delayed evaluation; thunks; streams

slides created by Marty Stepp
http://www.cs.washington.edu/341/
Lazy evaluation

• **lazy evaluation**: delaying a computation until it is needed 
  *(or skipping it entirely, if its result is never used)* 
  *(or avoiding re-computing a previously computed value)*

• Where are some places Java uses lazy evaluation?
  - short-circuiting booleans with && and ||
  - skip evaluation of the un-taken branch of an if/else
  - (advanced) interning of strings
  - (advanced) classes are not loaded until they are referenced
• Scheme mostly uses eager evaluation, but ...

• unused branches of if/cond aren't evaluated

  (if test
   expr1  ; true case
   expr2) ; false case

  ▪ How could we verify that this is so?
• suppose we have the following procedure:

\[
\text{(define (foo b e1 e2)}
\begin{align*}
&\text{(if b)} \\
&\quad (+ e1 e1 e1) \quad \text{; true case} \\
&\quad (* e2 e2)) \quad \text{; false case}
\end{align*}
\]

• will the following code evaluate both the expressions?

\[
\text{(foo #t (+ 2 3) (* 4 5))}
\]

- why or why not?
Procedures with side effects

• suppose we create a procedure with a side effect:
  
  (define (square x)
    (display "squaring ")
    (display x) (newline)
    (* x x))

• what output will the following code produce?
  
  (if (> 2 3) (square 4) (square 7))
• with the previously defined `square` plus the code below:

```
(define (foo b e1 e2)
  (if b
    (+ e1 e1 e1) ; true case
    (* e2 e2))) ; false case
```

• what output will the following code produce?

```
(foo (> 2 3) (square 4) (square 7))
```

- How can we modify it to evaluate only one of the two?
Thunks

- **thunk**: A piece of code or wrapper function used to perform a delayed computation.
  - a value that has already been "thought of"...think → thunk
  - first used in the influential ALGOL-60 language's compiler
  - also used as compatibility wrappers; in DLLs, inheritance...

- thunks are implemented as *zero-argument procedures*
  - instead of passing expression \( e \) (costly to compute?), pass a 0-arg procedure that, when called, computes/returns \( e \)
• we can modify our foo procedure to accept thunks:

\[
\text{(define (foo b th1 th2)}
\]
\[
\quad (\text{if b)}
\]
\[
\quad \quad (+ (\text{th1)} (\text{th1)} (\text{th1)}) \text{ ; true case)}
\]
\[
\quad \quad (* (\text{th2)} (\text{th2)})) \text{ ; false case)}
\]

• we'll also modify our call to pass two thunks:

\[
\text{(foo (> 2 3)}
\]
\[
\quad (\text{lambda ()} (\text{square 4)}))
\]
\[
\quad (\text{lambda ()} (\text{square 7)}))
\]

- now what output does the call produce?
Problem: re-evaluating thunks

- our foo procedure evaluates each thunk multiple times:

```
> (foo (= 2 2)
   (lambda () (square 4))
   (lambda () (square 7)))
```

- squaring 4
- squaring 4
- squaring 4
- squaring 4
- 16

- how can we stop it from re-computing the same value?
Language support for delays

(delay \textit{procedure call})

- some langs. include syntax to ease delayed computation
- delay accepts a call and, rather than executing it, wraps it in a structure called a \textit{promise} that can execute it later:

\begin{verbatim}
> (define x (delay (square 4)))
> x
#<struct:promise:x>
\end{verbatim}
Forcing a delayed execution

(force \textit{delay})

- \texttt{force} accepts a promise, executes it (if necessary), and returns the result

\begin{verbatim}
> (define x (delay (square 4)))
> x
#<struct:.promise:x>
> (force x)
16
> x
#<struct:promise!4>
\end{verbatim}
Use the force, Luke...

• we can modify our `foo` procedure to accept promises:

```scheme
(define (foo b p1 p2)
  (if b
      (+ (force p1) (force p1) (force p1))
      (* (force p2) (force p2))))
```

• we'll also modify our call to pass two promises:

```scheme
(foo (> 2 3)
   (delay (square 4))
   (delay (square 7)))
```

- now what output does the call produce?
• **stream**: An "infinite" list.
  - example: the list of all natural numbers: 1, 2, 3, 4, ...

• Whuck?
  - can't *actually* be infinite, for obvious reasons
  - but *appears* to be infinite, to the code using the list
  - *idea*: delay evaluation of each list pair's tail until needed
    – uses a procedure to describe the element that comes next

• like Unix pipes: `cmd1 | cmd2`; 2 "pulls" input from 1
Streams in Scheme

• a stream is a *thunk* that, when called, returns a pair:

  \((\text{next-answer} \ . \ \text{next-thunk})\)

- first element: \((\text{car } (\text{stream}))\)
- second element: \((\text{car } ((\text{cdr } (\text{stream}))))\)
- third element: \((\text{car } ((\text{cdr } ((\text{cdr } (\text{stream})))))))\)

• nice division of labor:
  - stream's creator knows how to generate values
  - client knows how many are needed, what to do with each
Examples of streams

; an endless list of 1s.
(define ones (lambda () (cons 1 ones)))

; a list of all natural numbers: 1, 2, 3, 4, ...
(define (nat-nums2)
    (define (helper x)
        (cons x (lambda () (helper (+ x 1))))
    )
    (helper 1))

; a list of all powers of two: 1, 2, 4, 8, 16, ...
(define (nat-nums2)
    (define (helper x)
        (cons x (lambda () (helper (* x 2))))
    )
    (helper 1))
(define ones (lambda () (cons 1 ones)))

• accessing the elements of a stream:
  ▪ first element: (car (ones))
  ▪ second: (car ((cdr (ones))))
  ▪ third: (car ((cdr ((cdr (ones))))))
  ▪ fourth: (car ((cdr ((cdr ((cdr (ones))))))))
  ...  

• Remember, parentheses matter! (e) calls the thunk e.
Stream exercises

• Define a stream called `harmonic` that holds the elements of the harmonic series: \(1 + 1/2 + 1/3 + 1/4 + \ldots\)

• Define a stream called `fibs` that represents the Fibonacci numbers. ALL OF THEM!

\[
\begin{align*}
&> \ (\text{car} \ (\text{fibs})) \\
&\quad 1 \\
&> \ (\text{car} \ ((\text{cdr} \ (\text{fibs})))) \\
&\quad 1 \\
&> \ (\text{car} \ ((\text{cdr} \ ((\text{cdr} \ (\text{fibs})))))) \\
&\quad 2 \\
&> \ (\text{car} \ ((\text{cdr} \ ((\text{cdr} \ ((\text{cdr} \ (\text{fibs})))))))) \\
&\quad 3 \\
&> \ (\text{car} \ ((\text{cdr} \ ((\text{cdr} \ ((\text{cdr} \ (\text{fibs})))))))) \\
&\quad 5
\end{align*}
\]
Useful stream procedures

; convenience procedures to create and examine a stream
(define-syntax cons-stream (syntax-rules ()
  ((cons-stream x y) (cons x (delay y)))))
(define car-stream car)
(define (cdr-stream stream) (force (cdr stream)))
(define null-stream? null?)
(define null-stream '())

; returns the first n elements of the given stream
(define (stream-section n stream)
  (cond ((= n 0) '())
        (else (cons (head stream) (stream-section (- n 1)
                                      (tail stream))))))

; merges two streams together
(define (add-streams s1 s2)
  (let ((h1 (head s1)) (h2 (head s2)))
    (cons-stream (+ h1 h2)
                 (add-streams (tail s1) (tail s2))))))
Using the stream procedures

> (define ones (cons-stream 1 ones))
> (stream-section 7 ones)
(1 1 1 1 1 1 1)

> (define (integers-starting-from n)
   (cons-stream n (integers-starting-from (+ n 1))))
> (define nat-nums (integers-starting-from 1))
> (stream-section 10 nat-nums)
(1 2 3 4 5 6 7 8 9 10)

> (define fibs (cons-stream 1
   (cons-stream 1 (add-streams (tail fibs) fibs))))
> (stream-section 14 fibs)
(1 1 2 3 5 8 13 21 34 55 89 144 233 377)