17
Infinite data structures and lazy programming

Combining functions as first-class values, algebraic data types, and references enables programming with infinite data structures, the surprising topic of this chapter. We'll build infinite lists (streams) and infinite trees. The primary technique we use, lazy evaluation, has many other applications.

17.1 Delaying computation

OCaml is an EAGER language. Recall the semantic rule for function application from Chapter 13:

$$\frac{\begin{array}{l} P \Downarrow \texttt{fun } x \texttt{ -> } B \\ Q \Downarrow v_Q \\ B[x \mapsto v_Q] \Downarrow v_B \end{array}}{P\ Q \Downarrow v_B} \quad (R_{app})$$

According to this rule, before generating the result of the application (by substituting into the body expression B), we *first* evaluate the argument Q. Similarly, in a local `let` expression,

$$\frac{\begin{array}{l} D \Downarrow v_D \\ B[x \mapsto v_D] \Downarrow v_B \end{array}}{\texttt{let } x \texttt{ = } D \texttt{ in } B \Downarrow v_B} \quad (R_{let})$$

before substituting the definition D into the body expression B, we *first* evaluate D to a value.

There are disadvantages of this eager evaluation approach. For instance, if the argument value is not used in the body of the function, the computation to generate the value will still be carried out, an entirely wasted effort. An extreme case occurs when the computation of the argument value doesn't even terminate:

```
# let rec forever n = 1 + forever n ;;
val forever : 'a -> int = <fun>
# (fun x -> "this value ignores x") (forever 42) ;;
Line 1, characters 5-6:
1 | (fun x -> "this value ignores x") (forever 42) ;;
         ^
Warning 27: unused variable x.
Stack overflow during evaluation (looping recursion?).
```

If we had delayed the computation of `forever 42` until after it had been substituted in as the argument of the function, we would never have had to evaluate it at all, and the evaluation of the full expression would have terminated with `"this value ignores x"`.

Examples like this indicate the potential utility of LAZY EVALUATION – being able to DELAY computation until such time as it is needed, at which time the computation can be FORCED to occur.

There are, in fact, constructs of OCaml that work lazily. The conditional expression `if` $\langle expr_{test} \rangle$ `then` $\langle expr_{true} \rangle$ `else` $\langle expr_{false} \rangle$ delays evaluation of $\langle expr_{true} \rangle$ and $\langle expr_{false} \rangle$ until after evaluating $\langle expr_{test} \rangle$, and in fact will refrain from evaluating the unchosen branch of the conditional entirely. Thus the following computation terminates, even though the `else` branch, if it were evaluated, would not.

```
# if true then 3 else forever 42 ;;
- : int = 3
```

Another construct that delays computation is *the function itself*. The body of a function is not evaluated until the function is applied. If application is postponed indefinitely, the body is never evaluated. Thus the following "computation" terminates.

```
# fun () -> forever 42 ;;
- : unit -> int = <fun>
```

This latter approach provides a universal method for delaying and forcing computations: wrapping the computation in a function (delay), applying the function (forcing) if and when we need the value. What should the argument to the function be? Its only role is to postpone evaluation, so there needn't be a real datum as argument – just a `unit`. As noted above, we refer to this wrapping a computation in a function from unit as *delay* of the computation. Conversely, we *force* the computation when the delayed expression is applied to unit so as to carry out the computation and get the value.

Though OCaml is eager in its evaluation strategy (with the few exceptions noted), some languages have embraced lazy evaluation as the default, starting with Rod Burstall's Hope language and finding its widest use in the Haskell language named after Haskell Curry (Figure 6.2).

We'll make use of lazy evaluation in perhaps the most counter-intuitive application, the creation and manipulation of infinite data structures. We start with the *stream*, a kind of infinite list.

17.2 Streams

Here's a new algebraic data type definition for the STREAM.

```
# type 'a stream = Cons of 'a * 'a stream ;;
type 'a stream = Cons of 'a * 'a stream
```

It may look familiar; it shares much in common with the algebraic type definition of the polymorphic list, from Section 11.1, except that it dispenses with the `Nil` marking the end of the list.

We can define some operations on streams, like taking the head or tail of a stream.

```
# let head (Cons (hd, _tl) : 'a stream) : 'a = hd ;;
val head : 'a stream -> 'a = <fun>

# let tail (Cons (_hd, tl) : 'a stream) : 'a stream = tl ;;
val tail : 'a stream -> 'a stream = <fun>
```

It's all well and good to have streams and functions over them, but how are we to build one? It looks like we have a chicken and egg problem, requiring a stream in order to create one. Nonetheless, we press on, building a stream whose head is the integer 1. We start with

```
let ones = Cons (1, ...) ;;
```

We need to fill in the `...` with an `int stream`, but where are we to find one? How about the `int stream` named `ones` itself?

```
# let ones = Cons (1, ones) ;;
Line 1, characters 20-24:
1 | let ones = Cons (1, ones) ;;
                        ^^^^
Error: Unbound value ones
```

Of course, that doesn't work, because the name `ones` isn't itself available in the definition. That requires a `let rec`.

```
# let rec ones = Cons (1, ones) ;;
val ones : int stream = Cons (1, <cycle>)
```

It works! And the operations on this stream work as well:

```
# head ones ;;
- : int = 1
# head (tail ones) ;;
- : int = 1
# head (tail (tail ones)) ;;
- : int = 1
```

Its head is one, as is the head of its tail, and the head of the tail of the tail. It seems to be an infinite sequence of ones!

What is going on here? How does the implementation make this possible? Under the hood, the components of an algebraic data type have implicit pointers to their values. When we define `ones` as above, OCaml allocates space for the cons without initializing it (yet) and connects the name `ones` to it. It then initializes the contents of the cons, the head and tail, a pair of hidden pointers. The head pointer points to the value 1, and the tail points to the cons itself. This explains where the notation `<cycle>` comes from in the REPL printing out the value. In any case, the details of how this behavior is implemented isn't necessary to make good use of it.

Not all such cyclic definitions are well defined however. Consider this definition of an integer `x`:

```
# let rec x = 1 + x ;;
Line 1, characters 12-17:
1 | let rec x = 1 + x ;;
                 ^^^^^
Error: This kind of expression is not allowed as right-hand side of
  `let rec'
```

We can allocate space for the integer and name it `x`, but when it comes to initializing it, we need more than just a pointer to `x`; we need its value. But that isn't yet defined, so the whole process fails and we get an error message.

17.2.1 Operations on streams

We can look to lists for inspiration for operations on streams, operations like map, fold, and filter. Here is a definition for `map` on streams, which we call `smap`:

```
# let rec smap (f : 'a -> 'b) (s : 'a stream) : ('b stream) =
#   match s with
#   | Cons (hd, tl) -> Cons (f hd, smap f tl) ;;
val smap : ('a -> 'b) -> 'a stream -> 'b stream = <fun>
```

or, alternatively, using our recent definitions of `head` and `tail`,

```
# let rec smap (f : 'a -> 'b) (s : 'a stream) : ('b stream) =
#   Cons (f (head s), smap f (tail s)) ;;
val smap : ('a -> 'b) -> 'a stream -> 'b stream = <fun>
```

Now, we map the successor function over the stream of ones to form a stream of twos.

```
# let twos = smap succ ones ;;
Stack overflow during evaluation (looping recursion?).
```

Of course, that doesn't work at all. We're asking OCaml to add one to each element in an infinite sequence of ones. Luckily, `smap` isn't tail recursive, so we blow the stack, instead of just hanging in an infinite loop. This behavior makes streams as currently implemented less than useful since there's little we can do to them without getting into trouble. If only the system were less eager about doing all those infinite number of operations, doing them only if it "needed to".

The problem is that when calculating the result of the map, we need to generate (and cons together) both the head of the list (`f (head s)`) and the tail of the list (`smap f (tail s)`). But the tail already involves calling `smap`.

Why isn't this a problem in *calling* regular recursive functions, like `List.map`? In that case, there's a base case that is eventually called.

Why isn't this a problem in *defining* regular recursive functions? Why is there no problem in defining, say,

```
let rec fact n =
  if n = 0 then 1
  else n * fact (pred n) ;;
```

Recall that this definition is syntactic sugar for

```
let rec fact =
  fun n ->
    if n = 0 then 1
    else n * fact (pred n) ;;
```

The name `fact` can be associated with a function that uses it because *functions are values*. The parts inside are not further evaluated, at least not until the function is *called*. In essence, a function delays the latent computation in its body until it is applied to its argument.

We can take advantage of that in our definition of streams by using functions to perform computations lazily. We achieve laziness by wrapping the computation in a function delaying the computation until such time as we need the value. We can then force the value by applying the function.

To achieve the delay of computation, we'll take a stream not to be a cons as before, but a delayed cons, a function from `unit` to the cons. Other functions that need access to the components of the delayed cons can force it as needed. We need a new type definition for streams, which will make use of a simultaneously defined auxiliary type `stream_internal`:[1]

```
# type 'a stream_internal = Cons of 'a * 'a stream
#  and 'a stream = unit -> 'a stream_internal ;;
type 'a stream_internal = Cons of 'a * 'a stream
and 'a stream = unit -> 'a stream_internal
```

An infinite stream of ones is now defined as so:

[1] The `and` connective allows mutually recursive type definitions. Unfortunately, OCaml doesn't allow direct definition of nested types, like

```
type 'a stream = unit -> (Cons of 'a * 'a stream)
```

```
# let rec ones : int stream =
#   fun () -> Cons (1, ones) ;;
val ones : int stream = <fun>
```

Notice that it returns a delayed cons, that is, a function which, when applied to a unit, returns the cons.

We need to redefine the functions accordingly to take and return these new lazy streams. In particular, `head` and `tail` now force their argument by applying it to unit.

```
# let head (s : 'a stream) : 'a =
#   match s () with
#   | Cons (hd, _tl) -> hd ;;
val head : 'a stream -> 'a = <fun>

# let tail (s : 'a stream) : 'a stream =
#   match s () with
#   | Cons (_hd, tl) -> tl ;;
val tail : 'a stream -> 'a stream = <fun>

# let rec smap (f : 'a -> 'b) (s : 'a stream) : ('b stream) =
#   fun () -> Cons (f (head s), smap f (tail s)) ;;
val smap : ('a -> 'b) -> 'a stream -> 'b stream = <fun>
```

The `smap` function now returns a lazy stream, a function, so that the recursive call to `smap` isn't immediately evaluated (as it was in the old definition). Only when the cons is needed (as in the `head` or `tail` functions) is the function applied and the cons constructed. That cons itself has a stream as its tail, but that stream is also delayed.

Now, finally, we can map the successor function over the infinite stream of ones to form an infinite stream of twos.

```
# let twos = smap succ ones ;;
val twos : int stream = <fun>
# head twos ;;
- : int = 2
# head (tail twos) ;;
- : int = 2
# head (tail (tail twos)) ;;
- : int = 2
```

We can convert a stream – or at least the first `n` of its infinity of elements – into a corresponding list,

```
# let rec first (n : int) (s : 'a stream) : 'a list =
#   if n = 0 then []
#   else head s :: first (n - 1) (tail s) ;;
val first : int -> 'a stream -> 'a list = <fun>
```

allowing us to examine the first few elements of the streams we have constructed:

```
# first 10 ones ;;
- : int list = [1; 1; 1; 1; 1; 1; 1; 1; 1; 1]
```

```
# first 10 twos ;;
- : int list = [2; 2; 2; 2; 2; 2; 2; 2; 2; 2]
```

So far, we've constructed a few infinite streams, but none of much interest. But the tools are in hand to do much more. Think of the natural numbers: $0, 1, 2, 3, 4, 5, \ldots$. What is this sequence? We can think of it as the sequence formed by taking the natural numbers, incrementing them all to form the sequence $1, 2, 3, 4, 5, 6, \ldots$, and then prepending a zero to the front, as depicted in Figure 17.1.

We'll define a stream called `nats` in just this way.

```
# let rec nats =
#   fun () -> Cons (0, smap succ nats) ;;
val nats : int stream = <fun>
# first 10 nats ;;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
```

Let's just pause for a moment to let that sink in.

Start with the natural numbers
0 1 2 3 4 5 6 7 ...
Increment them
1 2 3 4 5 6 7 8 ...
Prepend a zero
0 1 2 3 4 5 6 7 8 ...

Figure 17.1: Creating an infinite stream of natural numbers by taking the natural numbers, incrementing them, and prepending a zero.

A function to map over two streams simultaneously, like the `List.map2` function, allows even more powerful ways of building streams.

```
# let rec smap2 f s1 s2 =
#   fun () -> Cons (f (head s1) (head s2),
#                   smap2 f (tail s1) (tail s2)) ;;
val smap2 : ('a -> 'b -> 'c) -> 'a stream -> 'b stream -> 'c stream
  = <fun>
```

We can, for instance, generate the Fibonacci sequence (see Exercise 33) in this way. Figure 17.2 gives the recipe.

Start with the Fibonacci sequence
0 1 1 2 3 5 8 ...
Take its tail
1 1 2 3 5 8 13 ...
Sum them
1 2 3 5 8 13 21 ...
Prepend a zero and one
0 1 1 2 3 5 8 13 21 ...

Figure 17.2: Creating an infinite stream of the Fibonacci numbers.

```
# let rec fibs =
#   fun () -> Cons (0,
#     fun () -> Cons (1,
#       (smap2 (+) fibs (tail fibs)))) ;;
val fibs : int stream = <fun>
```

Here, we've timed generating the first 10 elements of the sequence. It's slow, but it works.

```
# Absbook.call_reporting_time (first 10) fibs ;;
time (msecs): 2.447128
- : int list = [0; 1; 1; 2; 3; 5; 8; 13; 21; 34]
```

17.3 Lazy recomputation and thunks

Recall the definition of streams:

```
type 'a stream_internal = Cons of 'a * 'a stream
 and 'a stream = unit -> 'a stream_internal ;;
```

Every time we want access to the head or tail of the stream, we need to rerun the function. In a computation like the Fibonacci definition above, that means that every time we ask for the n-th Fibonacci number, we recalculate all the previous ones – more than once. But if the value being forced is pure, without side effects, there's no reason to recompute it. We should be able to avoid the recomputation by *remembering* its value the first time it's computed, and using the remembered value from then on. The term of art for this technique is MEMOIZATION.[2]

[2] Not "memorization". For unknown reasons, computer scientists have settled on this bastardized form of the word.

We'll encapsulate this idea in a new abstraction called a THUNK, essentially a delayed computation that stores its value upon being forced. We implement a thunk as a mutable value (a reference) that can be in one of two states: not yet evaluated or previously evaluated. The type definition is thus structured with two alternatives.

```
# type 'a thunk = 'a thunk_internal ref
#  and 'a thunk_internal =
#    | Unevaluated of (unit -> 'a)
#    | Evaluated of 'a ;;
type 'a thunk = 'a thunk_internal ref
and 'a thunk_internal = Unevaluated of (unit -> 'a) | Evaluated of
  'a
```

Notice that in the unevaluated state, the thunk stores a delayed value of type `'a`. Once evaluated, it stores an immediate value of type `'a`.

When we need to access the actual value encapsulated in a thunk, we'll use the force function. If the thunk has been forced before and thus evaluated, we simply retrieve the value. Otherwise, we compute the value, remember it by changing the state of the thunk to be evaluated, and return the value.

```
# let rec force (t : 'a thunk) : 'a =
#   match !t with
#   | Evaluated v -> v
#   | Unevaluated f ->
#       t := Evaluated (f ());
#       force t ;;
val force : 'a thunk -> 'a = <fun>
```

Here's a thunk for a computation of, say, factorial of 15. To make the timing clearer, we'll give it a side effect of printing a short message.

```
# let fact15 =
#   ref (Unevaluated (fun () ->
#                       print_endline "evaluating 15!";
#                       fact 15)) ;;
val fact15 : int thunk_internal ref = {contents = Unevaluated
  <fun>}
```

which can be forced to carry out the calculation:

```
# Absbook.call_reporting_time force fact15 ;;
evaluating 15!
time (msecs): 0.021935
- : int = 1307674368000
```

Now that the value has been forced, it is remembered in the thunk and can be returned without recomputation. You can tell that no recomputation occurs because the printing side effect doesn't happen, and the computation takes orders of magnitude less time.

```
# fact15 ;;
- : int thunk_internal ref = {contents = Evaluated 1307674368000}
# Absbook.call_reporting_time force fact15 ;;
time (msecs): 0.000954
- : int = 1307674368000
```

17.3.1 The Lazy Module

Thunks give us the ability to delay computation, force a delayed computation, and memoize the result. But the notation is horribly cumbersome. Fortunately, OCaml provides a module and some appropriate syntactic sugar for working with lazy computation implemented through thunks – the `Lazy` module.

In the built-in `Lazy` module, the type of a delayed computation of an `'a` value is given not by `'a thunk` but by `'a Lazy.t`. A delayed computation is specified not by wrapping the expression in `ref (Unevaluated (fun () -> ...))` but by preceding it with the new keyword `lazy`. Finally, forcing a delayed value uses the function `Lazy.force`.

Availing ourselves of the `Lazy` module, we can perform the same experiment more simply:

```
# let fact15 =
#   lazy (print_endline "evaluating 15!";
#         fact 15) ;;
val fact15 : int lazy_t = <lazy>
# Lazy.force fact15 ;;
evaluating 15!
- : int = 1307674368000
# Lazy.force fact15 ;;
- : int = 1307674368000
```

Now we can reconstruct infinite streams using the `Lazy` module. First, the stream type:

```
# type 'a stream_internal = Cons of 'a * 'a stream
#  and 'a stream = 'a stream_internal Lazy.t ;;
type 'a stream_internal = Cons of 'a * 'a stream
and 'a stream = 'a stream_internal Lazy.t
```

Functions on streams will need to force the stream values. Here, for instance, is the `head` function:

```
let head (s : 'a stream) : 'a =
  match Lazy.force s with
  | Cons (hd, _tl) -> hd ;;
```

Exercise 173

Rewrite `tail`, `smap`, `smap2`, and `first` to use the `Lazy` module.

The Fibonacci sequence can now be reconstructed. It runs hundreds of times faster than the non-memoized version in Section 17.2.1:

```
# let rec fibs =
#   lazy (Cons (0,
#     lazy (Cons (1,
#       smap2 (+) fibs (tail fibs))))) ;;
val fibs : int stream = <lazy>
# Absbook.call_reporting_time (first 10) fibs ;;
time (msecs): 0.005960
- : int list = [0; 1; 1; 2; 3; 5; 8; 13; 21; 34]
```

17.4 Application: Approximating π

A nice application of infinite streams is in the numerical approximation of the value of π. In 1715, the English mathematician Brook Taylor showed how to approximate functions as an infinite sum of terms, a technique we now call TAYLOR SERIES. For instance, the trigonometric arctangent function can be approximated by the following infinite sum:

$$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots$$

As a special case, the arctangent of 1 is $\frac{\pi}{4}$ (Figure 17.4). So

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$$

and

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \cdots \quad .$$

We can thus approximate π by adding up the terms in this infinite stream of numbers.

We start with a function to convert a stream of integers to a stream of floats.

```
# let to_float = smap float_of_int ;;
val to_float : int stream -> float stream = <fun>
```

Next, we build a stream of odd integers to serve as the denominators in all the terms in the Taylor series:

```
# let odds = smap (fun x -> x * 2 + 1) nats ;;
val odds : int stream = <lazy>
```

Figure 17.3: English mathematician Brook Taylor (1685–1731), inventor of the Taylor series approximation of functions.

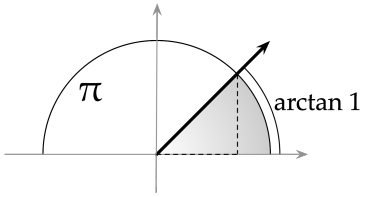

Figure 17.4: The arctangent of 1, that is, the angle whose ratio of opposite to adjacent side is 1, is a 45 degree angle, or $\frac{\pi}{4}$ in radians.

and a stream of alternating positive and negative ones to represent the alternate adding and subtracting:

```
# let alt_signs =
#   smap (fun x -> if x mod 2 = 0 then 1 else -1) nats ;;
val alt_signs : int stream = <lazy>
```

Finally, the stream of terms in the π sequence is

```
# let pi_stream = smap2 ( /. )
#                   (to_float (smap (( * ) 4) alt_signs))
#                   (to_float odds) ;;
val pi_stream : float stream = <lazy>
```

A check of the first few elements in these streams verifies them:

```
# first 5 odds ;;
- : int list = [1; 3; 5; 7; 9]
# first 5 alt_signs ;;
- : int list = [1; -1; 1; -1; 1]
# first 5 pi_stream ;;
- : float list =
[4.; -1.3333333333333326; 0.8; -0.571428571428571397;
  0.44444444444444442]
```

Now that we have an infinite stream of terms, we can approximate π by taking the sum of the first few elements of the stream, a PARTIAL SUM. The function `pi_approx` extracts the first n elements of the stream and sums them up using a fold.

```
# let pi_approx n =
#   List.fold_left ( +. ) 0.0 (first n pi_stream) ;;
val pi_approx : int -> float = <fun>
# pi_approx 10 ;;
- : float = 3.04183961892940324
# pi_approx 100 ;;
- : float = 3.13159290355855369
# pi_approx 1000 ;;
- : float = 3.14059265383979413
# pi_approx 10000 ;;
- : float = 3.14149265359003449
# pi_approx 100000 ;;
- : float = 3.14158265358971978
```

After 100,000 terms, we have a pretty good approximation of π, good to about four decimal places.

Of course, this technique of partial sums isn't in the spirit of infinite streams. Better would be to generate an infinite stream of all of the partial sums. Figure 17.5 gives a recipe for generating a stream of partial sums from a given stream. Starting with the stream, we take its partial sums (!) and prepend a zero. Adding the original stream and the prepended partial sums stream yields…the partial sums stream. This technique, implemented as a function over streams, is:

The given sequence

```
       1 2  3  4  5  6  7
...
```

…and its partial sums

```
       1 3  6 10 15 21 28
...
```

Prepend a zero to the partial sums

```
     0 1 3  6 10 15 21 28
...
```

…plus the original sequence

```
     1 2 3  4  5  6  7  8
...
```

…yields the partial sums

```
     1 3 6 10 15 21 28 36
...
```

Figure 17.5: Creating an infinite stream of partial sums of a given stream, in this case, the stream of positive integers. We prepend a zero to the sequence's partial sums and add in the original sequence to generate the sequence of partial sums. Only by virtue of lazy computation can this possibly work.

```
# let rec sums s =
#   smap2 ( +. ) s (lazy (Cons (0.0, sums s))) ;;
val sums : float stream -> float stream = <fun>
```

Now the first few approximations of π are easily accessed:

```
# let pi_approximations = sums pi_stream ;;
val pi_approximations : float stream = <lazy>
# first 5 pi_approximations ;;
- : float list =
[4.; 2.66666666666666696; 3.46666666666666679; 2.89523809523809561;
 3.33968253968254025]
```

If we want to find an approximation within a certain tolerance, say ϵ, we can search for two terms in the stream of approximations whose difference is less than ϵ.

```
# let rec within epsilon s =
#   let hd, tl = head s, tail s in
#   if abs_float (hd -. (head tl)) < epsilon then hd
#   else within epsilon tl ;;
val within : float -> float stream -> float = <fun>
```

We can now search for a value accurate to within any number of digits we desire:

```
# within 0.01 pi_approximations ;;
- : float = 3.13659268483881615
# within 0.001 pi_approximations ;;
- : float = 3.14109265362104129
```

Continuing on in this vein, we might explore methods for SERIES ACCELERATION – techniques to cause series to converge more quickly – or apply infinite streams to other applications. (In fact, series acceleration is the subject of Section **??**.) But for now, this should be sufficient to give a sense of the power of computing with infinite streams.

Exercise 174

As mentioned in Exercise 33, the ratios of successive numbers in the Fibonacci sequence approach the golden ratio (1.61803…). Show this by generating a stream of ratios of successive Fibonacci numbers and use it to calculate the golden ratio within 0.000001.

17.5 Problem section: Circuits and boolean streams

A boolean circuit is a device with one or more inputs and a single output that receives over time a sequence of boolean values on its inputs and converts them to a corresponding sequence of boolean values on its output. The building blocks of circuits are called *gates*. For instance, the *and* gate is a boolean device with two inputs; its output is `true` when its two inputs are both `true`, and `false` if either output is false. The *not* gate is a boolean device with a single input; its output is `true` when its input is `false` and vice versa.

In this problem, you'll generate code for modeling boolean circuits. The inputs and outputs will be modeled as *lazy boolean streams*.

Let's start with an infinite stream of `false` values.

Exercise 175

Define a value `falses` to be an infinite stream of the boolean value `false`.

Exercise 176

What is the type of `falses`?

Exercise 177

A useful function is the `trueat` function. The expression `trueat n` generates a stream of values that are all `false` except for a single `true` at index `n`:

```
# first 5 (trueat 1) ;;
- : bool list = [false; true; false; false; false]
```

Define the function `trueat`.

Exercise 178

Define a function `circnot : bool stream -> bool stream` to represent the *not* gate. It should have the following behavior:

```
# first 5 (circnot (trueat 1)) ;;
- : bool list = [true; false; true; true; true]
```

Exercise 179

Define a function `circand` to represent the *and* gate. It should have the following behavior:

```
# first 5 (circand (circnot (trueat 1)) (circnot (trueat 3))) ;;
- : bool list = [true; false; true; false; true]
```

A *nand* gate is a gate that computes the negation of an *and* gate. That is, it negates the *and* of its two inputs, so that its output is `false` only if *both* of its inputs are `true`.

Exercise 180

Succinctly define a function *circnand* using the functions above to represent the *nand* gate. It should have the following behavior:

```
# first 5 (circnand falses (trueat 3)) ;;
- : bool list = [true; true; true; true; true]
# first 5 (circnand (trueat 3) (trueat 3)) ;;
- : bool list = [true; true; true; false; true]
```

17.6 A unit testing framework

With the additional tools of algebraic data types and lazy evaluation, we can put together a more elegant framework for unit testing. Lazy evaluation in particular is useful here, since a unit test is nothing other than an expression to be evaluated for its truth at some later time when the tests are run. Algebraic data types are useful in a couple of ways, first to package together the components of a test and second to express the alternative ways that a test can come out.

Of course, tests can pass or fail, which we represent by an expression that returns either `true` or `false` respectively. But tests can have

other outcomes as well; there are other forms of failing than returning `false`. In particular, a test might raise an exception, or it might not terminate at all. In order to deal with tests that might not terminate, we'll need a way of safely running these tests in a context in which we cut off computation after a specified amount of time. The computation will be said to have TIMED OUT. To record the outcome of a test, we'll define a variant type:

```
# type status =
#   | Passed
#   | Failed
#   | Raised_exn of string  (* string describing exn *)
#   | Timed_out of int      (* timeout in seconds *) ;;
type status = Passed | Failed | Raised_exn of string | Timed_out of
  int
```

A unit test type will package together such a delayed expression, the test condition itself, a mnemonic label for the test, and a timeout period in seconds.

```
# type test =
#   { label : string;
#     condition : bool Lazy.t;
#     time : int } ;;
type test = { label : string; condition : bool Lazy.t; time : int;
  }
```

Notice that the condition of the test is a lazy boolean, so that the condition will not be evaluated until the test is run.

To construct a test, we provide a function that packages together the components.[3]

[3] We make use of an optional argument for the time, which defaults to five seconds if not provided. For the interested, details of optional arguments are discussed here.

```
# (* test ?time label condition -- Returns a test with the
#    given label and condition, with optional timeout time
#    in seconds. *)
# let test ?(time=5) label condition =
#   {label; condition; time} ;;
val test : ?time:int -> string -> bool Lazy.t -> test = <fun>
```

The crux of the matter is the running of a test. Doing so generates a value of type `status`. The `run_test` function will be provided a function `continue` to be applied to the label of the test and its status. For instance, an appropriate such function might print out a line in a report describing the outcome, like this:

```
# (* present labels status -- Prints a line describing the
#    outcome of a test. Appropriate for use as the continue
#    function in run_test. *)
# let present (label : string) (status : status) : unit =
#   let open Printf in
#   match status with
#   | Passed ->
```

```
#       printf "%s: passed\n" label
#   | Failed ->
#       printf "%s: failed\n" label
#   | Timed_out secs ->
#       printf "%s: timed out after %d seconds\n" label secs
#   | Raised_exn msg ->
#       printf "%s: raised %s\n" label msg ;;
val present : string -> status -> unit = <fun>
```

The `run_test` function needs to evaluate the test by forcing evaluation of the delayed condition. As a first cut, we'll look only to the normal case, where a test returns `true` or `false`.

```
# (* run_test test continue -- Runs the test, applying the
#    continue function to the test label and status. *)
# let run_test ({label; condition; _} : test)
#              (continue : string -> status -> unit)
#            : unit =
#     let result = Lazy.force condition in
#     if result then continue label Passed
#     else continue label Failed ;;
val run_test : test -> (string -> status -> unit) -> unit = <fun>
```

But what if the test raises an exception? We'll evaluate the test condition in a `try` ⟨⟩ `with` ⟨⟩ to deal with this case.

```
# (* run_test test continue -- Runs the test, applying the
#    continue function to the test label and status. *)
# let run_test ({label; condition; _} : test)
#              (continue : string -> status -> unit)
#            : unit =
#     try
#       let result = Lazy.force condition in
#       if result then continue label Passed
#       else continue label Failed
#     with
#     | exn -> continue label
#                (Raised_exn (Printexc.to_string exn)) ;;
val run_test : test -> (string -> status -> unit) -> unit = <fun>
```

Finally, we need to deal with timeouts. We appeal to a function `timeout` that forces a lazy computation, but raises a special `Timeout` exception if the computation goes on too long. The workings of this function are well beyond the scope of this text, but we provide the code in Figure 17.6.

Using the `timeout` function to force the condition and checking for the `Timeout` exception handles the final possible status of a unit test.

```
# (* run_test test continue -- Runs the test, applying the
#    continue function to the test label and status. *)
# let run_test ({label; time; condition} : test)
#              (continue : string -> status -> unit)
#     : unit =
```

```
# (* timeout time f -- Forces delayed computation f, returning
#    what f returns, except that after time seconds it raises
#    a Timeout exception. *)
#
# exception Timeout ;;
exception Timeout

# let sigalrm_handler =
#   Sys.Signal_handle (fun _ -> raise Timeout) ;;
val sigalrm_handler : Sys.signal_behavior = Sys.Signal_handle <fun>

# let timeout (time : int) (f : 'a Lazy.t) : 'a =
#   let old_behavior =
#     Sys.signal Sys.sigalrm sigalrm_handler in
#   let reset_sigalrm () =
#     ignore (Unix.alarm 0);
#     Sys.set_signal Sys.sigalrm old_behavior in
#   ignore (Unix.alarm time) ;
#   let res = Lazy.force f in
#   reset_sigalrm () ; res ;;
val timeout : int -> 'a Lazy.t -> 'a = <fun>
```

Figure 17.6: The function `timeout` used in the evaluation of unit tests, based on the `timeout` function of Chailloux et al. (2000)

```
#   try
#     if timeout time condition
#     then continue label Passed
#     else continue label Failed
#   with
#   | Timeout -> continue label (Timed_out time)
#   | exn     -> continue label
#                  (Raised_exn (Printexc.to_string exn)) ;;
val run_test : test -> (string -> status -> unit) -> unit = <fun>
```

By iterating over a list of unit tests, we can generate a nice report of all the tests.

```
# (* report tests -- Generates a report based on the
#    provided tests. *)
# let report (tests : test list) : unit =
#   List.iter (fun test -> run_test test present) tests ;;
val report : test list -> unit = <fun>
```

With this infrastructure in place, we can define a test suite that demonstrates all of the functionality of the unit testing framework.

```
# let tests =
#   [ test "should fail" (lazy (3 > 4));
#     test "should pass" (lazy (4 > 3));
#     test "should time out" (lazy (let rec f x = f x in f 1));
#     test "should raise exception" (lazy ((List.nth [0; 1] 3) = 3))
#   ] ;;
val tests : test list =
  [{label = "should fail"; condition = <lazy>; time = 5};
   {label = "should pass"; condition = <lazy>; time = 5};
   {label = "should time out"; condition = <lazy>; time = 5};
   {label = "should raise exception"; condition = <lazy>; time =
  5}]

# report tests ;;
should fail: failed
should pass: passed
should time out: timed out after 5 seconds
should raise exception: raised Failure("nth")
- : unit = ()
```

Figure 17.7: Peter Landin (1930–2009), developer of many innovative ideas in programming languages, including the roots of lazy programming. His influence transcended his role as a computer scientist, especially in his active support of gay rights.

17.7 A brief history of laziness

The idea of lazy computation probably starts with Peter Landin (Figure 17.7). He observed "a relationship between lists and functions":

> In this relationship a nonnull list *L* is mirrored by a none-adic function *S* that produces a 2-list consisting of (1) the head of *L*, and (2) the function mirroring the tail of *L*. . . . This correspondence serves two related purposes. It enables us to perform operations on lists (such as generating them, mapping them, concatenating them) without using an "extensive," item-by-item representation of the intermediately resulting lists;

> and it enables us to postpone the evaluation of the expressions specifying the items of a list until they arc actually needed. The second of these is what interests us here. (1965)

The idea of a "function mirroring the tail of" a list is exactly the delaying of the tail computation that we've seen in the `stream` data type.

Landin is notable for many other ideas of great currency. For instance, he invented the term "syntactic sugar" for the addition of extra concrete syntax to abbreviate some useful but otherwise complicated abstract syntax. His 1966 paper "The next 700 programming languages" (Landin, 1966) introduced several innovative ideas including the "offside rule" of concrete syntax, allowing the indentation pattern of a program to indicate its structure. Python is typically noted for making use of this Landin innovation. Indeed, the ISWIM language that Landin described in this paper is arguably the most influential programming language that no one ever programmed in.

Following Landin's observation, Wadsworth proposed the lazy lambda calculus in 1971, and Friedman and Wise published an article proposing that "Cons should not evaluate its arguments" in 1976. The first programming language to specify lazy evaluation as the evaluation regime was Burstall's Hope language (which also introduced the idea, found in nascent form in ISWIM, of algebraic data types). A series of lazy languages followed, most notably Miranda, but the lazy programming community came together to converge on the now canonical lazy language Haskell, named after Haskell Curry.

17.8 Supplementary material

- Lab 14: Lazy programming and infinite data structures: Implementing laziness as user code
- Lab 15: Lazy programming and infinite data structures: Using OCaml's native lazy module
- Problem set A.7: Refs, streams, and music