Alternative Evaluation Orders: Delay and laziness

When are expressions evaluated?

Bonus: memoization

Delayed evaluation with thunks

effect emulation of lexically-scoped call-by-name semantics

Thunk $fn () \rightarrow e$

$\star n$, a zero-argument function used to delay evaluation
$\star w$ to create a thunk from an expression:
"thunk the expression"

No new language features.

fun ifok x y z =
   if x then y () else z ()

fun fact n =
   ifok (n = 0)
      (fn () \rightarrow 1)
   (fn () \rightarrow n * (fact (n - 1)))

Eager evaluation: evaluate arguments first

call-by-value semantics

When do arguments/subexpressions evaluate (ML, Racket, ...)?

- Function arguments: once, before calling function
- Conditional branches: only one branch, after checking condition

fun iffy x y z =
   if x then y else z

fun facty n =
   iffy (n = 0)
   1
   (n * (facty (n - 1)))

What's wrong?

Thunk: evaluate when value needed

effect emulation of lexically-scoped call-by-name semantics

fun f1 th =
   if ... then 7 else th() ...

fun f2 th =
   if ... then 7 else th() + th()

fun f3 th =
   let val v = th ()
   in if ... then 7 else v + v end

fun f4 th =
   if ... then 7 else
   let val v = th () in v + v end

See code examples
Lazy evaluation: evaluate first time value needed
  call-by-need semantics

Argument/subexpression evaluated zero or one times,
no earlier than first time result is actually needed.

Result reused (not recomputed) if needed again anywhere.

Benefits of delayed evaluation, with minimized costs.

Explicit laziness with promises:
  • Promise.delay (fn () => x * f x)
  • Promise.force p

Promises: explicit laziness
(a.k.a. suspensions)

signature PROMISE =
sig
  (* Type of promises for 'a. *)
  type 'a t

  (* Take a thunk for an 'a and
    make a promise to produce an 'a. *)
  val delay : (unit -> 'a) -> 'a t

  (* If promise not yet forced, call thunk and save.
     Return saved thunk result. *)
  val force : 'a t -> 'a

end

Limited mutation hidden in ADT.

Streams: infinite sequence of values

• Cannot make all the elements now.
• Make one when asked, delay making the rest with a thunk.

• Interface/idiom for division of labor:
  • Stream producer
  • Stream consumer
  • Interleave: production and consumption in time, but not in code.

• Examples:
  • UI events
  • UNIX pipes: bg diff delay.sml | grep "thunk"
  • Sequential logic circuit updates (CS 240)
Streams in ML: false start

Let a stream be a thunk that, when called, returns a pair of
• the next element; and
• the rest of the stream.

```plaintext
fn () => (next_element, nextThunk)
```

Given stream s, get elements:
• First:  `let val (v1,s1) = s ()`
• Second: `val (v2,s2) = s1 ()`
• Third:  `val (v3,s3) = s2 ()` ...

Type of s? s1? s2? s3? ...

---

Streams in ML: recursive types

Single-constructor datatype allows recursive type:

```plaintext
datatype 'a Scons = Scons of 'a * (unit -> 'a Scons)
type 'a stream = unit -> 'a Scons
```

Given a stream s:
• First:  `let val Scons(v1,s1) = s ()`
• Second: `val Scons(v2,s2) = s1 ()`
• Third:  `val Scons(v3,s3) = s2 ()` ...

Type of s? s1? s2? s3? ...

---

Stream consumers

Find index of first element in stream for which f returns true.

```plaintext
fun numuntil f stream =
  let fun consume stream acc =
    let val Scons(v,s) = stream ()
    in
      if f v
      then acc
      else consume s (acc + 1)
    end
    in consume stream 0 end
  : ('a -> bool) -> 'a stream -> int
```

---

Stream producers

```plaintext
fun ones () = Scons (1,ones)
val rec ones = fn () => Scons (1,ones)
```

Create next thunk via delayed recursion!
• Return a thunk that, when called, calls the outer function recursively.

```plaintext
val nats =
  let fun f x = Scons (x, fn () => f (x + 1))
  in fn () => f 0 end
val powers2 =
  let fun f x = Scons (x, fn () => f (x * 2))
  in fn () => f 1 end
```
Getting it wrong

Tries to use a variable before it is defined.

```ml
val ones_bad = Scons (1, ones_bad)
```

Would call `ones_worse` recursively immediately (and thus infinitely). Fortunately does not type-check.

```ml
fun ones_worse () = Scons (1, ones_worse ())
```

**Correct:** thunk that returns Scons of value and stream (thunk).

```ml
fun ones () = Scons (1, ones)  
val rec ones = fn () => Scons (1, ones)
```

---

Lazy by default?

**ML:**
- Eager evaluation. Explicit emulation of laziness when needed (promises).
- Immutable data, bindings. Explicit mutable refs when needed (refs).
- Side effects anywhere.

**Pros:**
- avoid unnecessary work, build elegant infinite data structures.

**Cons:**
- difficult to predict evaluation order → difficult to control/predict:
  - Space usage: when will environments become unreachable?
  - Side-effect ordering: when will effects execute?

**Haskell:**
- Non-strict evaluation, except pattern-matching. Explicit strictness when needed.
- Usually implemented as lazy evaluation.
- Immutable everything. Emulate mutation/state when needed.
- Side effects banned/restricted/emulated.

---

Bonus: Memoization

see memo.sml

**Not delayed evaluation, but...**

- Promises (call-by-need) are memoized thunks (call-by-name), though memoization is more general (more multiple arguments).
- Can use an indirect recursive style similar to streams (without delay)
  - Actually fixpoint...

**Basic idea:**

- Save results of expensive pure computations in mutable cache.
- Reuse earlier computed results instead of recomputing.
- Even for recursive calls.

**Benefits:**

- Save time when recomputing.
- Can reduce exponential recursion costs to linear (and amortized by repeated calls with same arguments).

See also: dynamic programming (CS 231)