Techniques for Supporting Lazy Evaluation and Call-by-name

Comp 311
Rice University
Corky Cartwright

Hacking Lazy Evaluation and Call-by-name

- Mainstream programming languages discourage the use of lazy evaluation by only supporting strict constructors and 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 *with no side effects*, but Haskell is so radical that all data constructors are lazy as well. The conjoining of either call-by-name or lazy evaluation with mutation generates horrible results. Modern languages with the exception 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") do not support call-by-name evaluation. Experts know that call-by-name should only be used with argument expressions with no side-effects.
- Nevertheless, there are straightforward ways to "hack" support for lazy evaluation and call-by-name in many mainstream languages.
  - Manual use of thunks (described on the next slide) which is notationally ugly.
  - Macros to support clean notation for lazy evaluation in call-by-value languages).

Using Thunks to Defer Evaluation

- In Racket, what construct suppresses evaluation of program text? **`lambda`**-abstraction. In fact, this property holds for all languages that provide reasonable support for functions as data. We need to 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 we define unary functions that take an input of the degenerate argument type **`Unit`** which only has one element (denoted **`()`**) which is never used.
- 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 lazy list **`l`**? By evaluating **`((first l))`** and **`((rest l))`**. If **`l`** is a thunked lazy cons construction, all that **`((rest l))`** evaluates is the body of the second thunk (*N* in our example).
- This approach is mathematically clean but nearly unreadable.

Improving the Ugly Notation

- Wrapping all arguments to lazy constructions in thunks and explicitly applying all of the values embedded in such constructions using application to no arguments (in languages in the ML family where lambda-abstractions must have at least one argument, 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 syntactic rule performed by the compiler that expands a macro invocation (which typically looks just like a function application) into standard source code that actually implements the macro operation.

Brief Aside Regarding Macros

- The basic idea behind macros is extremely simple: macros are syntactic abbreviations. A macro has free metavariables that stand for program expressions. Neither the left-hand side or right-hand side of a macro can mention program variables. In Core Racket, the **`delay`** primitive in Advanced HTDP Racket dialect *cannot* be defined as a Core Racket function! Why? Because all Racket functions defined using **`lambda`** or **`define`** are strict, but **`delay`** is not. On the other hand, a version of **`delay`** is trivial to define as a macro abbreviation:

```
(delay M) => (lambda () M)
```

- Given this formulation of **`delay`**, it is trivial to define the **`force`** function

```
(define (force s) (s))
```

Brief Aside Regarding Macros II

- Racket uses a slightly more complex representation so it can recognize delayed values using the predicate **`promise?`**, namely by wrapping the thunk in a built-in unary constructor called **`promise`**:

```
(define-struct promise (thunk))
(delay M) => (make-promise (lambda () M))
(define (force s) ((promise-thunk s)))
```

- Most macro systems go far beyond simple abbreviations, enabling fancy macros to introduce new variable bindings. Keeping macro variables separate from program variables (from the context where the macro is expanded) is a surprisingly subtle language design problem and most macro systems get it wrong. This issue has been extensively studied in the Scheme literature under the subject heading of "hygienic macros".
- The HTDP Advanced language includes support for macros.

Using Macros to Implement Lazy Constructors

- We can implement any lazy constructor as a macro that maps its argument expressions to the application of a corresponding strict constructor to thunks wrapping the argument expressions as lambda-abstractions--just as we implemented **`delay`** as the strict **`promise`** constructor applied to a **lambda**-abstraction wrapping the delayed argument.
- Macros are under-utilized in modern languages because surface (concrete) program syntax is simply a sequence of tokens rather than a tree with internal structure. So macros map strings to expanded strings which may or may not be translated to the intended syntax by the language parser because the precedence rules governing the parsing of language source text are complex. To prevent misinterpretation, programmers typically include extra parentheses in such macros, making them difficult to read. C macros are a great example.
- All high-level programming languages conceptually have an intelligible tree-based (abstract) syntax that programmers never see. In this representation, macros are easy to express and understand. The Racket/Scheme/Lisp family of languages is ideal for macros because the funky parenthesized concrete syntax of Lisp is similar in structure to abstract syntax.

Representing Lazy **cons** as a Macro

- Racket has a very sophisticated macro system but it is not included in any of the HTDP dialects. Select the "Racket" language to run the code below
- In Racket simple macros are defined using the construct **`define-syntax-rule`**.
- To learn more about Racket macros, read "Fear of Macros", a document linked from the Racket Guide or 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: In contrast to the HTDP languages, the "Racket" language requires the use of **`car`** and **`cdr`** instead of **`first`** and **`rest`**. I presume that Lisp tradition is being respected.

Example cont.

- The simple functional code in our macros is not efficient because it re-computes the values of expressions! The thunks embedded inside a **`lazy-cons`** construction can be evaluated many times. More elaborate macros can be defined that include updateable cells inside the **`lazy-cons`** construction so that the construction arguments are only evaluated once.
- How do we avoid re-computation in functional languages?
  - Factor out common sub-expressions using **`local`** or **`let`**.
  - 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). This optimization is often called memoization.

Memoization

- Most important manual optimization in functional programming, yet it is not functional!
- Rule of thumb: mutation is OK if it is encapsulated (externally invisible)!
- Common special case: the mutated cell is *quasi-constant*: “not yet computed” or a constant.
- 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.