CS 312

Lecture 27
29 April 2008

Lazy Evaluation,
Thunks, and Streams

Evaluation

- SML as you know it (substitution semantics)

  `if true then e`$_1$ `else e`$_2$ $\longrightarrow$ `e`$_1$

  `if false then e`$_1$ `else e`$_2$ $\longrightarrow$ `e`$_2$

- "if" *eagerly* evaluates condition expression to true or false, *lazily* evaluates e_1, e_2
- In general: subexpressions either eagerly or lazily evaluated
  - Function bodies: lazily evaluated

    `fn (x) => e` is a value

Factorial - right and wrong

```
fun factorial (n : int) : int =
  if n <= 0 then 1 else n*factorial(n-1)
```

When evaluating **`factorial 0`**,

when do we evaluate **`n*factorial(n-1)`**?

```
fun factorial2 (n : int) : int =
  my_if(n <= 0, 1, n*factorial(n-1))
```

When evaluating **`factorial2 0`**,

when do we evaluate **`n*factorial(n-1)`**?

Eager evaluation in ML

- Function arguments evaluated before the function is called (and values are passed)
- **`if`** condition evaluated after guard evaluated
- Function bodies not evaluated until function is applied.
- Need some laziness to make things work...

Laziness and redundancy

- Eager language (SML): *call by value*

  `let x = v in` $e_2 \longrightarrow e_2\{v/x\}$

  `(fn(x) =>` e_2`) (v)` $\longrightarrow e_2\{v/x\}$

  – Bound value is evaluated eagerly before body e_2

- Lazy language (Haskell): *call by name*

  `let x =` e_1 `in` $e_2 \longrightarrow e_2\{e_1/x\}$

  `(fn(x) =>` e_2`)` $(e_1) \longrightarrow e_2\{e_1/x\}$

  – e_1 is not evaluated until **x** is used
  – Variable can stand for unevaluated expression
  – But: what if **x** occurs 10 times in e_2 ?

A funny rule

- **`val f = e`** evaluates **e** once “right away”.
- **`val f = fn()=>e`** evaluates **e** every time but not until **f** is called.
- What if we had

  `val f = Thunk.make (fn()=> e)`

which evaluates **e** once, but not until we use **f**.

A general mechanism for lazy evaluation.

The Thunk ADT

```
signature THUNK = sig
    (* A `a thunk is a lazily
     * evaluated expression e of type
     * `a. *)
    type 'a thunk
    (* make(fn()=>e) creates a thunk
     * for e *)
    val make : (unit->'a) -> 'a thunk
    (* apply(t) is the value of its
     * expression, which is only evaluated
     * once. *)
    apply : 'a thunk -> 'a
  end
```

Lazy languages

- Implementation has to use a ref. (How else could **`Thunk.apply e`** act differently at different times?)
- Some languages have *special syntax* for lazy evaluation.
- Algol-60, Haskell, Miranda:

  `val x = e` acts like

  `val x = Thunk.make (fn()=> e)`
- We *implemented* lazy evaluation using refs and functions – lazy functional languages have this implementation baked in.

Streams

- A stream is an "infinite" list – you can ask for the rest of it as many times as you like and you'll never get null.
- Can pass a series of values between different modules with loose coupling, no side effects
- The universe is finite, so a stream must really just *act* like an infinite list.
- Idea: use a function to describe what comes next.

The Stream ADT

```
signature STREAM =
  sig
    (* An infinite sequence of `a *)
    type 'a stream
    (* make(b,f) is the infinite sequence
     * [b,f(b),f(f(b)), …] *)
    val make: ('a*('a->'a)) -> 'a stream
    (* next[x0,x1,x2,…] is (x0, [x1,x2,…]) *)
    val next: 'a stream -> ('a*'a stream)
  end
```

Example: infinite list of primes

State w/o destructive update

- We can model infinite sequences (of numbers, of circuit states, of whatever) without destroying old versions with refs.
- In fact, the stream is non-imperative! (if function is non-imperative)
- …

Implementing streams (wrong)

Intuitively:

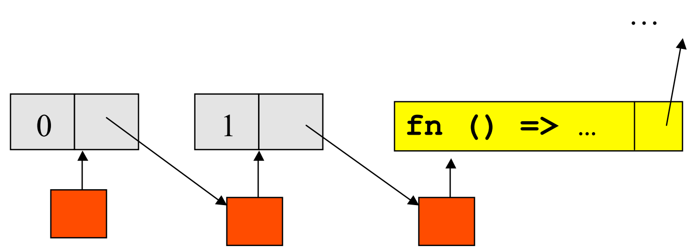
```
datatype 'a stream =
  Cons of ('a * 'a stream)
fun make (init:'a, f:'a -> 'a): 'a stream =
   Cons(init, make (f init, f))

fun next (Str(th):'a stream): 'a*'a stream =
   th
```

But what is **make** *going to do?*

The Punch Line

If only there were a way to delay the making of the rest of the stream until the previous items had been accessed…



(Implementation: **`stream.sml`**)

Streams via functions

```
structure Stream :> STREAM =
  struct
    datatype 'a stream =
      Cons of unit -> ('a * 'a stream)

    fun make (init : 'a, f : 'a -> 'a) : 'a stream =
      Cons(fn () => (init, make (f init, f)))

    fun next (Cons(F): 'a stream): 'a * 'a stream =
      F()
end
```

Streams via thunks

```
structure Stream :> STREAM =
  struct
    datatype 'a stream =
      Cons of ('a * 'a stream) Thunk.thunk

    fun make (init : 'a, f : 'a -> 'a) : 'a stream =
      Cons(Thunk.make(fn() =>
             (init, make (f init, f))))

    fun next (Cons(th): 'a stream): 'a * 'a stream =
      Thunk.apply th
  end
```

Advantage: stream values are computed at most once (and only if needed)

Summary

ADTs for lazy computation:

- Thunk – one lazy expression
- Stream – lazily computed infinite list

- Lazy language: can make recursive data structures, lists *are* streams
  `val lst = 1::lst`

- Try it out!