Lazy Evaluation or Non-strict Constructors

Comp 311
Rice University
Corky Cartwright

Some Basic Definitions

- The element ⊥ (called bottom) implicitly exists in all data types (domains) because we can write a divergent function of any type using recursion. What does ⊥ evaluate to? Nothing! It diverges. ⊥ is not a value! In contrast to runtime errors, ⊥ 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. Even imperative programs are inherently partial. 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 ⊥ and errors.

Strictness

- A conventional primitive function that evaluates all of its arguments is *strict*: 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 the **`make-`**`<struct-name>` operations created by Racket **`define-struct`**) are conventional primitive functions.
- Lazy constructors are *not* conventional constructors; they never diverge or return error elements. (Note: I am confining my attention to constructors that are non-strict in all arguments. Some constructors are lazy for selected arguments.)
- 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 with lazy constructors differ on whether they support equality testing of lazy constructions. Of course, the `equals?` operator diverges when applied to two identical infinite lazy trees. It also diverges or aborts with an error if it inspects a tree node that is $\perp$ or throws an error during the comparison process.

Lazy data types (domains)

- Every lazy constructor **c** has an associated type (unimaginatively called) **c** consisting of the elements
$\{\perp\} \cup \{$**c**(v_1). …, **c**(v_i), …$\}$ (assuming **c** is unary)
- where v_1 …, $v_{i,}$ … are arbitrary Lazy Racket values.
In statically typed languages, each v_i is restricted to a specified type.
- But there is a catch. The domain of program values is closed under infinite ascending chains of finite values where the ordering is tree approximation. Limit points of ascending chains of finite elements (infinite trees) are values in an intuitive sense, but computations only consider all finite approximations. A finite tree approximates itself and any elaboration of itself where a bottom is replaced by another value.

Tree Approximation

- In Racket with lazy constructors, consider any Racket expression *M* built solely from lazy constructors and atomic constants. *Without loss of generality*, we assume that the outermost constructor is **`cons`**. So *M* must have the form

  `(cons` *`U V`***`)`**

  where *U* and *V* are Racket expressions built solely from lazy constructors.
- Without evaluating *U* and *V*, we do not know what the left and right subtrees of *M* are. But we know **`(cons?`** *`M`***`)`** is **`true`**. In our semantic model,

  `(cons ⊥ ⊥)`

  approximates all (lazy) trees with **`cons`** at the root.
- In general, a tree with some ⊥ leaves approximates any tree that replaces the ⊥ leaves by values (which may contain embedded ⊥ leaves). Hence,

  `(cons ⊥ ⊥) ⊑ (cons ⊥ (cons ⊥ ⊥))`

  where ⊑ denotes the approximation relation on lazy trees. **`(cons ⊥ ⊥)`** also approximates infinitely many other finite and infinite trees,

Lazy Racket and LazyRacket

- DrRacket includes support for a language called Lazy Racket that is "experimental" and not particularly well implemented. Here is a sample Lazy Racket program (with two expressions following the **`define`** block) and its evaluation that drives this point home:

```
    (define a (/ 1 0))
    (define id (lambda (x) x))
    a
    (id a)
→   ...
    (delay …)
    :division by zero
```

Note that (*i*) **`a`**, (*ii*) the identity function applied to **`a`** , and (iii) do not produce the same results when evaluated. You can try this example in DrRacket on your own if you are interested. Lazy Racket delays the evaluation of variables at the top level which is wrong because it breaks the identity of a bound variable and the value to which it is bound. The evaluation of **`a`** should generate a **`:division by zero`** error. **`a`** and the application **`(/ 1 0)`** should have the same meaning. We will ignore Lazy Racket..

Lazy Racket and LazyRacket

- We will define our own lazy dialect of lazy Racket called LazyRacket (one word) based on Core Racket. In fact, the syntax of LazyRacket is identical to the syntax of Core Racket. The only difference will be the rules for evaluating function applications (the beta-reduction rule) and constructor applications (including **`cons`**).
- In LazyRacket, **`lambda`**-abstractions are reduced using the *call-by-name* version of beta-reduction, which is the form of beta-reduction that appears in the original pure lambda-calculus. The only other change from the semantics of Core Racket is that all constructors including **`cons`** are lazy (non-strict). As a result, the reduction semantics for LazyRacket is identical to that for Core Racket except for three small (but very significant) changes.
    1. To support call-by-name evaluation of program-defined functions, the definition of beta reduction is slightly different (as shown on the next slide).
    2. To support the lazy evaluation of constructors, we add all expressions of the form
       (c M_1 M_2 ... M_n)
       to the set of *values.* Hence, there is no evaluation inside constructions just as there is no evaluation inside **`lambda`**-abstractions.
    3. the reduction rules for the applications of accessor operations like **`first`** are slightly revised as shown on the next slide.

LazyRacket Semantics

- The *call-by-name* beta-reduction rule in LazyRacket is:

  ((lambda (x_1 **...** x_n**) E)** M_1 **...** M_n**)** ⇨ $E_{[M_1 \texttt{ for } x_1] \ldots [M_n \texttt{ for } x\_n]}$

  where $E_{[M_1 \texttt{ for } x_1] \ldots [M_n \texttt{ for } x\_n]}$ means **E** with all free occurrences of x_1, …, x_n replaced by M_1, …, M_n. We can 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.
- The only other changes to the evaluation rules for Core Scheme are a change to the definition of *values* described on the previous slide and a slight revision of the rules for reducing applications of accessors shown on the next slide.

LazyRacket Semantics cont.

- The reduction rules for construction accessors such as **`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

  The reduction rules for applications of declared accessors are analogous.
- Nothing else changes from Core Racket!

Examples

- Problem 2 from HW03 in LazyRacket rather than Core Racket

```
    (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
```

- Lazy construction

```
     (rest (cons (/ 1 0) empty)) => empty
```

More Examples

Simple example in Lazy Racket (using **cons**)

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

What would this program mean in Core Racket

```
    (define zeros (cons 0 zeros))
    (first zeros)
=>  zeros is used before its definition
```

More Examples cont.

Simple example involving lazy primitive operations

```
   (define zeros (cons 0 zeros))
   (rest zeros)                ;; zeros is not a value
=> ...
   (rest (cons 0 zeros))       ;; (cons ...) is a value
=> ...
   zeros                       ;; zeros is not a value
=> ...
   (cons 0 zeros)              ;; (cons ...) is a value
```

Terminology

Beware: in the literature, the term *lazy evaluation* does not have completely consistent usage.

- In some usages, it only refers to the semantics of constructors, which only involves changing the definitions of *value* and the corresponding applications of accessors.
- In others, the term *lazy evaluation* includes using *call-by-name beta reduction* as well. I prefer the former because it embraces the idea that constructors can be lazy in the context of call-by-value language like Core Racket or Java. In Scala, a call-by-value language, constructors can be declared lazy independently of the semantics of beta-reduction.