Streams

November 1, 2001

CS 131: Programming Languages

Question

- Suppose the set S satisfies two conditions:
  - The value `nil` is a member of S.
  - Whenever n is an integer and $l \in S$ we have $(n::l) \in S$.
- Does this uniquely define the set of all integer lists?

Inductive Types

- Lists (and trees, and NQSML programs, and most of the examples of datatype we have seen) are said to be *inductive* types.
  - Inductively defined
  - Guaranteed to be finite
    - Traversal of any such data structure will terminate.
  - Cannot write `val rec ones = 1 :: ones`
- How could we get infinite lists in SML?

Cyclic Lists

- One way to build infinite lists is to make a list containing a loop.
  - The only way to make cyclic structures in SML is to use `ref` and assignment.

```
datatype cyclist = NIL
                 | CONS of int * (cyclist ref)

fun print_cyclist NIL          = ()
  | print_cyclist (CONS(n,r)) = (print (Int.toString n);
                                  print " ";
                                  print_cyclist (!r))
```

Cyclic Lists

```
val r = ref NIL        : cyclist ref
val z = CONS(3, r)     : cyclist
val y = CONS(2, ref z) : cyclist
val x = CONS(1, ref y) : cyclist
r := x;

print_cyclist x
```

Criticism

- Such cyclic lists don't always suffice
  - Yields infinite lists, but only those whose contents eventually start repeating.
- Lots of interesting infinite sequences we still can't represent

Streams

- How can we get "real" infinite lists?
  - Answer: *streams*
- A stream is a lazily computed list
  - Elements computed "on demand"
  - Introduce a new type, `'a stream`.
    - Depending on implementation, streams may or may not be required to be infinite. We will permit finite streams.

Applications of Streams

- Infinite sequences:
  - Interesting sequences of integers
    - The integers n and greater
    - The sequence of all prime numbers
  - Representing real numbers
    - As a stream of the digits in the decimal expansion
    - As a stream of increasingly precise approximations
  - Time series data
    - Bit output of a clocked digital circuit

Applications of Streams

- Finite sequences
  - Expressing solutions to questions with multiple answers, not all of which may be needed
    - E.g., find all occurrences of `"giraffe"` in the string `s`.
  - Comparison of two separately-computed finite sequences
    - Can stop computing the sequences as soon as a mismatch is found.
  - Generate-and-test
    - If you're only looking for the first element of a sequence satisfying a given property, why bother computing the entire sequence?

Applications of Streams

- Input/Output
  - User input can be viewed as a stream of actions
    - Keystrokes
    - Mouse clicks
  - Streams can be used to handle I/O in languages without side-effects (e.g., Haskell)
    - Whole program could be a big function mapping stream of program inputs to stream of program outputs.
    - Special case: stream transformers like UNIX pipes

Stream Interface (partial)

```
type 'a stream

val sempty : unit -> 'a stream
val snull  : 'a stream -> bool
exception Empty
val shd    : 'a stream -> 'a
val stl    : 'a stream -> 'a stream
val scons : 'a * 'a stream -> 'a stream

val take  : int -> 'a stream -> 'a list
val drop  : int -> 'a stream -> 'a stream

val delayed : (unit -> 'a stream) -> 'a stream
```

Stream Implementation

```
datatype 'a cell = Nil
                 | Cons of 'a * 'a stream
withtype 'a stream = 'a cell susp
```

```
fun sempty() = delay(fn () => Nil)

fun snull(str) =
  (case (force str) of
     Nil => true | _ => false)
```

Stream Implementation

```
datatype 'a cell = Nil
                 | Cons of 'a * 'a stream
withtype 'a stream = 'a cell susp
```

```
exception Empty
fun shd(s) = (case (force s) of
                Nil => raise Empty
              | Cons(h,_) => h)
fun stl(s) = (case (force s) of
                Nil => raise Empty
              | Cons(_,t) => t)

fun scons(h:'a, t:'a stream) =
  delay(fn () => Cons(h,t))
```

Stream Implementation

```
datatype 'a cell = Nil
                 | Cons of 'a * 'a stream
withtype 'a stream = 'a cell susp
```

```
fun take 0 s = []
  | take n s = (shd s) :: (take (n-1) (stl s))

fun drop 0 s = s
  | drop n s = drop (n-1) (stl s)

fun delayed (f : unit -> 'a stream) : 'a stream =
   delay (fn () => force (f()))
```

Programming with Streams

```
val onetwothree : int stream =
   scons(1,scons(2,scons(3,sempty())))

val ones : int stream =
   let
      fun t() = scons(1, delayed t)
   in
      delayed t
   end

val ten_ones = take 10 ones
val onehundred_ones = take 100 ones
```

```
- take 10 ones;
val it = [1,1,1,1,1,1,1,1,1,1] : int list
```

Programming with Streams

```
fun smap (f:'a->'b) (s:'a stream) : 'b stream =
  delayed
   (fn () => if (snull s) then
                sempty()
             else
                scons(f(shd s), smap f (stl s)))

fun sfilter (p:'a->bool) (s:'a stream) : 'a stream =
  delayed
   (fn () => if (snull s) then
                sempty()
             else if (p (shd s)) then
                scons(shd s, sfilter p (stl s))
             else
                sfilter p (stl s))
```

Programming with Streams

```
val succ n = n+1
val nats =
  let
     fun t() = scons(0, smap succ (delayed t))
  in
     delayed t
  end

fun upfrom n0 =
   let
      fun t n () = scons(n, delayed (t (n+1)))
   in
      delayed (t n0)
   end
val nats' = upfrom 0
```

Programming with Streams

```
fun isPrime (n:int) : bool = ...

val prime_stream = sfilter isPrime (upfrom 2)
```

Programming with Streams

```
fun divides m n = (n mod m = 0)

fun sieve (s:int stream) =
  delayed
   (fn () =>
     scons(shd s,
           sieve (sfilter (not o divides (shd s))
                          (stl s))))


val prime_stream = sieve (upfrom 2)
```

```
- take 30 prime_stream;
val it =
  [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,
   67,71,73,79,83,89,97,101,103,107,109,113] : int list
```

Even Lazier Functions

```
(* Delay even computing the head of the list.
 *)
fun slcons(h:'a susp, t:'a stream) =
      delayed (fn () => scons(force h, t))

(* Lazily computes the tail.  Note that if the input
   is empty, the exception is also delayed.
 *)
fun sltl(s : 'a stream) =
      delayed (fn () => stl(s))
```

Digression

- SML also handles I/O via streams, but uses *imperative* streams

```
structure TextIO :
  sig
    type instream  (* stream of characters *)

    val stdIn  : instream
    val openIn : string -> instream
    val input  : instream -> string
    ...
    val setPosIn : instream * StreamIO.in_pos -> unit
  end
```

Streams in Haskell

- In lazy functional languages like Haskell, all lists are potentially infinite.
  - Even cons is call-by-need (delays evaluating its arguments)
  - In both arguments!

```
ones = 1 : ones
nats = 1 : (map (\n -> n+1) nats)
w    = tail ((3/0) : [4])
```

Lazy Streams in SML/NJ

- It is perfectly reasonable to add lazy features to a "strict" or "eager" language like SML:
  - Research versions of SML/NJ (>= 110.5) have built-in support for lazy data structures

```
datatype lazy 'a stream = Nil
                        | Cons of 'a * 'a stream

val rec lazy ones = Cons(1,ones)

fun      shd  (Cons(h,_)) = h
fun      stl  (Cons(_,t)) = t
fun lazy sltl (Cons(_,t)) = t
```