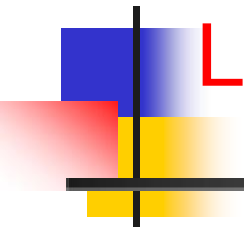
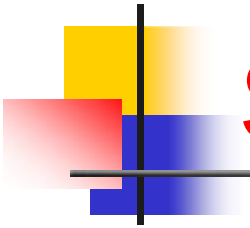


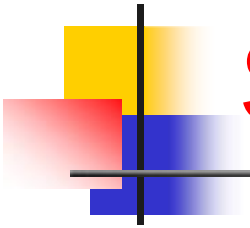
Lazy Evaluation or Non-strict Constructors

Comp 311
Rice University
Corky Cartwright



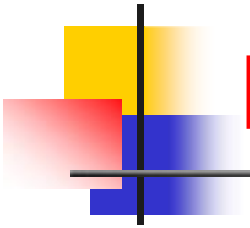
Some Basic Definitions

- 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 some eminent 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.



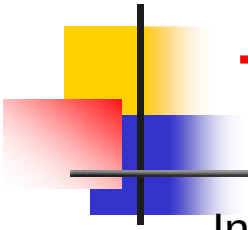
Strictness

- A conventional primitive function that evaluates all of its arguments is *strict*: it diverges if any of their arguments are ⊥ (ignoring aborting errors).
- Assume all data domains include aborting errors. Then a conventional primitive function *f* that evaluates all of its arguments is *error-strict* iff the following condition holds: if an argument a_i diverges or returns an aborting error element *and* all preceding arguments evaluate to ordinary values (non-error elements), then the function returns the result of evaluating a_i. (In the context of aborting errors, we must similarly revise the definition of *strict*.)
- 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. In principle, some constructors are lazy only for selected arguments.)
- An *n*-ary lazy constructor **`c`** takes *n* argument expressions M_1, …, M_n and leaves them unevaluated. It returns a *value* c(M_1, …, 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 ⊥ 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
  $\{\bot\} \cup \{\mathbf{c}(v_1), \ldots, \mathbf{c}(v_i), \ldots\}$ (assuming **c** is unary)
- where $v_1 \ldots, v_i, \ldots$ 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 the lazy domain, but computations only manipulate finite approximations including suspensions (unknown values) . A finite tree approximates itself and any elaboration of itself where a bottom leaf is replaced by another value. Infinite trees are values in the lazy data domain, but only finite approximations are used in actual computations.



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,

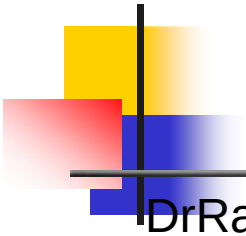
  `(cons` ⊥ ⊥**`)`**

  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`** ⊥ ⊥**`)`** approximates infinitely many other finite and infinite trees,



Lazy Racket and LazyRacket

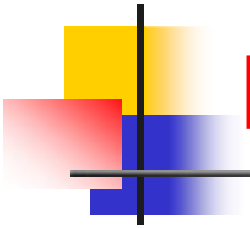
DrRacket includes support for a language called Lazy Racket that is "experimental" and flawed. Here are two sample Lazy Racket programs and their results that illustrate this point:

```
(define AND (lambda (x y) (and x y)))
(AND false (/ 1 0)
→ false

(and false (/ 1 0)
→ (delay ...)
```

I don't think the current DrRacket implementation corresponds to a tractable set of reduction rules; it is simply broken. As any of you who take Comp 411 will learn, it is straightforward to implement an interpreter for lazy Racket. I think the developers of DrRacket simply made some mistakes in the implementation.

We will define our own dialect of lazy Racket (called LazyRacket [one word] as a simple variation of Core Racket.



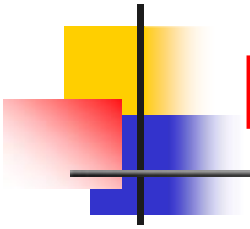
LazyRacket

In fact, the syntax of LazyRacket is identical to the syntax of Core Racket. The only difference between Core Racket and LazyRacket are the rules for evaluating function applications (the beta-reduction rule), constructor applications (including **`cons`**), the definition of *values*, and accessor applications.

- In LazyRacket, **`lambda`**-abstractions are reduced using the *call-by-name* version of beta- reduction (the form of beta-reduction that appears in the original pure lambda-calculus) omitting the *value-restriction* on arguments:

  $$\texttt{((lambda (}x_1\ x_2\ \ldots\ x_n\texttt{) E) }M_1\ M_2\ \ldots\ M_n\texttt{)} \rightarrow \texttt{E[}x_1 \leftarrow M_1\texttt{, }x_2 \leftarrow M_2\texttt{, } \ldots\ x_n \leftarrow M_n\texttt{]}$$

- All constructor applications $\texttt{(c }M_1\ M_2\ \ldots\ M_n\texttt{)}$ are *values*, *i.e.*, the argument expressions M_1, M_2, ..., M_n are **not** required to be values.
- The law for evaluation accessor applications is generalized to address the fact that the selected argument expression is not necessarily a value.

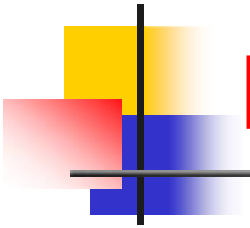


LazyRacket Semantics

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

  ((lambda (x_1 x_2 ... x_n) E) M_1 M_2 ... M_n) → E[x_1 ← M_1, x_2 ← M_2, ... X_n ← M_n]

  where **E[x_1 ← M_1, x_2 ← M_2, ... X_n ← M_n]** means **E** with all free occurrences of **x_1**, …, **x_n** replaced by **M_1**, …, **M_n**. We can duck the complication of using *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 simply 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 extension of the rules for reducing applications of accessors shown on the next slide.



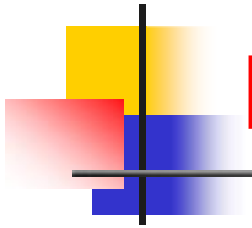
LazyRacket Details

The reduction rules for construction accessors such as **`first`** and **`rest`** are genralized 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.



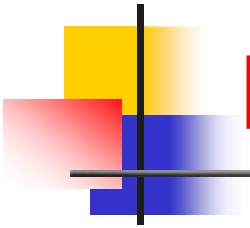
Examples

- 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)   ; beta-reduction yet (/ 1 0) is not a value
=> ...
   false
```

- Accessing a field of a lazy construction

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



More Examples

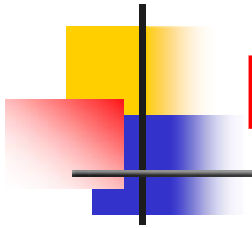
Simple example in Lazy Racket (using **cons**)

```
   (define zeros (cons 0 zeros)) ; rhs is a
   value (first zeros)
=> ...
   (first (cons 0 zeros))
=> ...
   0
```

What would this program mean in Core Racket

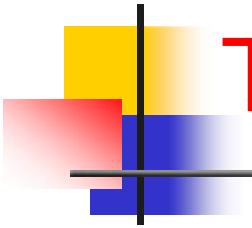
```
    (define zeros (cons 0 zeros))
    (first zeros)
=> zeros is used before its definition
```

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* has one of two different meanings:

- In some usages, it only refers to the semantics of constructors, which only involves changing the definitions of *value* and the corresponding accessor operations.
- 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 function application (call-by-vaue vs. call-by-name beta-reduction).