Streams

COMP 320

Call By Value

SML, like many languages uses call by value in evaluating expressions

In evaluating expressions, we first evaluate e1 to some value v1 and then substitute the v1 for x in the body

```
(fn x => e) e1 -> [v1/x]e

let x = e1 in e -> [v1/x]e
```

This happens whether or not we need the value (i.e. whether or not there are any occurrences of x in e)

An Extreme Example

```
fun horriblecomputation(x:int):int =
    let fun ackermann(0:int, n:int):int = n+1
          | ackermann(m, 0) = ackermann(m-1, 1)
          | ackermann(m, n) = ackermann(m-1, ackermann(m, n-1))
        val y = Int.abs(x) mod 3 + 2
        fun count(0) = ackermann(y, 4)
          | count(n) = count(n-1)+0*ackermann(y,4)
        val large = 1000
    in
        ackermann(y, 1)*ackermann(y, 2)*ackermann(y, 3)*count(large)
    end;
```

Horrible example

Using call by value (eager evaluation) the argument is evaluated and bound even if the variable is not needed:

```
let
  val x = horriblecomputation (234)
in
  5
end
```

Call by Name

Some languages (Haskell) use a call by name (lazy evaluation) approach

Bind the variable to an unevaluated expression and only evaluate it when evaluating when needed

```
(fn x => e) e1 -> [e1/x]e

let x = e1 in e -> [e1/x]e
```

In the previous example the horriblecomputation would never be evaluated

The evaluation is **suspended** and only **forced** when needed

Memoization

Using call by name (lazy evaluation) the argument may be evaluated many times:

```
let
  val x = horriblecomputation (234)
in
  x + x
end
```

Languages that implement lazy evaluation use call-by-need or memoization

When the unevaluated expression is evaluated for the first time, the value is memoized x is bound to that value

Benefits of Lazy Evaluation

Supports demand-driven computation

- We only compute a value if it is needed at some point
- Useful when dealing with online data structures

Supports infinite data structures

- Representing all prime numbers cannot be done eagerly
- With lazy evaluation we create as many as we need

Supports interactive data structures

- For example a sequence of inputs
- User inputs are created on demand as a response to the progress of computation

Methodology

To prevent the eager evaluation of an expression we can embed it in a function

We never evaluate inside the body of a function

For example, to suspend the evaluation of 4*7 we can wrap it as `fn () => 4 * 7`

This is a function of type `unit -> int`

We can generalize this to suspend the evaluation of expressions of any type

Suspending Computation

We can generalize and define a datatype with a constructor that tags such suspended computations:

```
datatype 'a susp = Susp of (unit -> 'a)
```

To delay a computation we can wrap it in a suspension

```
delay: (unit -> 'a) -> 'a susp

fun delay c = Susp c
```

To force the computation of the value held in the suspension

```
fun force (Susp c) = c ()
```

Back to our horrible computations

```
let
  val x = horriblecomputation (234)
in
  x + x
end
```

Can become

```
let
  val x = Susp (fn () => horriblecomputation (234))
in
  force x + force x
end
```

Defining Infinite Structures

We can use the concept of delaying evaluation to define infinite structures

Given a stream of natural numbers such as 1 2 3 4 …, we can only make observations and examine parts of the stream.

Programs that manipulate streams do not terminate but we can make observations at each step and produce results. This is unlike a program with an infinite loop which is not productive.

A Direct Definition

```
datatype 'a stream = STREAM of unit -> ('a * 'a stream)

fun force (STREAM s) = s()

fun head s = let val (x,_) = force(s) in x end

fun tail s = let val (_,x) = force(s) in x end

fun consstream(x,s) = STREAM(fn () => (x,s))
```

Natural Numbers

```
fun nums_from n = STREAM(fn () => (n, nums_from (n+1)))


val naturals = nums_from 0
```

Functions on Streams

```
fun take n s = if n=0 then [] else
                  let val (x,s1) = force s
                  in
                     x :: (take (n-1) s1)
                  end

fun mapstream f s = STREAM(fn () =>
                        let val (x,xs) = force(s) in
                             (f x, mapstream f xs) end)
```

Stream of Primes

We can build a stream of prime numbers using the Sieve of Eratosthenes method

```
fun filter p s = STREAM(fn () =>
                        let val(x,s') = force s
                        in
                            if (x mod p) = 0
                            then force(filter p s')
                            else (x, filter p s')
                        end)
```

Stream of Primes

```
fun sieve s = STREAM(fn () =>
                    let val (x,s') = force s in
                        (x, sieve(filter x s'))
                    end)

val primes = sieve(nums_from 2)
```

SML/NJ Lazy Types

SML/NJ supports lazy types directly and allows us to attach the keyword lazy to a datatype declaration

This feature must be enabled by executing the following at the top level

```
Compiler.Control.Lazy.enabled := true;

open Lazy;
```

Then we can define

```
datatype lazy 'a stream = Cons of 'a * 'a stream;
```

There is no base case!

Lazy specifies the values of ‘a stream are computations of the form Cons (v1, v2) where v1 is of type ‘a and v2 is another suspended computation of the same type

Creating values

```
val rec lazy ones = Cons (1, ones);
```

This defines an infinite stream of 1's of type `int stream`

Lazy means we are bind ones to a computation, not a value

Rec means that the computation is recursive. It is constructed using Cons from the integer 1 and the same computation

We can use pattern matching to inspect the values:

```
val Cons (h, t) = ones
```

Extracts the "head" and "tail' if the stream, binding h to 1 and t to ones

Pattern matching forces evaluation of the computation to the extent required by the pattern.

```
val Cons (h, Cons (h', t')) = ones
```

Head and Tail for Streams

We can define functions over lazy data structures:

```
fun shd (Cons (x, _)) = x;
fun stl (Cons (_, s)) = s;
```

For shd we extract a value of type ‘a from a value of type ‘a stream which has the form

Cons(e1, e2)

Stream Map Function

When applying smap to a stream, it should set up another stream

When the new stream is forced to obtain the head element, the function should be applied to it

```
(*smap : ('a -> 'b) -> 'a stream -> 'b stream *)
fun smap f =
    let
      fun lazy loop (Cons (x, s)) = Cons (f x, loop s)
    in
      loop
    end;
```

This is a staged computation. The partial application to a function produces a function that loops over the given stream, applying the function to each element

Leaving out “lazy” would force the computation of the head rather than jus set up a future computation

Natural Numbers

Applying smap allows us to define the infinite stream of natural numbers

```
fun succ n = n+1;
val one_plus = smap succ;
val rec lazy nats = Cons (0, one_plus nats);
```

Stream Filter

We can filter out all elements of a stream that do not satisfy a given predicate

```
sfilter : ('a -> bool) -> 'a stream -> 'a stream)


fun sfilter pred =
    let
     fun lazy loop (Cons (x, s)) =
         if pred x then
            Cons (x, loop s)
         else
            loop s
    in
     loop
    end;
```

Sieve

Using sfilter we can define a sieve that takes a stream of numbers and retains only the numbers not divisible by a preceding number in the stream

```
fun m mod n = m - n * (m div n);

fun divides m n = n mod m = 0;

fun lazy sieve (Cons (m, s)) =
        Cons (m, sieve (sfilter (not o (divides m)) s));
```

Applying sieve to the natural numbers >= 2 gives the infinite stream of primes

```
val nats2 = stl (stl nats);

val primes = sieve nats2;
```

Adding Two Streams

```
fun add (Cons (x, xs), Cons(y,ys)) =
          Cons (x+y, fn() => add (force xs, force ys))
```

Fibonacci Sequence

```
val FibStream ;
  let
    fun fib a b = Cons(a, (fn () => fib b (a+b)))
  in
    fib 0 1
  end
```

Inspecting Values in a Stream

To inspect the values of a stream, it is often useful to build a list with the first n elements of the stream

```
fun take 0 s = nil
  | take n (Cons (x, s)) = x :: take (n-1) s;
```