CSE 341:
Programming Languages

Winter 2006

Lecture 14— Delayed Evaluation, Memoization, Thunks, Streams

Today

- Delaying evaluation: Function bodies evaluated only at application

- Key idioms of delaying evaluation

  - Conditionals

  - Streams

  - Laziness

  - Memoization

- In general, evaluation rules defined by language semantics

  - Some languages have "lazy" function application!

Delayed Evaluation

For each language construct, there are rules governing when subexpressions get evaluated. In ML, Scheme, and Java:

- function arguments are "eager" (*call-by-value*)

- conditional branches are not

We could define a language in which function arguments were not evaluated before call, but instead at each use of argument in body. (*call-by-name*)

- Sometimes faster: (lambda (x) 3)

- Sometimes slower: (lambda (x) (+ x x))

- Equivalent if function argument has no effects/non-termination

Thunks

A "thunk" is just a function taking no arguments, which works great for delaying evaluation.

- Instead of passing a value directly, pass a thunk (function) which yields the value when it is called

If thunks are lightweight enough syntactically, why not make if eager? (Smalltalk does this!)

Streams

- A stream is an "infinite" list — you can ask for the rest of it as many times as you like and youll never get null.

- The universe is finite, so a stream must really be an object that acts like an infinite list.

- The idea: use a function to describe what comes next.

Note: Deep connection to sequential feedback circuits

Note: Connection to UNIX pipes

Best of both worlds?

The "lazy" (*call-by-need*) rule: Evaluate the argument the first time it's used. Save answer for subsequent uses.

- Asymptotically it's the best

- But behind-the-scenes bookkeeping can be costly

- And it's hard to reason about with effects

  - Typically used in (sub)languages without effects

- Nonetheless, a key idiom with syntactic support in Scheme

  - And related to *memoization*

Memoization

A "cache" of previous results is equivalent if results cannot change.

- Could be slower: cache too big or computation too cheap

- Could be faster: just a lookup

  - On homework: An example where it's a *lot* faster by preventing an exponential explosion.

An association list is not the fastest data structure for large memo tables, but works fine for 341.

Question: Why does `assoc` return the pair?