Techniques for Supporting Lazy Evaluation

Comp 311
Rice University
Corky Cartwright

Approaches to Hacking Lazy Evaluation

- Mainstream programming languages discourage the use of lazy evaluation by using call-by-value argument passing in methods/procedures, the primary mechanism for defining new program operations.
- There are good software engineering justifications for this bias. Supporting a coherent, intellectually tractable formulation of call-by-name argument passing requires a truly radical language design like Haskell, but this design is so radical that all data constructors are lazy by default! Call-by-name and mutation interact with horrible results. Modern languages with the exceptions of Haskell which has no mutation and Scala where the inclusion of support for call-by-name is really an implicit admission that this language is “For Experts Only”.
- Nevertheless, there are straightforward ways to “hack” support for lazy evaluation in many mainstream languages.
  - Manual use of thunks (suspensions) and evaluation (forcing) which is notationally painful.
  - Macros (the only notationally clean way to do it)

Using Thunks to Defer Evaluation

- In Racket, what construct suppresses evaluation of program text? In fact, this property holds for all languages that provide reasonable support for functions as data.
  Explicitly encapsulate the program text for evaluation later. How can we do this?
  By making the program text the body of a function of no arguments (in ML a unary function that takes the degenerate type `Unit`) that we can evaluate on demand.
- To make the Racket cons operation effectively lazy, we pass it the arguments **`(lambda () `*`M`*`)`** and **`(lambda () `*`N`*`)`** instead of ***`M`*** and ***`N`***. How do we observe the values of the **`first`** and **`rest`** portions of such a list **`l`**? By evaluating **`((first l))`** and **`((rest l))`**. If the **`rest`** has been constructed using laziness, all that **`((rest l))`** evaluates is ***`N`*** which performs an effectively lazy **`cons`** by wrapping its two arguments in thunks.

Improving the Ugly Notation

- Wrapping all argument to lazy constructions in thunks and explicitly applying all of the values embedded in such constructions using application (to no arguments, except in ML languages where the application is to the degenerate **`unit`** value) is ugly, ugly, …
- The workaround: define lazy constructors as macros that expand to the corresponding strict constructor composed with thunk wrapping for each argument.
- What is a macro? A rule performed by the compiler that expands a macro invocation (which typically looks just like a function application) into standard language code that actually implements the macro operation.

Example: a Racket Macro for Lazy cons

- The workaround: define lazy constructors as macros that expand lazy constructor applications to application of the corresponding strict constructor composed with thunk-wrapping each argument.
- What is a macro? A rule performed by the compiler that expands a macro invocation (which typically looks just like a function application) into standard language code that actually implements the macro operation.
- Macros are under-utilized in modern languages because surface (concrete) program syntax is so ugly and messy to manipulate. Strings separated by varying amounts of whitespace. Ugh!
- Programs conceptually have an intelligible tree-based (abstract) syntax that programmers never see. At this level macros are easy to express and understand. The Racket/Scheme/Lisp family of languages is ideal for macros because concrete syntax $\approx$ abstract syntax.

Example cont.

- Racket incorporates a very sophisticated macro system
- Simply macros are defined using the construct **`define-syntax-rule`**.
- To learn more about Racket macros, read the Chapter 16 of the Racket Guide (bundled as part of your DrRacket installation) entitled *Macros.*
- Using **`define-syntax-rule`**, we can easily define lazy-cons, lazy-first, and lazy-rest as follows:

  #lang racket

  ```
  (define-syntax-rule (lazy-cons f r)
    (cons (lambda () f) (lambda () r)))
  (define-syntax-rule (lazy-first lc) ((car lc)))
  (define-syntax-rule (lazy-rest lc) ((cdr lc)))
  ```

  Note: The standard dialect of Racket supports macros, while the HTDP languages do not. Standard Racket requires the use of **`car`** and **`cdr`** above instead of **`first`** and **`rest`** for technical reasons.

Example cont.

- The simple functional code in our macros is not efficient because it re-computes the values of expressions! (In a purely functional language we never need to evaluate an expression more than once! Why?)
- How do we avoid re-computation in functional languages?
  - Factor out common sub-expressions using local
  - If our functional language accommodates mutation (Racket/Scheme/Lisp/ML except Haskell), we can use benign mutation to cache values when factoring is insufficient (e.g., naïve Fibonacci)
  - Important optimization in many contexts, not just lazy evaluation.

Memoization

- Most important manual optimization in functional programming, yet it is not functional!
- Rule of thumb: mutation is OK if it is encapsulated!
- Such mutation is “assign once” changing unbound (often represented by a default value such as 0 or empty) to a binding.
- In standard memorization, recursive calls are recorded in a table (often a hash table) and function evaluation avoids performing the same computation by consulting the table before executing the body.
- We are going to take a glimpse at the core imperative features of Racket in the next lecture, but you will not have to write any imperative code in Racket; I find this form of optimization more natural in the context of Java.