CS 251 Part 3: When Things Happen
Delay and Laziness

When are expressions evaluated?

Bonus: memoization
Topics

• Eager evaluation order (review)
  – call-by-value
• Delayed evaluation with thunks
  – emulating call-by-name
• Lazy evaluation with promises
  – emulating call-by-need
• Infinite sequences with streams
• Memoization (bonus)
Eager evaluation: 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

```plaintext
fun fact n = 
  if (n = 0) then 1 else (n * (fact (n - 1)))

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

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

What's wrong?
Delayed evaluation with thunks

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

Thunk  fn ( ) => e

- n. a zero-argument function used to delay evaluation
- v. to create a thunk from an expression:
  "thunk the expression"

No new language features.

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

fun fact n =
  if_by_name (fn () => n = 0)
    (fn () => 1)
    (fn () => n * (fact (n - 1)))
Thunk: evaluate when value needed

explicit 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

- # evaluations?
- Faster?
- Slower?
- Side effects?
**Lazy evaluation**: first time value is 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
Promises: delay and force
\textit{(a.k.a. suspensions)}

structure Promise :> PROMISE =
struct
  datatype 'a promise = Thunk of unit -> 'a
  | Value of 'a

  type 'a t = 'a promise ref

  fun delay thunk = ref (Thunk thunk)

  fun force p =
    case !p of
      Value v => v
    | Thunk th =>
      let val v = th ()
      val _ = p := Value v
      in v end
end

Limited mutation hidden in ADT.
Stream: infinite sequence of values

Infinite sequence:
  – Cannot make all the elements now.
  – Make one when asked, delay making the rest.

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

Examples:
  – UI events
  – UNIX pipes: `git 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.

\[ \text{fn} () => (\text{next_element}, \text{next_thunk}) \]

Given stream \( s \), get elements:
– First: let \( \text{val} (v_1,s_1) = s () \)
– Second: \( \text{val} (v_2,s_2) = s_1 () \)
– Third: \( \text{val} (v_3,s_3) = s_2 () \) ...
Streams in ML: recursive types

Single-constructor datatype allows recursive type:

```ml
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.

\[
\text{fun } \text{firstindex } f \text{ stream } = \\
\begin{align*}
\text{let fun } \text{consume } stream \text{ stream } acc &= \ \\
\text{let val } Scons (v,s) &= \text{stream } () \ \\
\text{in } \\
\text{if } f \ v \\
\text{then } acc \\
\text{else consume } s (acc + 1) \\
\text{end} \\
\text{in con}sume \text{ stream } 0 \text{ end}
\end{align*}
\]

\[\text{: ('a -> bool) -> 'a stream -> int}\]
Stream producers

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.

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

Stream producers

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.

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.

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

Would call ones_worse recursively *immediately* (infinitely). Does not type-check.

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

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

```
fun ones () = Scons (1, ones)
val rec ones = fn () => Scons (1, ones)
```
Bonus: Lazy by default?

ML:
- Eager evaluation. Explicitly emulate laziness when needed (promises).
- Immutable data, bindings. Explicit mutable cells when needed (refs).
- Side effects anywhere.

Pros: avoid unnecessary work, build elegant infinite data structures.
Cons: difficult to control/predict evaluation order:
- Space usage: when will environments become unreachable?
- Side-effect ordering: when will effects execute?

Haskell: canonical real-world example
- 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 (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)