Lazy Evaluation or Non-strict Constructors

Comp 311
Rice University
Corky Cartwright

Some Basic Definitions

- The element $\perp$ (called bottom) implicitly exists in all data types (domains) because we can write a divergent function of any type using recursion. What does $\perp$ evaluate to? Nothing! It diverges. $\perp$ is not a value! In contrast to an error, $\perp$ does not appear as an discrete event during evaluation.
- Many computer scientists (following logicians like Kleene) prefer to leave divergence implicit and only talk about total functions (functions that never diverge). This is a widely held point-of-view. The logical framework Coq for specifying the behavior of programs and proving their properties (correctness) only supports total functions.
- I dissent from this view. Functional programs inherently define partial functions. Some of them (most of the ones we use in practice) are total (assuming we consider errors as legal return values). It is easy to take logical theories of computational domains (e.g., Peano's axioms for the natural numbers) and slightly revise them to include $\perp$ and errors.

Strictness

- A conventional primitive function that evaluates all of its arguments is *strict*: if it diverges if any of their arguments are $\perp$ (ignoring aborting errors).
- If we include aborting errors, a conventional primitive function that evaluates all of its arguments is *error-strict*: if an argument a_i diverges or returns an aborting error element *and* all preceding arguments evaluate to non-error elements, then the function returns the result of evaluating a_i.
- Conventional constructors (as in Racket **`define-struct`**) are conventional primitive functions.
- Lazy constructors are not conventional constructors. Moreover, they never diverge or return elements. (Note: I am confining my attention to constructors that are non-strict in all arguments. Some constructors are only lazy in selected argument positions.)
- An n-ary lazy constructor **`c`** takes n argument expressions $M_1, \ldots, M_n$ and leaves them unevaluated. It returns a *value* $c(M_1, \ldots, M_n)$ called a *lazy value* or a *lazy construction*. Programming languages differ on whether they support equality of lazy constructions.

Lazy data types (domains)

- Every lazy constructor **c** has an associated type (unimaginatively called) **c** consisting of the elements
  $\{\perp\} \cup \{\mathbf{c}(v_1). \ldots, \mathbf{c}(v_n)\}$
- where $v_1 \ldots, v_n$ are arbitrary Lazy Racket values.
  In statically typed languages, each v_i is restricted to a specified type.
- But there is a catch. Values are closed under infinite ascending chains of finite values where the ordering is tree approximation. A finite tree approximates itself and any elaboration (replacing of bottom by a value) of itself where a bottom is replaced by another value.

Lazy data types (domains)

- Every lazy constructor **c** has an associated type (unimaginatively called) **c** consisting of the elements
  $\{\perp\} \cup \{\mathbf{c}(v_1). \ldots, \mathbf{c}(v_n)\}$
- where $v_1 \ldots, v_n$ are arbitrary Lazy Racket values.
  In statically typed languages, each v_i is restricted to a specified type.
- But there is a catch. Values are closed under infinite ascending chains of finite values where the ordering is tree approximation. A finite tree approximates itself and any elaboration (replacing of bottom by a value) of itself where a bottom is replaced by another value.

Lazy Racket and LazyRacket

- In DrRacket, the definition of Lazy Racket (available as an "experimental" language is a mess from a semantic point-of-view, *i.e.*, its reduction semantics is quirky and requires introducing faux constructors.
- Problem every **`define`** operation wraps its right hand side in **`delay`**, which is a unary lazy faux constructor with **`force`** as its field name. The **`cons`** constructor is unchanged except for implicitly wrapping its arguments with **`delay`**. Explicit use of the **`delay`** operation is an ugly hack, but it was done for the sake of compatibility with ordinary Racket.
- In its place, we are going to define LazyRacket for purposes of hand evaluation only. In LazyRacket, **`lambda`** abstractions are call-by-name and **`cons`** is truly lazy. The reduction semantics for LazyRacket is identical to that for ordinary Racket except for beta-reduction rules, the rules for reducing applications primitive operations **`cons`**, **`first`**, and **`rest`**, and a small change to the definition of values.

LazyRacket Semantics

- The beta-reduction rule in LazyRacket is:

  `((lambda (`x_1 … x_n`) E)` M_1 … M_n) => $E_{[M_1 \text{ for } x_1] \ldots [M_n \text{ for } x\_n]}$

  where $E_{[M_1 \text{ for } x_1] \ldots [M_n \text{ for } x\_n]}$ means **E** with all free occurrences of x_1 … x_n replaced by M_1 … M_n. We duck the complication of *safe-substitution* by prohibiting the *reuse* of variable names bound in the sequence of **`define`** operations at the beginning of a program. (Recall Problem 5 on Homework 3.) In other words, variables that are bound by **`define`** must be unique. This reduction rule is the beta-reduction rule from the classic lambda calculus.

- There only other changes from the evaluation rules for Core Scheme are described on the next slide.

LazyRacket Semantics cont.

- The definition of value differs in one respect, namely that all applications of the form

  (cons M_1 M_2**)**

  are values.
- The reduction rules for **`first`** and **`rest`** are revised to match this change in the definition of values:

  (first (cons M_1 M_2**))** => M_1

  (rest (cons M_1 M_2**))** => M_2
- Nothing else changes from Core Racket!

Examples

- Problem 1 from HW03

```
    (and false (/ 1 0)) => false
=> false
```

- Problem 2 from HW03

```
     (define AND (lambda (x y) (if x y false)))
     (AND false (/ 1 0))
 => ...
     ((lambda (x y) (if x y false)) false (/ 1 0))
 => (if false (/ 1 0) false)
 => false
```

- These examples only show the differences between call-by-name and call-by-value beta reduction.

More Examples

Simple example involving lazy primitive operations

```
    (define zeros (cons 0 zeros))
    (first zeros)
=> ...
    0
```

What would this program mean in Core Racket

```
    (define zeros (cons 0 zeros))
    (first zeros)
=> run-time error: zeros is undefined
```

Why? The meaning of **`cons`** is different!

More Examples cont.

Simple example involving lazy primitive operations

```
    (define zeros (cons 0 zeros))
    (rest zeros)
=>  ...
    (rest (cons 0 zeros))
=>  ...
    zeros
=>  ...
    (cons 0 zeros)
```