CS 312

1 May 2003

Lazy Evaluation,
Thunks, and Streams

Evaluation

- SML as you know it (substitution semantics)

  `if true then` e_1 `else` e_2 $\mapsto$ e_1

  `if false then` e_1 `else` e_2 $\mapsto$ e_2

- "if" *eagerly* evaluates condition expression to true or false, lazily evaluates e_1, e_2

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

- In general: subexpressions either eagerly or lazily evaluated
  - Function bodies: lazily evaluated

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)

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

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

  - Bound value is evaluated eagerly before body e_2

- Lazy language:

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

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

  - 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 = fn()=>e` evaluates e every time but not until f is called.
- `val f = e` evaluates e once "right away".
- 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

Lazy Evaluation

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

which evaluates e once, but not until we use f

- Best of both worlds: no redundant evaluations, no unnecessary evaluations
- But…harder to reason about when something happens (but maybe you don't care!)
- How to make sure we evaluate e at most once?

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.
- 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

That was cool…

- We could 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)
- State without the destructive updates…

Implementing streams (wrong)

Intuitively:

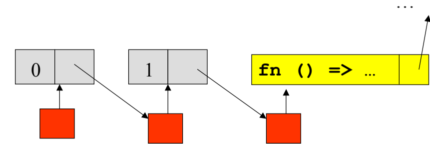
```
datatype 'a stream =
  Cons of ('a * 'a stream)
fun make (init:'a, f:'a -> 'a): 'a stream =
   Cons(init, make (f init, f))

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 – infinite sequence, lazily computed

- Lazy language: can make recursive data structures, streams are lists

  `val lst = 1::lst`

- Try it out!