CSE 413
Programming Languages & Implementation

Hal Perkins

Autumn 2014

Delayed Evaluation, Thunks, Streams, Memoization

Today

- Racket top-level: forward references and evil mutation
- `cons` and mutable `mcons` cells
- Delaying evaluation: Function bodies evaluated only at application
- Key idioms of delaying evaluation
  - Conditionals
  - Laziness
  - Streams
  - Memoization
- In general, evaluation rules defined by language semantics
  - Some languages have "lazy" function application!

Top-level definitions

Racket top-level allows forward references and mutation of bindings

- Racket (and Scheme) do have assignment: `(set! x e)`
  - But used *only when really! appropriate!!*
- What should a name clash do? (In fact, it's mutation.)
- How can you program defensively?
  - General point: Make a local copy!
- What do Racketeers do in practice?
  - Don't mutate top-level bindings
  - Use a module system for namespace management

cons and mcons

- `cons` just makes a pair
  - By convention and standard library, lists are nested pairs that eventually end with `null`
- In Racket, `cons` cells are immutable (several good reasons for this)
- `mcons` cells are mutable — mutable pairs are sometimes useful
  - Racket has a parallel universe of functions for these: `mcons`, `mcar`, `mcdr`, `mpair?` (also `mlist` and more if you put `(require racket/mpair)` at the top of your code)
  - Can mutate the car and cdr of a `mcons` cell with `set-mcar!` and `set-mcdr!`

Delayed Evaluation

For each language construct, there are rules governing when subexpressions get evaluated. In Racket, Java, and most conventional languages:

- 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 *only* if function arguments have no side effects and terminate when evaluated

Thunks

We know how to delay evaluation: put expressions in a function!

- Behave just the same thanks to closures
- Call the function when you need the value

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

- Can be verbed: *thunk* the expression

Example: Can't define `if` with eager evaluation, but can with thunks.

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 Racket
  - Which we reimplemented with `force-eval` and `delay-eval`
  - And related to *memoization*

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 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

- One new value on each clock cycle

Note: Connection to UNIX pipes

- `cmd1 | cmd2` has `cmd2` "pull" data from `cmd1`.

Streams in Racket

A pretty straightforward idiom:

- A stream is a thunk that when called returns a pair:

  `(next-answer . next-thunk)`

- So "going another iteration with result `pr`" is `((cdr pr))`
- One thunk creating another thunk: use recursion
- Nice division of labor:
  - stream-creator knows how to generate values
  - stream-client knows how many are needed and what to do with each
- (No new semantics; just new idiom)

Using Streams

Given a stream `st`, the client can get any number of elements

- First: `(car (st))`
- Second: `(car ((cdr (st))))`
- Third: `(car ((cdr ((cdr (st))))))`

(Usually bind `(cdr st())` to a variable or pass it to a recursive function)

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
- In our fibonacci example it turns an exponential algorithm into a linear algorithm

An association list is not the fastest data structure for large memo tables, but works fine for 413.

Question: Why does `assoc` return the pair?