

```
charity> def pred: nat -> nat
             = zero() => zero()
             | succ m => m.
charity> pred succ succ zero.
succ(zero) : nat
charity> pred zero.
zero : nat
```

Notice the predecessor of `zero` is `zero`. Each phrases of a case operation starts with a pattern. Intuitively, the input data is matched sequentially down the sequence of patterns (separated by slashes) until a match is found. At the first match so found, the computation indicated by the term to the right of the arrow is initiated. In order for a case statement to be well-formed there must be patterns which match every possible value of the type of input data.

Once one has the predecessor function it is possible to program a truncated subtraction operation: this is often called the `monus` function:

```
charity> def monus: nat * nat -> nat
             = (n,m) =>   {| zero: () => n
                           | succ: x => pred x
                           |} m.
charity> monus(succ succ succ zero,succ succ zero).
succ(zero) : nat
```

The syntax of the case operation is only slightly different from that of the fold operation. A case statement matches the current value against the alternative permissible evaluated forms of an element of that datatype. Notice that barbed wire does not encloses the case statement: it is enclosed by curly brackets. Furthermore, there are no colons in the case statement. As mentioned before, however, there must be a pattern to cover each constructor of the datatype to be matched.

2.1.4 Boolean values

Two natural numbers are equal if and only if

$$monus(n, m) + monus(m, n) = 0.$$

Thus, we can determine the equality of numbers on the basis of their absolute difference. The only problem is that we should like the system to reply `true` or `false`. This means we need to introduce another important datatype:

```
charity> data bool -> C =  true: 1 -> C
                         | false: 1 -> C.
```

This may also be written more succinctly as:

```
charity> data bool -> C =  true|false: 1 -> C.
```

where the saving uses the fact that the constructors have the same type. The notation is particularly useful when one wishes to introduce datatypes to represent sets of atomic values.

This datatype consists of the two constant constructors `true` and `false`: maps from this datatype are determined by where these constants go.

A proposition is just a boolean valued function, that is a function with codomain the booleans. For example, we may determine whether a number is zero or not by using the case construct of the natural numbers as follows:

```
charity> def is_zero: nat -> bool
             = zero => true
             | succ _ => false.
```

We may test whether two numbers are equal as follows:

```
charity> def equal_nats: nat * nat -> bool
             = (n,m) => is_zero add(monus(n,m),monus(m,n)).
charity> equal_nats(succ succ zero,succ zero).
false : bool
charity> equal_nats(succ succ zero,succ succ zero).
true : bool
```

The case operation of this datatype is particularly important and can be used to implement an "if ... then ... else ..." or conditional operation:

```
             { true  => f
             | false => g
             } P
```

Here the function `f` is evaluated if the boolean function `P` evaluates to `true`. Otherwise, `P` *has to* evaluate to `false` and `g` is evaluated. As all well–typed charity programs terminate it is not possible (as in a more standard language) that `P` will simply not terminate: eventually `P` must return `true` or `false` .... of course, you may have to wait for an arbitrary long time for this event to actually happen (if you do not first run out of memory)!

There are a number of standard functions which are associated with Booleans. Here are four such functions developed using the case construct:

```
def and: bool * bool -> bool
    = (x,y) => { true => y
               | false => false
               } x.

def  or: bool * bool -> bool
    = (x,y) => { true => true
               | false => y
               } x.

def imply: bool * bool -> bool
    = (x,y) =>   { true => y
                 | false => true
                 } x.

def not: bool  -> bool
    = x => { true  => false
           | false => true
           } x.
```

The first three of these functions have the important property that their evaluation need not depend on their second argument (as it is only used in one of the case phrases). Notice that we could have written each of these functions so that their evaluation always depended on the evaluation of the second argument but not always on the first. We shall say that and, written as above, has its first argument **semi-essential**. If, no matter what values other arguments take and no matter how you write the code, one must always evaluate a given argument then we shall say that it is an **essential** argument. In this case, as we could have written the function in such a manner to make the evaluation of the first argument unnecessary in some circumstances we must conclude it is only semi-essential. These informally introduced dependencies shall become particularly important when we consider lazy evaluation and the efficiency of programs.

There is another more natural way to write these functions which use pattern matching at the definition rather than in the body. Let us consider the definition of and to illustrate how we can write things in different ways using the pattern matching. Our first attempt involves giving the complete operation table for and:

```
def and: bool * bool -> bool
    = (false,false) => false
    | (true, false) => false
    | (false, true) => false
    | (true,  true) => true.
```

This certainly leaves us in no doubt as to the function implemented! However, it is somewhat verbose. We may note that and is only true if both its arguments are true. This gives:

```
def and: bool * bool -> bool
    = (true,true) => true
    |     _       => false.
```

Where the underscore is an "anonymous" variable which we cannot use in the body of a function. It is used to give an "otherwise" option in the patterns. Notice that in these two definitions the order of evaluation of the arguments will be determined by the compiler ...

2.1.5 Pairs and abstract maps

Consider how one might write the factorial function. At each iteration one wants not only the factorial of that number but also the number of the iteration so that one can calculate the next factorial. This turns into the following charity fold:

```
def factorial:  nat -> nat
    = n => p1 {| zero:    () => (zero,succ zero)
               | succ: (x,y) => (succ x,mult(x,y))
               |} n.
```

Notice how in this fold the "state" is a pair of numbers. The first number simply counts the iterations while the second accumulates the factorial. To obtain the factorial it is necessary to project out the second coordinate of the result of the fold operation this is the meaning of the p1 applied to the fold.

Notice also the form of the succ phrase of the fold. The variable base is a pair of variables, (x,y), each of which has scope only over that phrase. In general, a variable base can be formed by pairing together any number of distinct variables.

There are two basic combinators that extract, respectively, the first and second coordinate of a pair:

$$p_0 : A \times B \to A; (x, y) \mapsto x$$

$$p_1 : A \times B \to B; (x, y) \mapsto y.$$

Notice that the mathematical statement of what p_1 does, namely $(x, y) \mapsto y$, looks remarkably like an abstact map and indeed it is. Thus, we can replace `p1` in the above program by the abstract map `{ (x,y) => y }`:

```
def factorial:  nat -> nat
    = n => {(x,y) => y} {| zero:    () => (zero,succ zero)
                         | succ: (x,y) => (succ x,mult(x,y))
                         |} n.
```

Sometimes a fold operation will produce a pair both of whose components will subsequently be used. In this case projecting will not suffice. Instead, we must use the technique of matching the result against a variable base (or patterns) and applying a function to the liberated components. For example an alternative way of determining whether two natural numbers are equal is to try to decrement the one number the other number of times. If the decrementing "underflows" we raise a flag otherwise we will examine what remains of the first number: only when this remainder is zero and no underflow has occurred are the numbers equal.

```
def equal_nats'(n,m) =  {(flag, zero)   => flag
                        |    _          => false
                        } {| zero: ()          => (true,m)
                           | succ: (_,zero)    => (false,zero)
                                 | (_,succ x)  => (true,x)
                            |} n.
```

Note that this is apparently a more complicated way of determining the equality of two natural numbers than the previously described technique. However, as an iteration over only one of the numbers is required, it is much more efficient!

Notice how the result of the fold state is a product of types `bool * nat` whose components are both used to determine whether equality has occurred. The components are accessed by matching the result against the variable base `(flag,num)` in the abstract map and then using these liberated components to determine the resulting truth-value.

2.1.6 Success or fail?

It is tempting to think that the use of flags is fundamental. However, there is another technique which is to tag data with a constructor. In this case we need only keep track of the data while the flag is "true" so we may choose a rather special (but ubiquitous) tagging mechanism: the "success or fail" datatype:

```
data SF(A) -> C = ss: A -> C
                | ff: 1 -> C.
```

Notice that this datatype depends on the type variable `A`: we say it is **parametric** in `A`. This means that we may fill in this type with any type to obtain $SF(nat)$ or $SF(bool)$, etc.

Here is another version of the equality test for natural numbers:

```
def equal_nats'': nat *  nat -> bool
    = (n,m) => { ss zero => true
               | _       => false
               } {| zero: () => ss m
                  | succ: ss succ x => ss x
                        | _         => ff
                  |} n.
```

While there is little difference between the two versions the second is probably marginally better as the structures involved are simpler.

2.1.7 Exercises

1. Write functions $min, max : nat * nat \longrightarrow nat$ for obtaining, respectively, the minimum and the maximum of two numbers.

2. Write functions $ttgt, lt, geq, leq : nat * nat \longrightarrow bool$ for testing, respectively, whether the first number is greater than, less than, greater than or equal, less than or equal to the the other.

3. Write a function to test whether a natural number is even $even : nat \longrightarrow bool$.

4. Write a function which returns the sum of all numbers less than or equal to the given number.

5. Write the "exclusive or" operation on booleans $bool * bool \longrightarrow bool$.

6. Write a test to determine whether one natural number divides another $divides : nat * nat \longrightarrow bool$.

7. Write a function to divide one natural number by another which returns the truncated division and the remainder $div : nat * nat \longrightarrow nat * nat$.

8. Write a function to determine whether a natural number is prime.

9. Write a function to determine the highest common factor of two natural numbers.

10. Write a function to determine the least common multiple of two numbers.

2.2 Lists and other inductive datatypes

Booleans and natural numbers are, in the scheme of things, rather simple datatypes. In this section we discuss further inductive datatypes such as lists, trees, and bushes. Perhaps the most important datatype to computer science is the list. We shall start our discussion with this datatype.

2.2.1 Introducing lists

This is how the list datatype is declared in charity :

```
data list(A) -> C = nil: 1 -> C
                  | cons: A * C -> C.
```

Notice that we have declared a **parametric** datatype of lists, that is the type variable `A` is left to be "filled in" later depending on how we use it. For example we could consider lists of booleans, `list(bool)`. An element of a list of booleans can take the following form:

$$cons(true, cons(false, cons(true, nil))).$$

Equally, we could have considered a list of natural numbers `list(nat)`. An element of this type has the following form:

$$cons(zero, cons(succsucczero, cons(succzero, nil))).$$

Charity recognizes the constructors of this datatype (i.e. the `nil` and the `cons`) and provides a more convenient syntax: the above lists can therefore be written more succinctly as:

$$[true, false, true] \;\; \text{and}[zero, succsucczero, succzero]$$

This is mighty convenient! However, you should realize that it is no more than a convenience: it does not reflect any underlying difference in the way lists are treated when compared to other datatypes.

Perhaps the first thing one wants to do with a lists is to append two lists. This is done using the fold construct:

```
charity> def append: list(A) * list(A)
             = (x,y) => {| nil:  ()    => y
                         | cons: (a,l) => cons(a,l)
                         |} x.
charity> append([true,false],[true,true]).
[true,false,true,true] : list(bool)
charity> append([succ(zero), zero], [succ(succ(zero)), zero, zero]).
[succ(zero), zero, succ(succ(zero)), zero, zero] : list(nat)
```

It is important to note that the way we have typed the function append implies that it should be applicable to a pair of lists of any type. The two tests illustrate the point. The ability to write **polymorphic** functions in this fashion is a powerful technique which significantly aids the clarity and modularity of programs.

You may note also that this is strikingly similar to the addition of natural numbers. Indeed, an alternative representation of the natural numbers is as `list(1)`, that is lists of the one element set, `1` (whose only element is `()`). Under this representation addition is append! This is another example of the power of polymorphism.

```
charity> append([(),(),()],[(),(),(),()]).
[(),(),(),(),(),(),()] : list(1)
```

Given a list of lists, a common operation one may wish to perform is to flatten it to a (single) level list. This operation can be written using a fold operation and the append function:

```
charity> def flatten: list(list(A)) -> list(A)
             = LL => {| nil: () => nil
                      | cons: (l,t) => append(l,t)
                      |} LL.
charity> flatten([[true,false],[],[true],[false,false,true]]).
[true,false,true,false,false,true] : list(bool)
```

It is easy to forget that once one has the datatype lists one has also list of lists, lists of lists of lists, etc. Charity allows one to compose datatypes arbitrarily to produce complex composite types.

2.2.2 Mapping over lists

Often one wishes to do some operation uniformly to each element of a list. For example, one may want to negate all the enties of a boolean list. One can do this using the fold operation as follows

```
charity> def NOT: list(bool) -> list(bool)
             = BL => {| nil: () => nil
                      | cons: (v,l) => cons(not v,l)
                      |} BL.
charity> NOT [true,false,true].
[false,true,false] : list(bool)
```

However, charity provides a more direct way of accomplishing this using the **map** operation. This allows one to write this negatation more succinctly:

```
charity> def NOT: list(bool) -> list(bool)
             = BL => list{not} BL.
charity> NOT [true,false,true].
[false,true,false] : list(bool)
```

It does not make much odds in terms of execution speed which way one does this. However, the map operation is more succinct and it is possibly easier for someone reading the program to understand. The map operation uses the name of the datatype (`list` in this case) followed by a sequence of abstract maps, separated by commas and enclosed in curly brackets (in this case only one abstract map is required, namely `x => not(x)` or more simply `not`). The number of abstract maps must correspond to the number of type parameters of the list datatype. If there had been more than one type parameter we would have had to have specified more abstract maps.

In order to increment all the elements of a list of numbers by some fixed number`y`, we can use the map contruct:

```
charity> def increment_list: list(nat) * nat -> list(nat)
             = (L,y) => list{x => add(x,y)} L.
charity> increment_list([succ zero, succ succ zero, succ succ zero]
                       ,succ zero).
[succ(succ(zero)), succ(succ(succ(zero))), succ(succ(succ(zero)))] : list(nat)
```

There are also some other more surprising things that one can accomplish using the mapping construct. Consider the following (actually not so wacky) definition of multiplication which uses the `flattening` function associated with lists:

```
charity> def wacky_mult: list(1) * list(1) -> list(1)
             = (x,y) => flatten list{() => y} x.
charity> wacky_mult([(),(),()],[(),(),(),(),()]).
[(),(),(),(),(),(),(),(),(),(),(),(),(),(),()] : list(1)
```

Can you work out why this works?

We have seen that the mapping function also allows one to do things like add some number to all the elements of a list - and just to confuse things we will now use the list representation of the natural numbers to accomplish the same thing:

```
charity> list{x => append([(),(),()],x)} [[()],[(),()],[(),(),()]].
[[(),(),(),()],[(),(),(),(),()],[(),(),(),(),(),()]] : list(list(1))
```

Notice that here we have directly evaluated an expression on its arguments without having first written a definition of a function. This is perfectly legal in charity .

Given two lists of items one may wish to form the list of pairs of items, for which the first component comes from the first list and the second component form the second list. This can be accomplished rather slickly with the map operation combined with flattening:

```
charity> def PAIRS: list(A) * list(A) -> list(A * A)
             = (Lx,Ly) => flatten list{ x => list{ y => (x,y) } Ly } Lx.
charity> PAIRS([true,false],[true,false]).
[(true,true),(true,false),(false,true),(false,true)]:list(bool)
```

Notice that the enumeration cycles through the second coordinate fastest.

2.2.3 More foldings of lists

If we have a list of booleans it is often useful to be able to form their disjunction. In charity this can be achieved by a fold operation:

```
charity> def OR: list(bool) -> bool
             = L => {|  nil: () => false
                     | cons: (b,v) => or (b,v)
                     |} L.
```

Notice that the disjunction of the empty list is `false` and the disjunction is formed by disjoining the elements of the list in sequence: starting at the last element (this is why this fold is often called a "fold right"). Notice also that we can write the `cons` phrase more succinctly as `z => or z` as there is no need to break up the variable base.

Similarly the conjunction of a list of booleans is given by:

```
charity> def AND: list(bool) -> bool
             = L => {|  nil: () => true
                     | cons: z => and z
                     |} L.
charity>
```

This time the conjunction is folded over the list starting at `true`.

If we wish to sum a list of natural numbers we can do this in an analogous manner by folding addition over the list starting at `zero`:

```
charity> def ADD: list(nat) -> nat
             = L => {| nil: () => zero
                     | cons: z => add z
                     |} L.
charity>
```

To get the product of the list we have:

```
charity> def MULT: list(nat) -> nat
             = L => {| nil: () => succ zero
                     | cons: z => mult z
                     |} L.
```

We have already seen how to flatten a list using a fold. We can also reverse a list using a fold:

```
charity> def reverse: list(A) -> list(A)
             = L => {| nil: () => []
                     | cons: (v,R) => append(R,[v])
                     |} L.
```

This is a näive reverse, as its complexity is of order the square of the length of the list. A reverse which is has complexity of order the length of the list can be written by constructing a new list from the old one by repeatedly removing the first element and push it onto the front of the new list:

```
charity> def rev: list(A) -> list(A)
             = L => p0 {| nil: () => ([],L)
                        | cons: (_,(R,[]))         => (R,[])
                              | (_,(R,cons(A,L))) => (cons(A,R),L)
                        |} L.
```

Notice how this fold uses a composite type and how it is necessary to project in order to select the first component of the state. Also notice how we have used pattern matching in the second phrase of the fold.

2.2.4 Introducing trees

The binary tree datatype, with a typical tree element, is defined as follows:

```
charity> data tree(A) -> S = leaf: A -> S
                           | branch: S * S -> S.
charity> branch(leaf true
               ,branch(branch(leaf false
                             ,leaf true
                             )
                      ,leaf true
                      )
               ).
branch(leaf(true)
      ,branch(branch(leaf(false),leaf(true)),leaf(true))) : tree(bool)
```

A common operation with trees is to traverse them and to collect the values at the leaves into a list. We may do this rather simply using the fold operation as follows:

```
charity> def traverse: tree(A) -> list(A)
             = T => {| leaf: a => [a]
                     | branch: z => append z
                     |} T.
charity>
traverse(branch(leaf true,branch(branch(leaf false,leaf true),leaf true))).
[true,false,true,true] : list(bool)
```

In addition using the fold over trees one can calculate the height, the size and many other common functions associated with trees. For example the size of a tree is the number of nodes (`branch`s) it possesses. This may be calculated by:

```
charity> def tree_size: tree(A) -> nat
             = T => {|   leaf: _ => zero
                     | branch: z => succ add z
                     |} T.
charity>
tree_size(branch(leaf true,branch(branch(leaf false,leaf true),leaf true))).
succ(succ(succ(zero))) : nat
```

To calculate the heights (minimum and maximum leaf height) you need the `max` and `min` functions for natural numbers. Can you work out how to do this?

In addition, other tree structures can be defined. A tree with more than two branches can be defined as follows:

```
data rose(A) -> C = rleaf: A -> C
                  | rnode: list(C) -> C.
```

These are called "Rose trees" after the man who first investigated them.

There are also trees that retain values at both the leaves and the nodes:

```
data rose'(A,B) -> C = rleaf': A -> C
                     | rnode': B * list(C) -> C.
```

These are often used in database applications.

How would you compute the size of a `rose'` tree and what would the traversal routines look like?

2.2.5 Definitions with function arguments

Often it is useful to have functions as arguments of definitions. This allows one to define a function dependent on filling in the values of certain other functions. While **charity** distinguishes sharply between "values" (those things which can be inputs of functions) and functions (essentially bits of code), there is a mechanism for passing functions as arguments of defined functions. This mechanism allows one can write very general and useful definitions and which can further enhance modularity.

For example suppose you wish to determine whether something in a list satisfies a given property P. As usual we regard a property to be a function from a type to the boolean datatype. One way to accomplish this is to map the property over the list and then form the disjunction of the result:

$$list(A) \longrightarrow list(bool) \longrightarrow bool.$$

This can be written in **charity** as:

```
def in_list{P: A -> bool}: list(A) -> bool
    = L => OR list{P} L.
```

Notice that the definition of `in_list` is dependent on a function argument `P`. Function arguments must be placed between curly brackets separated by commas *before* the value arguments.

We can now use this definition by providing a predicate:

```
charity> in_list{x => equal_nats(succ succ zero,x)
                }([succ zero,zero,succ succ zero,succ zero]).
true : bool
```

This method of passing functions as arguments of definitions provides a very powerful way of building general purpose and modular programs. For example `in_list` is playing the role of a membership predicate. We could, however, equally have used the definition to obtain a function to determine whether a number greater than 2 is in the list by passing the appropriate predicate as an argument of the definition.

```
charity> in_list{x => gt(succ succ zero,x)
                }([succ zero,zero,succ succ zero,succ zero]).
false : bool
```

Of course, we would have to have defined `gt`!

One common operation on the list datatype involves extracting those items for which a predicate is true from a list. This operation is often called *filtering*. For example, if we have a list of numbers

$$[1, 2, 3, 4, 5]$$

and want to keep all numbers less than 4 we could use the function:

```
def lt4: list(integer) -> list(integer)
    = L => {| nil: () => []
            | cons: (a, L') => { true () => cons(a, L')
                               | false() => L'
                               } lt(a, 4)
              |} L.
```

(where we assume we have the appropriate definitions).

This definition works, but is not very general. It is desirable to have a function which works for any predicate and list. This function is commonly called `filter` and is defined as:

```
def filter {P: A -> bool}: list(A) -> list(A)
    =  L => {|  nil: ()       => []
             | cons: (a, L') => { true  => cons(a, L')
                                | false => L'
                                } P a
             |} L.
```

Notice that the predicate `pred` has been passed as a function parameter to the `filter` and is used in the function body to determine whether an element should be included in the output.

This allows us to write the function `lt4` in a more general fashion:

```
def lt4 : list(integer) -> list(integer)
    = L => filter{n => lt(n, 4)} L.
```

and indeed may allow us to omit the explicit definition altogether as the required function can be obtained so directly from the more general filter function.

2.2.6 Exercises

1. Write a function to determine the length of a list,

$$len : list(A) \longrightarrow nat.$$

2. Write a function $upto : nat \longrightarrow list(nat)$ which returns the list of numbers less than or equal to the given number in increasing order.

   Now write a function which given a number returns all pairs of numbers less than or equal to the given number.

3. Write a function which given any list will return its list of "tails", that is the list of lists the element of which are the tails (last n elements for each $n = 0, 1, 2, ...$) of the given list.

4. Write a function which determines whether there is a number in a list which is greater than a given number,

$$exceed : nat * list(nat) \longrightarrow bool.$$

5. Write a function which given a list of ordered items returns the least item greater than a given item. (Hint: what happens if there is no such item? You need a datatype which handles this: the "success or fail" datatype will be useful.)

6. Write a function `sublist(L)` which given a list returns the list of all sublists of the list.

7. Write a function which returns the height of the highest true leaf in a tree of booleans,

$$true\_height : tree(bool) \longrightarrow nat.$$

8. Define the datatype of B-trees as follows:

```
data Btree(A,B) -> C = bleaf: A -> C
                     | bnode: C * (B * C) -> C.
```

   There are three possible ways to collect the values of a B-tree into a list: by a prefix traversal, an infix traversal, and a postfix traversal. Write functions corresponding to each of these traversals with type

$$Btree(A, A) -> list(A).$$

9. Write a function to traverse a Rose tree.

10. Write functions to determine the hieghts of trees and Rose trees.

11. Given two lists of natural numbers write a function which returns the pairs of numbers (first component in the first list and second component in the second list) which are ordered. Now write a function which returns those pairs in which the first component divides the second.

12. Define the datatype `bit` by:

```
data bit -> C = B0|B1: 1 -> C.
```

and define a function which takes in two lists of bits and returns a list of pairs of bits as if the shorter list had been padded on the front by `B0`s to make the lengths equal.

$$bit\_pair : list(bit) * list(bit) \longrightarrow list(bit * bit)$$

$$([B1, B0, B1], [B1, B1]) \mapsto [(B1, B0), (B0, B1), (B1, B1)].$$

(Hint you may have to reverse one of the lists!)

13. Write a full adder. This takes in three bits (the carry bit and one bit from each number) and returns the new carry and the addition,

$$full\_add : bit * (bit * bit) \longrightarrow bit * bit.$$

14. Now write a binary adder for arbitrary lists of bits,

$$bin\_add : list(bit) * list(bit) \longrightarrow list(bit).$$

15. Write a function `csublist(L)` which given a list returns the list of all contiguous sublists of the list. Now write a function which returns all the contiguous sublists of a given length `lencsublist(l,N)`.

    (Hint: if you know what the sublists of `La` are what are the contiguous sublists of `a::La`?)

16. Let `pred(x,y)` be a predicate (map to `bool`) on pairs of items. Write a `group{ pred } (L)` function which "groups" the elements of a list into a list of lists all of whose adjacent elements are related by `pred`. For example if `pred` is a test for equality then

$$group\{pred\}([1, 2, 2, 4, 5, 5, 6])$$

    will produce

$$[[1], [2, 2], [4], [5, 5], [6]].$$

    By convention, we will assume `group{ pred } ([]) = []`.

    Given that one can `sort` a list, write a program to determine how many occurrences of each item is in a list. For example

```
["john","mary","susan","john"]
                  --> [("john",2),("mary",1),("susan",1)]
```

2.3 The coinductive and the infinite

The purpose of this subsection is to introduce you to the coinductive datatypes of charity . These datatypes add a further dimension to the language and allow a rather different way of looking at many algorithms. Coinductive datatypes are potentially infinite structures: charity only computes the portion of the structure that is needed for the calculation at hand. However, this allows one to *think* in terms of having the whole structure available and to using it, for example, as a look-up table.

2.3.1 Introduction to coinductive datatypes

Coinductive datatypes come equipped with *destructors* rather than constructors. A destructor indicates how one can break down a datatype into "smaller" components. The simplest coinductive datatype is the product (which is built-in to charity): its destructors are the two projections (`p0` and `p1`). However, the product is not infinite: to obtain a (potentially) infinite datatype one must destruct to a type which is dependent on itself. Examples of potentially infinite datatypes are streams (or infinite lists), colists (or channels), and cotrees.

In charity a coinductive datatype is declared in a similar manner to an inductive datatype. For example, to declare a stream (an infinite list) parameterized by type A we would write:

```
data C -> inflist(A) = head : C -> A
                     | tail : C -> C.
```

This declaration says "maps from a type C to streams on A are determined by providing a map from that type to A and an endomorphism (self-map) on the type C." As for inductive types, this declaration also delivers two operations, in this case `head` and `tail`, which are called **destructors**. The types of these destructors are revealed by substituting the type variable C by the datatype itself.

Notice that in a coinductive declaration the introduced datatype is on the right of the arrow: if it had been an inductive definition we would have expected it to be on the left. Furthermore, the composite types in the phrases of a coinductive declaration are also on the right of the arrows rather than, as for inductive declarations, on the left. For this reason sometimes the coinductive datatypes are refered to as "right datatypes" (and the inductive datatypes as "left datatypes").

It is instructive to regard the type variable C in a coinductive declaration as a set of states. To obtain an infinite list, for example, one must provide an "output" map $\beta : C \longrightarrow A$, corresponding to `head`, and a change of state map $\alpha : C \longrightarrow C$, corresponding to `tail`. Once one has these one can obtain an infinite sequence of outputs from a starting state s_0:

$$(\beta(s_0), \beta(\alpha(s_0)), \beta(\alpha(\alpha(s_0))), ..., \beta(\alpha^n(s_0)), ...$$

this infinite sequence is an infinite list.

2.3.2 Unfolding

To actually obtain this infinite sequence in charity the **unfold** operation is used. Thus, if β had been the identity function and α the successor function on the natural numbers, we would obtain the sequence of natural numbers. This can be defined in charity by:

```
def nats: 1 -> inflist(nat)
    = () => (| x => head : x
            |      tail : succ x
            |) zero.
```

The funny braces of the unfold operation `(| ... |)` are called "lenses." Inside the lense one stipulates a variable base for the state which can be used by all the **threads** labelled by destructors. The threads are separated by slashes: in order to convey the sense of the unfold these slashes should be alighned vertically with the slash of the opening and closing lenses. Outside one puts the desired starting point for the state of the unfolding operation.

We can also form streams of streams. These may be thought of as infinite 2–dimensional arrays. Thus, the infinite array of numbers which increases diagonally across such an array:

```
0 1 2 3 ...
1 2 3 4
2 3 4 5
3 4 5 6 ...
.     .
:     :
```

may be defined in charity by:

```
def natsnats: 1 -> inflist(inflist(nat))
    = () =>  (| x =>  head :(| y => head : y
                             |      tail : succ y
                             |) x
             |       tail : succ x
             |) zero.
```

From a stream (such as `nats`) we can extract the first element by writing `head(nats)`, the second element by writing `head(tail(nats))`, the third element by writing `head(tail(tail(nats)))`, etc.

More generally it is useful to have a function which extracts the n^{th} element of a stream. This can be written by iterating over the `tail` map n–times and then taking the `head`:

```
def get:  inflist(A) * nat -> A
    = (IL,n) => head {| zero: () => IL
                      | succ: IL' => tail IL'
                      |} n.
```

It is much more convenient to write the decimal number rather than their unary representation. Thus, we may use the coercion from decimal numbers to unary to obtain a more useable function:

```
def GET:  int * inflist(A) -> A
    = (IL,n) => get(IL,int_2_nat n).
```

This allows us to obtain also the $(n, m)^{th}$ element of a 2–dimensional infinite array. Notice that the (n, m)–entry of `natsnats` is just $n + m$: so that this is just the infinite lookup table for addition! We may therefore define:

```
def add_lookup: nat * nat -> nat
    = (n,m) => get(n,get(m,natsnats)).
```

Can you write a function for multiplication which used a lookup table?

2.3.3 Records

Coinductive datatypes in addition to the unfold operation have a **record** and a **map** operation. The record allows one to create maps to the datatype which are defined on the components. Thus, if one wants to add a "one" on the front of the infinite list `nats` one would write

```
(* the above infinite list with a leading 1
   ie. (1,0,1,2,3,...)
*)
def onenats: 1 -> inflist(nat)
    = () => (head: succ zero, tail: nats).
```

The record uses round brackets with entries separated by commas and tagged by the destructors. The type of the term associated with a destructor in a record must match the output type of that destructor.

The record syntax is also the syntax used by charity to print coinductive datatypes. While this will sometimes make the output appear rather wordy, it does provide a uniform syntax for all coinductive datatypes. Charity elvaluates coinductive datatypes lazily: this means that it only displays (or develops) that part of the structure which has been demanded. There is a mechanism for forcing further evaluation of a displayed coinductive type called **poking**: it requires the user to interact with the system to simulate demand. Thus, the user must press `<return>` if he desires to see more components of an output with a coinductive component. With a persistent finger one can develop an infinite structure to any depth!

2.3.4 Mapping

The map operation on an infinite list allows one to change the values out of which the infinite list is made by a given operation. Thus, if one wanted to double every element of the stream of natural numbers to obtain the stream of even numbers one could write:

```
(* an infinite list of even numbers
   ie. (0,2,4,...)
*)
def evens = inflist{x => add(x,x)} nats.
```

Notice the syntax for this operation is (deliberately) exactly the same as for the map operation of inductive datatypes.

It is easy to underestimate the power of a mapping operation. Consider the following: you want to form a 2–dimensional infinite array whose (n, m)–entry is the pair consisting of the n–entry of one stream and the m–entry of another. The following produces the desired array of pairs:

```
def make_array: inflist(A) * inflist(B) -> inflist(A * B)
    = (sx,sy) => inflist{x => inflist{y =>(x,y)} sy} sx.
```

To obtain a three dimensional array from `sx`, `sy`, and `sz` one could then use:

$$make\_array(sx, make\_array(sy, sz)).$$

2.3.5 Simple sequences

The first and most obvious use of infinite lists is to generate sequences of numbers. Given a particular way of generating a sequence we may often use it directly as a **charity** program to generate an infinite list.

Factorials

An alternative way of obtaining the factorial of a number is to create a stream of factorials in which one can look-up the value sought. Recall that the definition of factorial is as follows:

$$\begin{array}{rcl} fact(0) & = & 1 \\ fact(n+1) & = & (n+1) \cdot fact(n). \end{array}$$

This gives the following coinductive definition:

```
def fact: 1 -> inflist(int)
    = () => (| (x,y) => head: x
             |          tail: (MULT(x,y),SUCC(y))
             |) (1,1).
```

Notice how all the computation is concentrated in the calculation of the tail map.

Fibonacci numbers

Another well-known sequence is the Fibonacci sequence. It is recursively defined as:

$$\begin{array}{rcl} fib(0) & = & 0 \\ fib(1) & = & 1 \\ fib(n+2) & = & fib(n) + fib(n+1). \end{array}$$

In **charity** this gives the following definition of Fibonacci's sequence:

```
def fib: 1 -> inflist(int)
    = () => (| (x,y) => head: x
             |          tail: (y,ADD(x,y))
             |) (0,1).
```

The definition gives a relatively efficient way of computing the sequence as it works from a current state (containing the last two numbers) rather than by the unwinding of a recursive specification which leads to an expenential explosion of repeated computations of previously encountered values.

Pascal's triangle

By using a stream of lists of numbers we can generate Pascal's triangle. Recall that to obtain the next row of Pascal's triangle one sums the two numbers immediately above (filling out with zeros). The only complication is producing the function to produce the next row from the last. The **charity** definition is:

```
def pascal: 1 -> inflist(list(int))
    = () =>
    (| x => head : x
     |      tail : cons {| nil : () => (0,[])
                        | cons: (n,(m,L)) => (n,(cons(ADD(n,m),L)))
                        |} x
     |)([1]).
```

To obtain the a binomial coefficient we need to extract the appropriate row from the triangle then the appropriate element from that list. We may do this using the `get` function followed by the `get_list` function. this latter uses the success or fail datatype:

```
def tail: list(A) -> list(A)
    = nil => nil
    | cons(_,L) => L.

def head: list(A) -> SF(A)
    = nil => ff
    | cons(a,_) => ss a.

def GET_list: list(A) * int -> SF(A)
    = (L,n) => head_list {| zero: () => L
                          | succ: L' => tail_list L'
                          |} int_2_nat n.

def binomial: int * int -> SF(int)
    = (n,m) = GET_list(m,GET(n,pascal)).
```

The Ackermann's function

Using an infinite list of infinite lists to build an infinite table, we can define Ackermann's function. This is of some theoretical interest as the inductive datatypes by themselves only provide the primitive recursive functions; however, with the introduction of coinductive datatypes we can define functions which are not primitive recursive.

The defining equations for the Ackermann's function are:

$$\begin{aligned}
\mathrm{ack}(0,n) &= s(n)\\
\mathrm{ack}(m+1,0) &= \mathrm{ack}(m,1)\\
\mathrm{ack}(m+1,n+1) &= \mathrm{ack}(m,\mathrm{ack}(m+1,n))
\end{aligned}$$

The first equation tells us that column 0 is an infinite list whose i^{th} value is $i+1$:

	0	$\cdots$
0	1	$\cdots$
1	2	$\cdots$
2	3	$\cdots$
$\vdots$	$\vdots$	$\ddots$

That is column 0 is the infinite list of numbers starting from 1. The second equation tells us that the value in row 0 of a column is the same as the value in row 1 of the previous column. While, the third equation tells us that the $n+1^{th}$ value in the $m+1^{th}$ column is the value in the previous column indexed by n^{th} value of the $m+1^{th}$ column.

	0	1	2	3	$\cdots$
0	1	2	3	5	$\cdots$
1	2	3	5	13	$\cdots$
2	3	4	7	29	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Since, a value in the $m+1^{th}$ column is determined from the last computed value in that column and a lookup in the previous column we can write the code for generating the next column as follows:

```
def col: inflist(int) -> inflist(int)
    = C => (| m => head: m
            |      tail: GET(C,m)
            |) GET(C,1).
```

This no allows us to generate the table for Ackermann's function:

```
def ack: 1 -> inflist(inflist(int))
    = () => (| C => head: C
             |      tail: col C
             |) tail nats.
```

2.3.6 Towers of Hanoi

It is interesting to discover that coinductive types can often be used to solve problems which traditionally have been regarded as prime examples of why general recursion is an indispensible programming tool. This is not to claim that the solution presented below has the same elegance as the general recursive solution (nor should we necessarily expect this), however, it does have an elegance and interest of its own.

A classic example of a problem whose solution can be very succinctly expressed by a recursive program is the towers of Hanoi. It is tempting to think, having seen the recursive solution, that the problem will be difficult – if not impossible – to solve in a language which lacks general recursion (such as charity). That it is neither should be a source of courage.

Certainly the approach to the problem must be rethought. The basic idea behind the solution to this problem we now present is that one uses solutions as the state of an infinite list. For example, if one knows how to move n disks from a source tower to a destination tower then to solve the problem for $n+1$ disks one first uses the previous solution to move all but the bottom disk onto the auxillary tower. This involves substituting the old destination tower by the new auxillary tower and the old auxillary by new destination tower. Next one moves the bottom disc to the destination tower. Finally one re-uses the solution to the n-disk problem to move the remainder from the auxillary tower to the destination tower. This is accomplished as before by an appropriate substitution of the old solution.

The idea of the charity program below is to produce an infinite list of solutions (for the case of one disk, two disks, ...). Each solution consists of a series of instructions of the form move a ring from this tower to that tower by giving the pair of towers.

To develop the program in charity we must start by introducing some preliminary datatypes. In particular, it is useful to have a special datatype for the towers, for the substitutions one wishes to preform on the toweres, and for moves (e.g. move a disk from this tower to that):

```
data towers -> C = source|dest|aux: 1 -> C

data C -> tower_sub(A) = SRC|DST|AUX: C -> A.

data C -> Move(A) = from|to : C -> A.
```

Clearly a crucial operation we will need is that of substitution. We shall write it as a function which takes in a tower substitution list and acts on a list of moves:

```
def subst: tower_sub(towers) * list(Move(towers)) -> list(Move(towers))
    = (sub,LM) => list{Move{source => SRC sub
                           |dest   => DST sub
                           |aux    => AUX sub
                           }
                      } LM.
```

What remains to be done is to form the infinite list of solutions:

```
def hanoi: 1 -> inflist(list(Move(towers)))
    = (| x => head: x
       |      tail: append(subst((SRC:source,DST:aux,AUX:dest),x)
                          ,cons((fst:source, snd:dest)
                               ,subst((SRC:aux,DST:dest,AUX:source),x)
                               )
                          )
       |) [(from:source,to:dest)].
```

The output of hanoi is a stream of solutions from which we can obtain the solution to a particular problem by using the get function.

2.3.7 Exercises

1. Give the definition of a stream of booleans which alternates `true` then `false` starting at `true`.

2. Given an infinite two dimensional array one can transpose it so that the (n, m)–entry is now the (m, n)–entry. Write the function `transpose`.

3. Give a definition of the infinite look–up table for multiplication and define a function `mult_lookup`.

4. Define a function `diagonal` which given a 2–dimensional infinite array returns the diagonal as a stream.

5. Given an increasing stream of natural numbers (such as the stream of squares $[0, 1, 4, 9, 25, ..]$), one may regarded it as a function on natural numbers. Write the definition of the stream which gives the truncated inverse (such as the truncated square root: $[0, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, ..]$). (Hint: the next number in the output stream is the successor only if the next number (in *nat*) exceeds or is equal to the current value of the function, otherwise it remains the same.)

6. Given two streams write the function which interleaves them to give a stream which takes values alternately from each of the original streams.

7. Given two streams write the function which produces a stream with the values of the original stream output in parallel as a pair.

8. Given a stream `s` define a function `accumulate` which has as n–entry the list of all the entries of `s` up to and including the n^{th}.

9. Given a stream of natural numbers write a function which returns the acculumated stream, in the sense of bing the sum of the previous values.

10. Given two streams of non-decreasing values write the function which produces a combined stream of non-decreasing values. Use this function to produce a stream of non-decreasing numbers which are either powers of two or powers of three.

11. Given a non-empty list of numbers (`nat *list(nat)`) write a function which returns the infinite list of numbers which can be expressed as a sum of elements from that list.

    For example, if the non-empty list was $(2, [])$ the colist would be the stream of even numbers. If the list where $(5, [7])$ then the colist would be:

    $$(0, 5, 7, 10, 12, 14, 15, 17, 19, 20, 21, ...)$$

    For any pair of mutually prime numbers is it alway the case that eventually one gets every number?

    (Hint: such an infinite list begins with zero (the empty sum). If one maintains a list of numbers consisting of those numbers not already listed (in the infinite list) which are the sum of a number already listed and a number in the given list, then the next number is the least number of this list.)

12. Write a program which returns the infinite list of ascending numbers whose prime factors lie in the set $\{3, 5, 7\}$ (Hamming).

13. Viewing an infinite list of infinite lists as a two dimensional infinite array, there is a way of enumerating the elements as a single infinite list by traversing the array in diagonals. Your task is to write a program which does this.

    One may break this down into two steps

    (a) Obtain an infinite list of non-empty lists ($A * list(A)$) from the two dimensional infinite array by collecting the diagonals into such a structure. (Why do you need non-empty lists?)

    (b) From an infinite list of non-empty lists produce a single infinite list.

2.4 The possibly infinite

Corresponding to each inductive datatype, which we may intuitively view as being finite, there is a coinductive datatype which is the possibly infinite version of that inductive datatype. To implement many algorithms in charity it is often necessary to transform data from its inductive form to its corresponding coinductive form and back again. In this latter form certain manipulations are more natural. We shall eventually demonstrate these ideas by looking at various algorithms. However, in this section we shall concentrate on introducing these "codatatypes" and their manipulations.

While it is very natural to transform data from an inductive to a coinductive datatype the reverse transformation is less natural. In fact, the technique for "collecting" a coinductive datatype back into its inductive form is an explicitly programmed traversal which has to have a bound. This is necessarily a rather technical piece of programming and it reflects the fact that the codatatype might be infinite so that in exploring it we must be very explicit about how far we will go.

2.4.1 The coNatural numbers

The first inductive datatype we met was the natural numbers: corresponding to these are the conatural numbers defined as follows:

```
data C -> conat = denat: C -> SF(C).
```

Another way of writing the datatype for the natural numbers is

```
data nat' -> C = ennat: SF(C) -> C.
```

Here `zero` is equivalently (though with some loss of clarity!) expressed `ennat(ff)` and `succ(zero)` as `ennat(ss(ennat(ff))` etc. To obtain the definition of `conat` from `nat'` we have simply reversed the arrows! This process of reversing arrows is called **dual**ing: the dual concept (datatype in this case) is traditionally named by prefixing a "co" onto the original name.

Notice, in the datatype definition of `conat`, we have used the success or fail datatype. This indicates that only *possibly* can we destruct again. One should think of the `conat`s as a box of floss from which one can pull a given length of string. As you are not able to see inside the box each time you pull you are never sure of whether there is more floss to come out or whether, in fact, you have exhausted the supply.

We may embed the natural numbers in the conatural numbers by:

```
def make_conat: nat -> conat
    = n => (| zero()  => denat: ff
            | succ(m) => denat: ss m
            |) n.
```

This is rather like preloading our box of floss with a given length of string.

It is of some interest is to note that this function is a map from an inductive type to a coinductive type, thus, we should be able to write the program as either an unfold, as above, since it ends at a coinductive, or as a fold as it starts at an inductive datatype. Here is the fold version of the program:

```
def make_conat': nat -> conat
    = n => {| zero: () => (denat: ff)
            | succ: m  => (denat: ss m)
            |} n.
```

Notice how it builds the datatype using records.

Before we push the analogy of floss boxes too far we should realize that the conatural numbers have an infinite element:

```
 def inf: 1 -> conat
    = () => (| x => denat:ss x |) ().
```

which is definitely not a natural number and would be a magic floss box which would never run out!

Just as one can add and multiply ordinary natural numbers, one can add and multiply conatural numbers:

```
def coadd(n,m) = (| ((denat:ff),(denat:ff))     => denat:ff
                  | ((denat:ff),(denat:ss m))  => denat:ss((denat:ff),m)
                  | ((denat:ss m),r)           => denat:ss(m,r)
                  |) (n,m).

def comult(n,m) = (| ( _ ,(denat:ff))           => denat:ff
                   | ((denat:ff), _ )           => denat:ff
                   | ((denat:ss(denat:ff)))
                     ,(denat:ss m))             => denat:ss(n,m)
                   | ((denat:ss(n)),r)          => denat:ss(n,r)
                   |)(n,m).
```

These definitions correspond quite closely to the recursive definitions:

$$
\begin{array}{lcl}
n+m: & & \\
0+0 & = & 0 \\
0+s(m) & = & s(0+m) \\
s(m)+r & = & s(m+r) \\
n\cdot m: & & \\
x\cdot 0 & = & 0 \\
0\cdot y & = & 0 \\
s(0)\cdot s(y) & = & s(n\cdot y) \\
s(x)\cdot r & = & s(x\cdot r)
\end{array}
$$

Notice in the last definition how one is decrementing the first number until it is 1 and then renewing it while decrementing the second number. This is an iterative way of defining multiplication.

Of some interest is the notion of zipping for the natural numbers:

```
def zip_conat(n,m) = (| ((denat:ff), _ ) => denat: ff
                      | ( _ ,(denat:ff)) => denat: ff
                      | ((denat:ss n),(denat:ss m)) => denat:ss(n,m)
                      |)(n,m).
```

for this is equivalent to taking the minimum of two numbers. It has been shown to be impossible in a formal system such as charity (while keeping to the first order fragment) to compute the minimum of two numbers in time proportional to the minimum ... essentially one is forced to choose the size of one of the inputs to drive the computation. This theoretical limitation is also related to the fact that zipping lists is also somewhat awkward in charity . However, zipping is an immediate and very natural operation for many coinductive datatypes.

2.4.2 Colists

The next consider that most common of datatypes: lists. Its sister datatype is that of colists (as given below). Colists are possibly infinite lists, while recall that lists necessarily have a finite length.

Starting with the definition of lists

```
data list(A) -> C = nil:      1 -> C
                  | cons: A * C -> C.
```

we may equivalently state it as

```
data list(A) -> C = enlist: SF(A * C) -> C.
```

which can now be dualed to obtain:

```
data C -> colist(A) = delist: C -> SF(A * C).
```

A colist behaves rather like a simple candy machine which does not have a window through which one can see its stock of candy. Each time you press the button either you will get a bit of candy or it will report that it is empty. As for conats one can have magic candy machines!

Our first step is to show how this candy machine can be stocked (in a finite way). This is the program for translating a list into a colist:

```
def make_colist =
    L => (| nil()      => delist: ff
          | cons(a,w) => delist: ss(a,w)
          |) L.
```

Can you write the fold version of this program?

On the other hand to collect a list from a colist is harder: it requires not only a colist as input but also a bound (a natural number). This ensures the program will not attempt to completely collect an infinite colist into a list.

Below is the fold for colists written using pattern matching on a composite type which holds the bits of a colist as it is destructed. The idea is to traverse down the colist collecting the elements into a list (last component of the state). Then the elements of this list are "popped off" and applied to the initial state using f_cons:

```
def fold_colist{f_nil: 1 -> C,f_cons: A * C -> C}:
              colist(A) * nat -> C
    = (coL,Bnd) =>
       p0  {| zero: () => (f_nil(),coL,[])
            | succ: (V,((delist:ff),[]))      => (V,(delist:b0()),nil))
```

```
                | (V,((delist:ff),[a|t]))   => (f_cons(a,V),(delist:ff),T))
                | (V,((delist:ss(a,CL)),t)) => (V,(CL,cons(a,t)))
            |} Bnd.

def collect_colist: colist(A) * nat -> list(A)
    = (CL,bnd) => fold_colist{nil,cons}(CL,bnd).
```

Notice that the bound actually has to be twice the length of the colist as we must traverse both down the colist breaking it into bits and then pop the elements off this created stack.

The complexity of folding over a colist may be enough to discourage you from extensive the use of colists. However, let us remind ourselves that some programs are more natural as colist programs. For example zipping colists is given by:

```
def zip_colist: colist(A) * colist(A) -> colist(A)
    = (L1,L2) =>
        (| (ss(a1,t1),ss(a2,t2)) => delist: ss((a1,a2),(delist t1,delist t2))
         |      _                => delist: ff
         |)(delist L1,delist L2).
```

Zipping lists is less natural: we must either mimic this which gets the result in the wrong order or fold on one of the lists which necessitates first reversing the other. There is no way out of this quandry (in the first order fragment of charity at least), which requires us to do a double traversal of the list.

```
def zip_list: list(A) * list(A) -> list(A)
    = (L1,L2) =>
  rev p1 {|  nil: () => ((L1,L2),[])
          | cons: ((cons(a1,L1),cons(a2,L2)),L) => ((L1,L2),cons((a1,a2),L))
                | ( _ ,L)                        => (([],[]),L)
          |} L1.
```

This begins to indicates some of the problems which arise when we wish to do a "simultaneous recursion" on an inductive datatype: these are more natural operations on coinductive types.

2.4.3 CoBTrees and zipping

One might think that "zipping" is restriced to natural numbers and lists, however, this is not the case. A BTree, recall, is a tree with its information at its nodes:

```
data BTree(A) -> C = bnode: C * (A * C) -> C
                   | btip: 1 -> C.
```

Corresponding to this datatype is the coinductive datatype:

```
data C -> coBTree(A) = deBTree: C -> SF(C * (A * C)).
```

Now to zip two coBtrees we simply overlay the trees pairing the information at the nodes, whenever both have information, and terminating the tree otherwise:

```
def zip_Btree: coBTree(A) * coBTree(B) -> coBTree(A * B)
    = (T,S) =>  (| ((deBtree: ss(t11,(a1,t12))),(deBtree: ss(t21,(a2,t22))))
                                => deBTree: ss((t11,t21),((a1,a2),(t12,t2)))
                 |        _     => deBTree: ff
                 |) (T,S).
```

2.4.4 Exercises

1. What are `coadd(inf,n)`, `comult(n,inf)`, and `zip(inf,n)`?
2. Is there a function for conatural numbers corresponding to `monus`? If so write it!
3. Write the fold version of `make_colist`.
4. Given an example of an infinite colist.
5. Write a function to turn an infinite list into a corresponding (infinite) colist.
6. Can you write a `coappend` function for colists?
7. Explain why you cannot write a `coreverse` or `coflatten` function for colists.
8. Can you write a wacky multiplication function for colists?
9. Can you write down the rules for when an inductive datatype is zippable?
10. Why is zipping (of say colists) not quite associative? Is there a "unit" for zipping? Is there a "zero" for zipping?
11. Write a function, which given the size of a cotree, will return successfully the tree. If the size is given incorrectly then it should fail. (Hint the "success or fail" datatype will be useful here. The idea is to write a collection function which traverses the tree).
12. We may represtent binary numbers as colists of booleans with the least significant bit first. Write routines for displaying (as a list of bits of a given length), for adding, and multiplying numbers so represented.
13. Write a `pushdown` function which inserts a new item into an ordered non-repeating colist of items. This function must maintain not only the order but also the non-repeating nature of the colist.

2.5 Where next?

We have now introduced nearly all the core constructs of charity ! What more is there to do? Alot! We must show how the language can be used to solve standard programming problems. This is the purpose of the remaining chapters of the first part: they are directed at particular problems such as implementing arithmetic, sequential processes, sorting, unification, and search.

Something the reader may find rather surprising is the extent to which the coinductive datatypes are used in the remainder of this part. Almost every significant program employs the coinductive datatypes and their properties in a non-trivial manner. Indeed, it quickly becomes apparent that it is only by extensively deploying these datatypes that a program of the correct complexity can be obtained!

When one considers the fact that coinductive datatypes are totally absent from almost all other programming languages it is reasonable to wonder at what is going on! The first thing that one should realise is that charity and its type system makes much sharper distinctions than traditional languages. In particular, charity distinguishes very sharply between what is "possibly infinite" and what is "definitely finite." This distinction forces one to be very disciplined about the patterns of recursion which can be employed with each type. Efficient algorithms, it turns out, are often obtained by judiciously mixing these patterns of recursion. In charity this means that one has to explicitly flip from one type to another to mimic the efficient algorithm.

There is, of course, a cost to maintaining these distinctions: an elegant (general) recursive algorithm – if it can be expressed at all in charity — may not be nearly as succinct or efficient. However, as we will discover, there is also sometimes significant value in being explicit about the patterns of recursion involved in an algorithm. The untangling of these patterns can sometimes suggest a clearer and cleaner view of what is happening.

Chapter 3

Examples of basic algorithms

A family of algorithms it is often viewed as difficult to implement in functional language are the so called dynamic algorithms. We open this chapter by discussing some simple examples of these. They provide examples where the phrases of the folds must necessarily have a complex form: the essence of a dynamic algorithms is that it encodes optimization information into a complex state.

There are very few large programs written which do not somewhere use a sorting algorithm. It is therefore appropriate that we discuss how to sort in charity . We show in this section that we can already achieve $O(n \log n)$ complexity by a merge sort. The sorting algorithms are particularly interesting as they use the coinductive datatypes in a non-trivial way.

The chapter ends with a brief discussion of searching in charity .

3.1 Dynamic programming

We shall look at three problems in this section which are traditionally solved by dynamic programming techniques. The first is the "CEO's wife's problem" and is particularly nice as the solution is rather elegant in charity ! More formally it is the problem of finding the largest independent set in a tree. The second problem is to find longest ordered subsequence of a sequence and the third, and last, is to find the longest common subsequence of two sequences.

3.1.1 Independent sets in trees

The wife of a CEO wishes to throw a party for her husbands firm. However, she wishes to avoid inviting both an employee and his/her boss in order to make it more convivial. Furthermore, she wants to invite as many people as possible – and if that means not inviting her husband she will simple arrange it when he is out of town.

An **independent** set of a graph (a set with a symmetric irreflexive relation) is a subset none of whose nodes are related. Thus, the problem expressed more abstractly is to find a maximal independent subset in the graph whose nodes are the people of the firm and whose relation is the employee-boss relation.

We shall start by determining the largest number of people she could invite to the party. Later we shall modify this codes to provide a list of people she should invite.

We shall need two auxillary functions, the first to sum the integers in a list the second to return the largest of two integers:

```
def SUM: list(int) -> int
    = L => {| nil: () => 0
            | cons: z => add_int z
            |} L.

def max_int: int * int -> int
    = (n,m) => { true => n
               | false => m
               } ge_int (n,m).
```

We shall store the hierachy of the firm as a degenerate Rose tree with no values at the nodes:

```
data DRoseTree -> C = bud: list(C) -> C.

def test1: 1 -> DRoseTree
    = () =>
      bud [ bud [ bud [ bud []
                      , bud []
                      , bud []
                      ]
                , bud []
                , bud [ bud [] ]
                , bud []
                , bud [ bud []
                      , bud [ bud [] ]
                      , bud []
                      ]
                ]
          , bud [ bud []
                , bud [ bud [] ]
                ]
          ].
```

To determine the size of the largest independent set we reason as follows: at each node we have two possibilities we either omit the node or include it. If we omit it we want to invite the maximum number of people from its subtrees. If we include it we cannot include any of its children (employees) so we must invite the maximum number of people from its grand-subtrees.

Thus, at each node we need to keep track of two numbers (n,m), where n is the maximum number of people if we include the node and m is maximum number of people if we exclude the node. Thus, at the bottom of the hierarchy, a worker with no employees to do his bidding, gets labelled $(1,0)$. A person with a list of employees whose subtrees are labelled $[(n_1,m_1),...,(n_r,m_r)]$ will be labelled as

$$(\sum_{i=1}^{r} m_i + 1, \sum_{i=1}^{r} \max(n_i,m_i)).$$

Notice that if we set the sum of an empty list to be 0 then the special case of a leaf will be also handled by this formula. Finally, at the top of the tree to find the largest independent set we take the maximum of the two numbers and the wife decides whether to include her husband or not!

Clearly, this is a fold over the Rose tree and this translates to the folloowing charity code:

```
def max_ind: DRoseTree -> int
   = T =>  max_int {| bud: L => ( succ_int SUM list{p1} L
                               , SUM list{max_int} L )
                    |} T.
```

Now to obtain a list of names we must modify this code. For example rather than choosing the maximum of two numbers we must choos e the longest of two lists:

```
 def max_list: list(A) * list(A) -> list(A)
    = (L1,L2) => { true  => L1
                 | false => L2
                 } ge (length L1,length L2).
```

Furthermore we need a datatype which allows names to be stored at nodes: this is a Rose tree

```
data RoseTree(A) -> C = rose: A * list(C) -> C.

def test2: 1 -> RoseTree(string)
     = () => rose("John", [ rose("Martha", [ rose("Trudy", [])
                                           , rose("Anthony", [])
                                           , rose("Rown", [])
                                           ] )
                          , rose("Casey", [])
                          , rose("Polly", [ rose("Robin", []) ] )
                          , rose("James", [])
                          , rose("Grayson", [ rose("Lee", [])
                                            , rose("Lisa", [ rose("Saul",[]) ])
                                                           , rose("Wayne", [])
                                                           ] )
                                            ] )
                          , rose("Susan", [ rose("Peter", [])
                                          , rose("Paul", [ rose("Mary", []) ] )
                                          ] )
                          ]).
```

Now the modified collection replaces summing by flattening, adding one by cons, and taking the mazimum number by taking the maximum list:

```
def MAX_IND: RoseTree(A) -> list(A)
    = T => max_list {| rose: (a,L) => ( cons(a,flatten list{p1} L)
                                      , flatten list{max_list} L )
                     |} T.
```

3.1.2 Finding the longest subsequence

The problem is as follows: given a list of integers find a sublist of greatest length which is ordered least first. This may seem like an easy problem but consider the following list:

$$[7, 4, 9, 5, 3, 8, 4, 6, 2, 7, 5, 11, 8].$$

Starting at the 7 we have a sublist $[7, 9, 11]$. Can we beat it? Starting at 4 we have $[4, 5, 7, 11]$; we notice that we may also skip the first 5 to pick up a six. However, we may obtain an even longer sequence starting at 3, $[3, 4, 6, 7, 11]$. This is a longest subsequence. Thus, the first least number may not generate the longest subsequence.

If we work from the back of the list to construct longest subsequences we notice that we may order the ordered sequences by their their first element and their length. A sequence which starts with a larger number can always replace a subsequence which is no longer but starts at a lower number. We say the first subsequence **dominates** the second. Thus, our strategy is to keep a list of dominant subsequences which can be used to arrive at a longest subsequence.

To grow this list of dominant sublists we must determine what happens when we add an element to the parent list. If we have a set of domininant sublists we can consider lengthening each (if this is possible) by the new element. Each new sublist could then become the dominant ordered sublist at its new length and we can determine this by comparing it to the old dominant sublist at that length.

To program this we first need a program to select the better of two lists as determined by the largest first element:

```
def better: list(int) * list(int) -> list(int)
    = (cons(n1,L1),cons(n2,L2)) => { true  => cons(n1,L1)
                                   | false => cons(n2,L2)
                                   } ge_int (n1,n2)
    | (nil,v) => v
    | (v,nil) => v.
```

Next we must be able to lengthen a sequence by adding an element: if the element is greater than the first element we simply use the convention of returning an empty list.

```
def lengthen: int * list(int) -> list(int)
    = (n,nil) => nil
    | (n,cons(m,L)) => { true  => cons(n,cons(m,L))
                       | false => nil
                       } ge_int (m,n).
```

Now if we have a sequence of dominant sublists of decreasing lengths and we add an element to the parent list we can update the dominant lists by lengthening each and comparing the result to the old dominant list at the greater length.

```
def best_subseq: int * list(list(int)) -> list(list(int))
    = (n,L) => { (nil,L') => L'
               | z => cons z
               } {| nil: () => ([n],[])
                  | cons: (v,(v',L)) => (lengthen(n,v), cons(better(v,v'),L))
                  |} L.
```

A longest subsequence is the dominant subsequence of greatest length thus the code for finding the longest subsequence is:

```
def MAX_subseq: list(int) -> list(list(int))
    = L => { cons(V,_) => V
           | _ => nil
           } {| nil: () => []
              | cons: z => best_subseq z
              |} L.
```

Note the complexity of this is $O(n^2)$ as at each iteration we must update the list of dominant sublist.

3.1.3 Finding the longest common subsequence

The problem is given two sequences of numbers to find the longest common subsequence. To see how to accomplish this we think of a structurally recursive algorithm on the first list. We shall hold at each state a sequence of partial matches to the second list. These partial matches will consist of pairs of lists (R, L) where L is the common list so far and R is the prefix of the list which is still unmatched. As we add an element to the first list we must modify the partial matches. This we do by modifying R to the prefix before the last occurrence of that element (see `prefix_to` below, and adding the element to the list recording the common sequence.

```
def prefix_to: A * list(A) -> SF(list(A))
    = (n,L) => {| nil: () => ff
                | cons: (a,ss L) => ss cons(a,L)
                      | (a,ff) => { true => ss []
                                  | false => ff
                                  } eq_int(a,n)
                |} L.

def lengthen_PM: int * (list(int) * list(int)) -> SF(list(int) * list(int))
    = (n,(L1,L2)) => { ss L => ss( L, cons(n,L2))
                     | ff => ff
                     } prefix_to (n, L1).
```

Now a partial match dominates another partial match with the same length common sequence whenever the available prefix to be searched is smaller. This gives the test:

```
def better_PM:
 (list(int) * list(int)) * (list(int) * list(int)) -> list(int) * list(int)
    = (P1,P2) => { true  => P1
               | false => P2
               } gt (length p0 P1, length p0 P2).
```

Every time we add a new element to the first list we need to update the partial matches so that at each length we are holding the best partial match.

```
def modify_PMS:
 int * (list(list(int) * list(int)) * list(int)) -> list(list(int) * list(int))
    = (n,(LPM,L2)) =>
                 { (ff,L') => L'
                 | (ss V,L') => cons(V,L')
                 } {| nil: () => (lengthen_PM(n,(L2,[])),[])
                    | cons: (V,(ss W,L)) =>
                                (lengthen_PM(n,V),cons(better_PM(V,W),L))
                          | (V,(ff,L)) => (lengthen_PM(n,V),cons(V,L))
                    |} LP.
```

Finally we find the greatest comon subsequence by selecting the longest partial match after iteratively adding elements to the parent list:

```
def common_subseq:  list(int) * list(int) -> list(int)
    = (L1,L2) =>
       { cons(V,_) => p1 V
       | _ => []
       } {|  nil: ()     => []
          | cons: z => modify_PMS z
          |} L1.
```

This is actually a rather inefficient implementation. One problem is that the step of for finding a prefix which requires the processing of the whole list. Also the equality tests require processing the list. This makes this $O(n^3)$.

3.2 Sorting

Perhaps one of the most common operations performed on a list of items is to sort them. In this section we investigate several sorting operations starting with some näive algorithms (using inductive datatypes) and progressing to more sophisticated algorithms (using coinductive datatypes).

It is well known that a list of n items in random order can be sorted in order $n \log n$ (and that this complexity cannot be bettered). Of course, näive algorithms, such as the selection and insertion sort below, do not achieve this complexity as they are order n^2. The objective of this section is therefore to show how one can write efficient sorting algorithms in charity which achieve the desired lower bound on complexity. It is of some interest to realize that these efficient algorithms rely fundamentally on the use of coinductive datatypes.

We start our discussion of sorting with the selection and insertion sort. Both these are order n^2 algorithms and, thus, are not very efficient for producing sorted lists. However, one should not dismiss them out of hand as they are quite effective in situations where the complete list may never be required and can be lazily produced.

To bring the complexity of producing a sorted list down it is necessary to resort to divide and conquer algorithms which break the list down into sublists recursively. In charity this corresponds to generating a coinductive tree. These techniques are illustrated by the implementation of the quick-sort and merge-sort which are discused below.

3.2.1 Order relations

A (partial) order relation $x \leq y$ on a set is a transitive, reflexive relation which is anti-symmetric in the sense that if $x \leq y$ and $y \leq x$ then $x = y$. An order relation is total in case for each pair (x, y) either $x \leq y$ or $y \leq x$. Normally when one discusses sorting algorithms one does so with a total order (relation) in mind.

In a (decidable) relation on a type `A` may equivalently be represented as a map $A * A \longrightarrow bool$. When one discusses sorting algorithms one really does so with respect to an arbitrary decidable relation represented as a boolean predicated as it is hard to legislate (through just a type system) that the relation actually must be a total order relation.

Before we start discussing sorting algorithms it is useful to have available a test set of values equipped with an order relation. A simple way to generate such a test set and its order relation is to specify one as follows:

```
data VAL -> C = aa|bb|cc|dd|ee: 1 -> C.

def leq_VAL: VAL * VAL -> bool
    = (aa,_)  => true
    | (bb,aa) => false
    | (bb,_)  => true
    | (cc,aa) => false
    | (cc,bb) => false
    | (cc,_) => true
    | (dd,ee) => true
    | (dd,dd) => true
    | (dd,_) => false
```

```
    | (ee,ee) => true
    | (ee,_) => false.
```

With some justification you may feel that this cannot generate challenging test examples. However, given an ordered type A there is a standard way of generating an ordering on the type $list(A)$. This is called the **lexicographical** ordering: it works by ordering on the first item which disagrees – if there is no such disagreement then the shorter list is taken first.

Here is a program for giving the lexicographical ordering on lists of items which are already ordered:

```
def lex_ord{leq: X * X -> bool}: list(X) * list(X) -> bool
    = (x,y) => { ((true,true),(nil,_)) => true
               | ((true,true), _)      => false
               | ((v,_),_)             => v
               } {| zero: () => ((true,true),(x,y))
                  | succ: ((true,true), (cons(hx,xs),cons(hy,ys)))
                             => ((leq(hx,hy),leq(hy,hx)),(xs,ys))
                        | S => S
                  |} length x.
```

We shall soon discover a more efficient (and more natural way) of writing this. However, in the meantime this does put at our disposal an infinite collection of ordered types:

```
           VAl   with  leq_VAL
     list(VAL)   with  lex_ord{leq_VAL}
list(list(VAL))  with  lex_ord{lex_ord{leq_VAL}}
                 ....
```

We may also use the built-in ordering on the standard types of integers and strings.

3.2.2 Selection Sort

The simplest way (and most inefficient way) of sorting a list is to select, in succession, the largest element out of that list while putting the selected elements in sequence into a new list.

```
def select{P: A * A -> bool}: list(A) -> SF(A * list(A))
    = L => {| nil : ()                => ff
            | cons: (a, ff)      => ss(a, [])
                  | (a, ss(b,L)) => { true  => ss(a, cons(b, L))
                                    | false => ss(b, cons(a, L))
                                    } P (b, a)
            |} L.

def select_sort{P: A * A -> bool}: list(A) -> list(A)
    = L => p0 {| zero: ()    => (nil,L)
               | succ:(AL, L) => { ff       => (AL, L)
                                 | ss(x, L) => (cons(x, AL),L)
                                 } select{P} L
               |} length L.
```

This, of course, is not very efficient: the operation of selecting has to traverse the list to return the largest element and the remainder of the list. This selection operation has to be then done repeatedly until the list is empty. Thus, there are $\frac{n\cdot(n-1)}{2}$ comparisons.

If we had been required to produce a colist we could alter the last step to the following:

```
def coselect_sort{P: A * A -> bool}: list(A) -> colist(A)
    = L => (| L' => delist: select{(a,b) => P(b,a)}(L') |) L.
```

As the colist is lazily evaluated, there could be some advantage to this especially if only the first few values will ever be accessed. Notice to obtain a colist in the correct order it was necessary to reverse the order relation.

3.2.3 Insertion Sort

Another simple way to sort a list is to create a new list by inserting the elements of the old list into it in such a manner as to maintain order. This is the basic idea behind an insertion sort. Here is a first stab at it in charity :

```
def insert{P: A * A -> bool}: A * list(A) -> list(A)
    = (v,L) => cons {| nil:()           => (v,nil)
                     | cons:(x,(y,RL)) => { true  => (x,cons(y,RL))
                                          | false => (y,cons(x,RL))
                                          } P(x,y)
                     |} L.

def insert_sort{P: A * A -> bool}: list(A) -> list(A)
    =  L => {| nil:()       => nil
             | cons:(x, RL) => insert{P}(x, RL)
             |} L.
```

Note that in this algorithm on each insertion the entire list (so far) is traversed. This gives an order $\frac{n\cdot(n-1)}{2}$ algorithm. However, notice that when we insert all that is actually required is to push the new element down into the list as far as it is needed. Thus, the program could be improved in order to, on average, halve the number of comparisons!

We notice also that the insertion can conveniently be viewed as a coinductive operation. Let us therefore write the program to produce a colist:

```
def pushdown{P}
    = (v,CL) =>
     (| (ss v,(delist:ff))        => delist: ss(v,(ff,ff))
      | (ss v,(delist:ss(a,CL)))  => { true  => ss(a,(ss v,CL))
                                       | false => ss(v',(ff,(delist:ss(a,CL))
                                       } P(a,v)
      | (ff,(delist:ss(a,CL))     => delist: ss(a,(ff,CL))
      | (ff,(delist:ff))          => delist: ff
      |) (ss v,CL).
```

```
def coinsert_sort{P}
    = L => {|  nil:()      => nil
            | cons:(x, RL) => pushdown{P}(x, RL)
            |} L.
```

Notice that in this case when the value is inserted the rest of the colist is returned more directly in the last pattern thread of the unfold. Of course, it is important to realize that we cannot simply return the rest of the colist as this would give a state of the wrong type.

Of course we have also introduced the complication that we are now returning a colist. To reobtain a list we would have to collect the colist. There are situations where having a sorted colist is quite desirable: for example when the elements to be sorted are generated in the course of a computation, it is convenient to be able to simply insert them into a colist. In fact, we shall shortly consider the example of maintaining an agenda for a search where this sort of insertion, or `pushdown`, operation is useful (as is the fact that it can be carried out lazily).

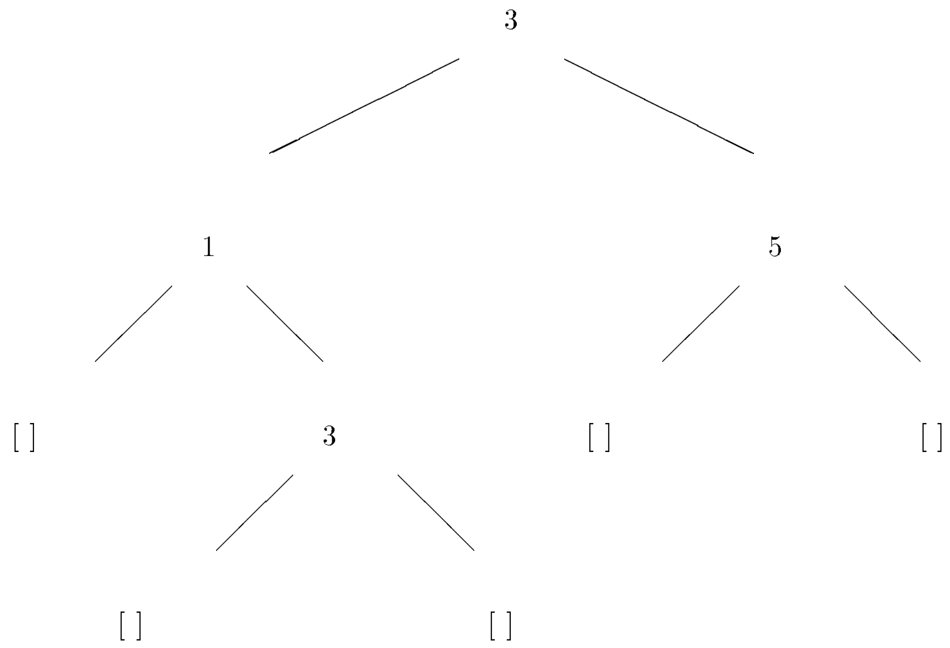
This is not to say that this is the most efficient method of maintaining an agenda: it is not. Divide and conquer techniques can also apply in this arena to produce more efficient algorithms.

Before leaving these two methods of sorting it is worth comparing them. In the selection sort the greatest element of an unsorted list must be extracted. In the insertion sort an element must be added to an already ordered list. In the latter case we may use the fact that the list we are manipulating is already structured to advantage: this, on average, halves the number of required comparisons (although notice that so far in our implementations there is still a residual overhead for the rest of the list). However, for the selection sort - and this is why the selection sort is potentially the less efficient - we cannot assume anything about the list from which the least element is to be extracted. Thus, we must traverse the whole list.

3.2.4 Quicksort

The quicksort algorithm is the first divide and conquer algorithm we consider. The idea behind the algorithm is to remove (in some manner) an element, called the pivot, from the unsorted list and to split the list into two sublists according to whether the other elements are less than or greater than that element. By repeatedly doing this on each sublist we will eventually reach the stage where the produced sublists are both empty. In fact, we will have created a tree of splittings with the pivot elements at the nodes: by traversing the this tree and collecting the node values in infix order we will obtain a sorted list.

The greatest gain in efficiency is when the list is divided equally: thus, the choice of pivot element is crucial to the effectiveness of the algorithm. If the list is divided in half each time then the algorithm is of order $n \log n$. In the worst case, however, the algorithm can be order n^2. For this reason often extra effort is made to ensure a sensible choice of pivot element. Here, however, we shall be content to pivot about the first element. In this case the list $[3, 1, 5, 3]$ will produce the following coinductive tree structure:



This process of building a tree is essentially an unfolding of a coinductive datatype. Therefore, the natural way to implement this in charity is to use a coinductive datatype. Unfortunately the very next step, the traverse of the tree, is naturally inductive in nature, and would most naturally implemented as a fold. However, at that stage we would have in our hand a (possibly infinite) coinductive datatype. Fortunately, the size of the tree generated is determined by the length of the list. Thus, even though we cannot fold we may collect the values.

We start by declaring a cotree structure, called a `cobtree`, which holds information at its nodes:

```
data C -> coBtree(A) = debtree: C -> SF(A * (C * C)).
```

The first stage in the quicksort algorithm is to split the list using a pivot: here, as mentioned earlier, we shall simply use the first element.

```
def pivot{P: A * A -> bool}: list(A) -> SF(A * (list(A) * list(A)))
    = nil       => ff
    | cons(a,L) => ss {|  nil:            () => (a,(nil,,nil))
                       | cons: (v,(a,(L,R))) => {  true => (a,(L,cons(v,R)))
                                                | false => (a,(cons(v,L),R))
                                                } P(a,v)
                       |} L.
```

Using this pivot function we can now grow a `cobtree`:

```
def grow_btree{P: A * A -> bool}: list(A) -> coBtree(A)
    = L => (| x => debtree: pivot{P}(x)
            |) L.
```

The final stage in the algorithm is to gather a sorted list by traversing the tree in infix order. We do this by inventing a little machine to iteratively break the cobtree into bits while collecting the values at the nodes in the right order.

This machine has three components: a list into which the answer is collected, an active subtree, and a dump stack of tree bits. The tree bits have two components a tree and a value. Depending on the structure of the active tree and the dump, one of three actions is taken:

Push: If the active tree is a node with a value, we push the left tree and the node value onto the dump, while making the right tree active.

Pop: If the active tree is a tip (that is an end of the tree) then we pop the top value of the dump stack adding the node value to the head of the answer list and making the tree the active tree.

Stop: If the active tree is a tip and the dump is empty we have finished.

It is not hard to see that the effect of this is to traverse the tree while collecting the node values in infix order as desired. In charity this can be expressed as:

```
def Bcollect: coBtree(A) * nat -> list(A)
    = (t,n) => p0
  {| zero:        () => ([],(debtree t,[]))
   | succ: (L,(ff,nil)) => (L,(ff,nil))                        % stop
         | (L,(ff,cons((t,a),D))) => (cons(a,L),(debtree t,D))  % pop
         | (L,(ss(x,t),D)) => (L,(debtree t,cons(x,D)))         % push
   |} n.
```

where the machine described above is used in the succ phrase of the fold.

This now gives us all the steps in the quicksort algorithm:

```
def quicksort{P: A * A -> bool}: list(A) -> list(A)
    = L => Bcollect(grow_btree{P} L
                   ,succ {x=>add(x,x)} length L
                   ).
```

where the bound on the collection algorithm is the size of the tree generated (which is $2 \cdot n + 1$ where n is the number of leaves).

3.2.5 Merge Sort

The quicksort cannot guarantee $n \log n$ performance so we end this discussion of sorting with an algorithm which does: the merge sort. Again this is a divide and conquer algorithm: the idea this time is to split the list into two sublists of equal (to within one) length and then to repeat this procedure on the sublists until singleton lists are obtained. Thus, again we are implicitly generating a tree. However, this time there are no comparisons in the tree generation stage: its objective is simply to split the list into equally parts. Clearly, again, this is an unfolding to a coinductive datatype.

The collection stage of the algorithm requires one to merge ordered lists from the leaves of the tree working upwards. Clearly this is a fold operation ... except (as for quicksort) we would be holding at that stage a coinductive datatype. Again, the length of the list to be sorted gives a bound on the size of this tree and thus the collection stage can be written as bounded fold on the cotree datatype.

We shall write this sorting algorithm for a colist paired with its length. One reason for doing this is that the merging operation is quite natural for colists but rather more awkward for lists. Merging has a similar problem to zipping: in both cases one would like to simultaneously recurse through the lists and this involves an unwanted reversal in the first-order fragment of charity . As we shall shortly discover, in the higher-order fragment this problem can be circumvented quite cleanly.

Our first problem is to divide up a colist into two equal subcolists. This we can achieve by forming the colist of elements with even depth and the colist of elements with odd depth:

```
def even: colist(A) -> colist(A)
    = CL => (| ss(a,(delist: ss(_,CL)) => delist: ss(a,delist CL)
             | ss(a,(delist: ff)       => delist: ss(a,ff)
             | ff                      => delist: ff
             |)  delist CL.

def odd: colist(A) -> colist(A)
    = (delist: ss z) => even p1 z
    | z              => z.
```

To build a tree from these splittings we need to declare a suitable coinductive datatype. We shall create a cotree with only two possible outcomes: a fork at which the current list can be split into two non-empty lists and a bud when the current list is a singleton. In order to create such a datatype it is first necessary to have a datatype which allows either a bud or a fork. This is a simple inductive datatype which we call `tSUM`. This datatype then allows us to build the coinductive datatype we need:

```
data tSUM(A,B) -> C =  bud: A -> C
                    |  fork: B * B -> C.

data C -> cotree(A) = deT: C -> tSUM(A,C).
```

In fact this datatype is (as the name suggests) the dual of the `tree` datatype. This we can see if we write the tree datatype employing the `tSUM` datatype:

```
data tree(A) -> C = enT: tSUM(A,C) -> C.
```

Our next task is to generate the cotree from the list using the splitting into odd and even. This, of course, is just an application of the unfold to the splitting function:

```
def split: A * colist(A) -> tSUM(A,A * colist(A))
    = (a,(delist: ff))      => bud(a)
    | (a,(delist:ss(b,CL))) => fork((a,even(CL)),(b,odd(CL))).

def make_cotree: A * colist(A) -> cotree(A)
    = (a,CL) => (| (a',CL') => deT: split(a',CL')
                |)(a,CL).
```

At this stage we have the cotree: we must now do a bounded fold over this cotree by merging ordered colists from the bottom up. Before we can do this we will certainly need the functions with which we intend to fold. Furthermore, and more challengingly, we shall have to develop a bounded fold combinator for cotrees.

We shall start by writing the merging operation. This requires one to create a new colist from two ordered colists. This can be done by pulling off the least elements of the two colists repeatedly. This would be a selection sort except for the fact that the two colists we are merging are ordered: this means that the least element of each colist is its first element. Thus, the merging simply involves selecting between the two first elements repeatedly. This gives the following code:

```
def merge{P: A * A -> bool}: colist(A) * colist(A) -> colist(A)
    = (CL1,CL2) => (| (ff,ff)   => delist: ff
                    | (ss(a,CL),ff) => delist: ss(a,(delist CL,ff))
                    | (ff,ss(b,CL)) => delist: ss(b,(ff,delist CL))
                    | (ss(a,CL1),ss(b,CL2))
                         => delist: { true  => ss(a,(delist CL1,ss(b,CL2)))
                                    | false => ss(b,(ss(a,CL1),delist CL2))
                                    } P(a,b)
                    |)(delist CL1,delist CL2).
```

Probably the most ticklish aspect of this implementation is the folding operation on cotrees. We will use a technique very similar to that used for the collection stage of the quicksort. That is we shall invent a machine which, if run for long enough, will deliver the answer. We shall, in fact, translate the tree structure into polish notation onto a dump stack which acts on a list of values. This will allow us to arrange for the merging operations to be applied to the correct sublists.

This machine consists of two components: a value stack and a dump stack. On the dump stack there will be two sorts of things: bits of trees and markers to indicate when the operation corresponding to a fork should be applied. The machine has four basic operations:

Pop bud: If the tree bit at the head of the dumpstack is a `bud` then the bud operation is applied to this value and the answer is pushed onto the value stack.

Push fork: If the tree bit at the head of the dumpstack is a `fork` then its subtrees are extracted and an operation marker followed by these tree bits are pushed onto the dump stack.

Pop Mark: If an operation marker is at the head of the dump stack then the fork operation is applied to the head two values of the value stack and the answer is pushed onto the value stack.

Stop: When the dump stack is empty the machine has finished.

This has to be written inside a bounded iteration. We shall assume that we provide the fold with the number of leaves of the tree. The number of required of iterations the machine is: one iteration for each leaf, two iterations for each fork (once to push a mark and once to operate on the mark). Thus, if there are n leaves we would have to iterate $2 \cdot (n+1) + n$ times.

```
data treeBIT(A,B) -> C = tFORK: 1 -> C
                       | tBIT: B -> C.
```

```
def cotree_fold{f0: A -> C,f1: C * C -> C}: cotree(A) * nat -> SF(C)
= (T,n) => { ([A],[]) => ss(A)
           | _ => ff
           } {|zero: () => ([],[tBIT(deT T)])
              |succ: (L,cons(tBIT(bud x),D)) => (cons(f0 x,L),D)
                                               % Pop bud
                   | (L,cons(tBIT fork(t1,t2),D))
               => (L,cons(tBIT deT t2,cons(tBIT deT t1,cons(tFORK,D))))
                                               % Push fork
                   | (cons(x,cons(y,L)),cons(tFORK,D)) => (cons(f1(y,x),L),D)
                                               % Pop mark
                   |  z => z
                                               % Stop
             |} {x => add(x,add(succ x,succ x))} n.
```

Notice that the fold opertion is written to end in the success or fail state. In this manner we can trap the cases where an insufficient number of iterations have been performed.

Finally, we can complete the merge sort: here we provide it in its raw form as a function from colists to success or fail colists:

```
def merge_sort{P: A * A -> bool}: colist(A) * nat -> SF(colist(A))
    = ((delist: ff),_) => (delist: ff)
    | ((delist:ss z),n) =>
         cotree_fold{v => (delist: ss(v,(delist:ff)))
                    ,merge{P}
                    }(make_cotree z,n).
```

The quicksort has the following desirable feature, the comparisons are done in the unfolding stage making the collection stage trivial. In the merge sort the unfolding operation does not involve comparisons, however, the folding stage does as it involves merging colists.

3.2.6 Exercises

1. Rewrite the insertion sort for inductive datatypes to avoid unnecessary comparisons.
2. Write a program to test whether a list is in order.
3. What is the precise complexity of the selection sort?
4. What is the precise complexity of the two insertion sorts as given above. Justify your calculation and check them by running the programs.
5. An incredibly silly way of ordering a list is to generate all the permutations of the list and then filter out those which are not in order. You will need the program to determine whether a list is in order and one to generate all the permutations!
6. Modify the given merge sort to be a function from lists to lists.
7. What is the complexity of the mergesort? Do the argument carefully!
8. Write a merge sort for lists: this entails writing a new serparate and merge routine. What are the problems with writing it this way?
9. Can you write a quicksort using only inductive datatypes? What is its complexity.

3.3 Basic search

In a sense the compexity of a problem is determined by how much blind search must remain in any solution to the problem. In attacking a search problem with success there are three crucial aspects to consider: what the problem gives you, what serendipity may provide, and the coverage raw computational speed can buy. Your aim is to use to best advantage these aspects in solving a problem in which blind exploration has an inevitable role.

It may seem that charity would be a disasterous language in which to try and express programs to solve search problems. The fact that we must write programs which terminate suggests search (surely an unbounded activity) would be impossible to express.

Of course, a good search must terminate (if only in failure) so that these arguments have little bearing on the suitability or no of charity . Indeed, we may use the coinductive datatypes of charity to capture generically some of the common aspects of search problems. So the purpose of this section is to show that it is possible to express searh problems quite economically in charity .

3.3.1 What is a search problem

While search problems come hidden in many forms their basic form is quite simple. They are partial automata with a start state and a set of goal states. Being a little more formal about this we may describe them in the following manner:

A **search problem** $\mathcal{P} = (S, A, P, \alpha, s_0, G)$ consists of

States: A set S of states which capture the static information of the problem,

Actions: A set A of actions which can be used to alter the state,

Permission set: A subset $P \subseteq A \times S$ of label state pairs which represent all the permissible labels (or legal actions) at states,

Transition: A map $\alpha : P \longrightarrow S$ from the permission set back to the set of states which indicates the state change induced by a permitted action,

Initial state: An initial state s_0 in which the problem starts,

Goal: A subset $G \subseteq S$ of the states into which it is required to drive the system.

In the study of a dynamical systems the notion of **state** is fundamental. The state of a system must contain sufficient information to allow the prediction of how the system will react to inputs (or external actions). A search problem may be viewed as a dynamical system which is being driven by external actions (which are under your control).

It is perhaps worth emphasizing at this stage that there is no assumption that any of the state, action, permission, or goal set are finite. In fact, in many search problems these will not be finite sets. Thus, although finite automata certainly give rise to search problems it is often the case that a search problem will have no (reasonable) finite presentation.

For example, in a board game such as peg solitaire the possible board positions are the states; the goal may be to remove by the action of jumping all but one of the pegs. Here the number of states and actions is finite albeit not something to be enumerated in a hurry! In a theorem prover the current deduction or theorem may be the state of the system, and an action might be a deduction which can be made from that theorem. In this case neither states nor actions are finite.

In a search problem the aim, of course, is not simply to understand how the system evolves but to derive a sequence of inputs which will drive it into a desirable state. A **solution** to a search problem is, thus, a list of actions p which can be used to drive the problem from its start state into one of its goal states. That is a list of actions which when used as inputs are always permitted and drive the problem from the initial state into a goal state.

To turn these rather abstract ideas into something which can be programmed we need to make some simplifying assumptions. The first is that the problem is **finitary**, that for each $s \in S$, the set $\{a \in A | (a, s) \in P\}$ is finite. That is at each state there are only a finite number of permissible actions (note this does not mean that the total set of actions must be finite).

When a search problem is finitary the partial map α can be equivalently expressed as a map:

$$\beta : S \longrightarrow list(A \times S)$$

which associates with each state the list of permissible actions together with the state into which the system is driven by that action. The one map β then encodes both the permission set and the action.

The second simplifying assumption is that the search problem is **decidable**: that is there is a function $goal : S \longrightarrow bool$ such that

$$x \in G \Leftrightarrow goal(x) = true.$$

This is a rather esoteric assumption: almost all practical search problems will satisfy this. However, there are some simply stated search problems (e.g. the search for Turing machines which correctly implement solutions to given problems: recall that the halting problem is undecidable) which are not decidable. In an undecidable search problem it is hard to know when you have have a solution: a rather undesirable feature!

These simplifying assumptions allow a search problem to be viewed as an element of a coinductive datatype:

```
data C -> ProblemTree(A,B) = evolve: C -> list(A * C)
                           |   goal: C -> bool.
```

Based on this datatype we shall shortly develop some basic search routines.

3.3.2 Elizabeth Farm

An example of an, albeit extremely simple, search problem is the following "Elizabeth Farm"[1] puzzle.

```
 -----------------------  --
|         |      |          |
|    BR      WC  |    KT    |
|         |      |          |
|----  --------------  -----|
|         |      |          |
|    LR      HA       DR    |
|         |      |          |
 -----------   -------------
```

Consider the problem of a robot trying to get from the kitchen to the bathroom in (a simplified) Elizabeth farm. Each room is a state and the outside is treated as a state. The robot can move north, south, west, or east and we ignore the complications introduced by being outside the house. The problem may be represented as follows:

The states can be represented by an inductive datatype which enumerates the rooms:

```
data eliz_rm -> C =  BR|DR|LR|KT|HA|WC|OUT: 1 -> C.
```

The labels are the moves which can be made, namely going north, south, east, or west. These can be represented by a similar datatype.

```
data compass -> C =  No|So|Es|We: 1 -> C.
```

Next we need to represent the possible moves that can be made from each state. This we do by associating with each room a list of possible moves represented as label state pairs:

[1]Elizabeth farm is an actual farm: it is John MacArthur's wife Elizabeth's farm. John MacArthur was an infamous officer - and later settler - in the New South Wales regiment in early Australian history. He introduced sheep farming to Australia, made his fortune, revolted against the then Govenor of the colony, who was none other than William Bligh of Bounty fame. Bligh brought MacArthur to trial, but the long trial in England failed to exact the retribution for the mutiny which he sought. Indeed, MacArthur used the time in England to great effect and advanced his wool trading interests. Elizabeth, during that crucial period, was left behind to run the farm.

```
def eliz_moves: eliz_rm -> list(compass * eliz_rm)
    = BR => [(So,LR),(Es,WC)]
    | LR => [(No,BR),(Es,HA)]
    | DR => [(No,KT),(We,HA)]
    | KT => [(No,OUT),(So,DR)]
    | HA => [(We,LR),(Es,DR)]
    | WC => [(We,BR)]
    | OUT => [(No,HA),(So,KT)].
```

We indicate the goal states by a predicate on the states:

```
def eliz_goal: eliz_rm -> bool
    =  WC => true
    |  _  => false.
```

Finally we can wrap up this search problem as an element of the coinductive data `ProblemTree`:

```
def eliz_farm
    = () =>  (| r => evolve: eliz_moves r
              |       goal: eliz_goal r
              |) OUT.
```

3.3.3 Simple search functions

Our objective in this subsection is to develop a simple yet general purpose search procedure. To do this we must develop functions which determine the next frontier of a breadth-first search from the last frontier. We shall do this in two steps as we must be careful to carry along the path which has been traversed to that stage.

A useful improvement is to prune the possible paths to avoid searching paths whose effect will be explored elsewhere or are simply known to be unproductive. This we introduce as a predicate *pred* on the proposed move and the path so far. For example in Elizabeth farm we may stop ourselves from going in circles simply by examining the path.

```
def frontier_step{P:A * list(A) -> bool}:
       ProblemTree(A) * list(A) -> list(ProblemTree(A) * list(A))
    = (prob,path) => {|nil:() => nil
                      |cons:((a,s),L) =>
                                 { true => cons((s,cons(a,path)),L)
                                 | false => L
                                 } P (a,path)
                      |} evolve prob.

def next_frontier{P:A * list(A) -> bool}:
  list(ProblemTree(A) * list(A)) -> list(ProblemTree(A) * list(A))
    = L => flatten list{frontier_step{P}} L.
```

We may now set up an infinite list of frontiers. In each frontier we may search for a goal state using:

```
def find_soln: list(ProblemTree(A) * list(A)) -> SF(list(A))
     = Front => {| nil: () => ff
                 | cons:((p,Prob),ff) => { true => ss p
                                         | false => ff
                                         } goal Prob
                       | (_,ss p) => ss p
                 |} Front.
```

This allows us to provide an infinite list which tells us whether there are solutions at each depth of search.

```
def search{P:A * list(A) -> bool}: ProblemTree(A) -> inflist(SF(list(A)))
    = prob => (| S => tail: next_frontier{P} S
               |      head: find_soln S
               |) cons ((prob,[]),[]).
```

Given this it is easy to develop a function which determines whether there is a solution at or below a given depth of search.

Exercises

1. Write a predicate to filter the paths which will stop the robot in Elizabeth farm from going round in circles! You may have to address the complication of going outside.

2. Three missionaries and three cannibals set out on a journey. On their travels they encounter a crocodile infested river. Looking about they find a small bark canoe which can hold no more than two people. The problem they confront is that if, in the process of ferrying themselves across the river, there are more cannibals than missionaries on either bank of the river (including when the boat unloads and loads) it is likely that the cannibals will revert to their normal behavior and make a meal of the out-numbered missionaries. Can they cross the river while avoiding such an outcome and if so how?

   Write a search program to discover how they surmount their problems.

3. A farmer with his hen, a bag of corn, and his dog comes to a river. There is an extremely small boat which allows him to take at most one of his possessions at any one time across the river. The complication is that he cannot leave the dog and the hen unsupervised on a bank (as the dog will most likely eat the hen), nor can he leave the hen and the corn unsupervised (as the hen will probably eat the corn). How does he get his possessions across the river without mishap?

   Write a search program to discover whether the farmer can surmount his problem.

4. Sometimes mathematical results can be versed as puzzles:

   By iteratively removing pairs of elements which sum to a prime, the set of numbers between (and including) 1 and $2 \cdot n$, $D_n = \{i | 1 \leq i \leq 2 \cdot n\}$, can be reduced to the empty set.

   So one may take $n = 2$ and try it out. So, for example, the set $D_2 = \{1, 2, 3, 4\}$ can be so reduced by first removing $\{1, 2\}$ and then $\{3, 4\}$, or by first removing $\{3, 4\}$ and then $\{1, 2\}$, or by first removing $\{2, 3\}$ then $\{4, 1\}$, etc.

   But can you always do this? Write a program to solve these puzzles.

5. The once very popular eight puzzle has a three by three board inhabited by eight sliding tokens: so one square is left blank. The tokens are numbered one through eight and the objective is to slide them into order from their starting configuration.

6. Write a general search routine to determine whether there is a solution below or at a given depth of search.

Chapter 4

Processes