What does my program mean?

Armando Solar Lezama
Computer Science and Artificial Intelligence Laboratory
M.I.T.

Adapted from Arvind 2010. Used with permission.

September 16, 2015

Meaning of a term

- The semantics must distinguish between terms that should not be equal, i.e., if

  $0 \equiv \lambda x.\lambda y.\ y;\quad 1 \equiv \lambda x.\lambda y.\ x\ y;\quad 2 \equiv \lambda x.\lambda y.\ x\ (x\ y)$

  Then $\lambda x.\lambda y.\ y \neq \lambda x.\lambda y.\ x\ y \neq \lambda x.\lambda y.\ x\ (x\ y)$
- The semantics must equate terms that should be equal, i.e., the terms corresponding to (plus 0 1) and 1 must have the same meaning
- A semantics is said to be *fully abstract* if two terms have different meaning according to the semantics then there exists a term that can tell them apart

In the λ-calculus it is possible to define the meaning of a term almost syntactically

Information content of a term

Instantaneous information: A term obtained by replacing each redex in a term by $\perp$ where $\perp$ stands for no information

Term	Instantaneous information
$(\lambda q.\lambda p.p\ (q\ a))\ (\lambda z.z)$	$\boxed{\perp}$
$\rightarrow \lambda p.p\ ((\lambda z.z)\ a))$	$\boxed{\lambda p.p\ \perp}$
$\rightarrow \lambda p.p\ \ a$	$\boxed{\lambda p.p\ a}$

β-reductions monotonically increase information

The meaning of a term is the maximum information that can be obtained by β-reductions

Is the meaning of a term simply its normal form?

- Yes, but…
- What if a term doesn't have a normal form?
  - Is the meaning of such terms always ⊥ ?

  - Consider Ω = (λx.x x) (λx.x x)
  - It doesn't have a normal form, and its meaning is ⊥
  - What about λx.x Ω ?
  - It doesn't have a normal form, but its meaning is λx.x ⊥

- What if a term has more than one normal form?
  - It would have two meanings. BAD
  - Not possible in the λ-calculus because of *Confluence* …

Can the choice of redexes lead to different meaning?

- A term may have multiple redexes
  1. ((λx.M) A) ((λx.N) B)

     ----- ρ_1---- ---- ρ_2----
  2. ((λx.M) ((λy.N)B))

     ---- ρ_2----

     ---------- ρ_1 --------
- ρ_1 followed by ρ_2 does not necessary produce the same term as ρ_2 followed by ρ_1
  - Notice in the second example ρ_1 can *destroy* or *duplicate* ρ_2 .
- Can our choice of redexes lead us to produce terms that are clearly different (e.g., x versus λy.y)?

Church-Rosser Property

A reduction system is said to have the *Church-Rosser property*, if $E \twoheadrightarrow E_1$ and $E \twoheadrightarrow E_2$ then there exits a E_3 such that $E_1 \twoheadrightarrow E_3$ and $E_2 \twoheadrightarrow E_3$.

$$\begin{tikzcd}
E \arrow[r, two heads] \arrow[d, two heads] & E_1 \arrow[d, two heads, dotted] \\
E_2 \arrow[r, two heads, dotted] & E_3
\end{tikzcd}$$

also known as *CR* or *Confluence*

If a system has the CR property then the divergence in terms due to the choice of redexes can be corrected

Church-Rosser Theorem

Theorem: The λ-calculus is CR.
(Martin-Lof & Tate)

- No satisfactory proof of this theorem was given until 1970 (30 years later!)
- The proof is elegant
- Requires showing how two divergent terms can be brought together in finite number of steps
  - strategy for choosing reductions

CR implies that if NF exists it is *unique*

Interpreters

An *interpreter* for the λ-calculus is a program to reduce λ-expressions to “answers”.

Requires:

- the definition of an *answer*
  - e.g., normal form?
- a *reduction strategy*
  - a method to choose redexes in an expression

Definitions of “Answers”

- *Normal form (NF):* an expression without redexes

- *Head normal form (HNF):*

  x is HNF
  (λx.E) is in HNF if E is in HNF
  (x E_1 … E_n) is in HNF
    Semantically most interesting- represents the information content of an expression

- *Weak head normal form (WHNF):*

  An expression in which the left most application is not a redex.

  x is in WHNF
  (λx.E) is in WHNF
  (x E_1 … E_n) is in WHNF
    Practically most interesting ⇒ “Printable Answers”

Two Common Reduction Strategies

- *applicative order:* right-most innermost redex
  aka call by value evaluation

- *normal order:* left-most (outermost) redex
  aka call by name evaluation

$$\underbrace{(\ \lambda x.y)\ (\underbrace{(\ \lambda x.x\ x)\ (\ \lambda x.x\ x)}_{\rho_2})}_{\rho_1}$$

← applicative order

← normal order

Computing a normal form

1. Every λ-expression does not have an answer *i.e.*, a NF or HNF or WHNF

$$(\lambda x.x\ x)\ (\lambda x.x\ x) \quad = \quad \Omega$$

$$\Omega \rightarrow \Omega \rightarrow \Omega \ \rightarrow \ldots$$

3. Even if an expression has an answer, not all *reduction strategies* may produce it

$$(\lambda x.\lambda y.y)\ \ \Omega$$

leftmost redex: $(\lambda x.\lambda y.y)\ \ \Omega \rightarrow \lambda y.y$

innermost redex: $(\lambda x.\lambda y.y)\ \ \Omega \rightarrow (\lambda x.\lambda y.\ y)\ \ \Omega \rightarrow \ldots$

Normalizing Strategy

A *reduction strategy* is said to be *normalizing* if it terminates and produces an answer of an expression whenever the expression has an answer.

aka *the standard reduction*

Theorem: Normal order (left-most) reduction strategy is normalizing for the λ-calculus.

A Call-by-name Interpreter

Answers: WHNF
Strategy: leftmost redex

Apply the function before evaluating the arguments

cn(E): Definition by cases on E

$$E \;=\; x \mid \lambda x.E \mid E\ E$$

```
$cn([[x]])$              $= x$
$cn([[\lambda x.E]])$    $= \lambda x.E$
$cn([[E_1\ E_2]])$       $= \textit{let}\ f = cn(E_1)$
                         in case f of
                             $\lambda x.E_3 \;=\; cn(E_3[E_2/x])$
                             $\_ \;=\; f\ E_2$
```

⟦…⟧ represents syntax
[[…]] is my ppt approx

Meta syntax

A Call-by-value Interpreter

Answers: WHNF

Strategy: rightmost-innermost redex but not inside a λ-abstraction

Evaluate the argument before applying the function

cv(E): Definition by cases on E

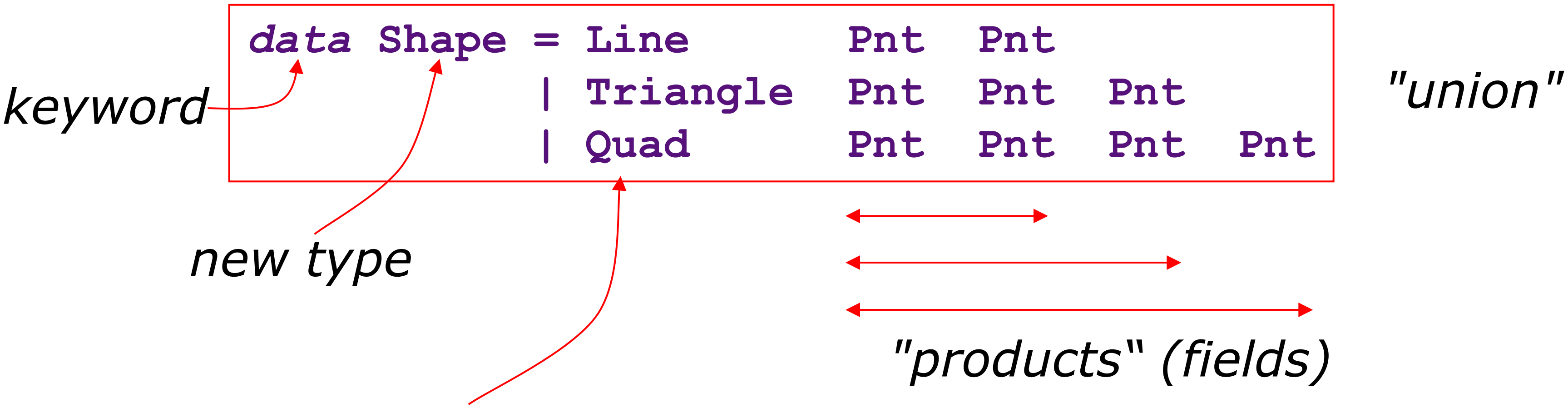
$E \; = \; x \; | \; \lambda x.E \; | \; E \; E$

```
cv([[x]])          =  $x$
cv([[$\lambda x.E$]])     =  $\lambda x.E$
cv([[$E_1$ $E_2$]])      =  let  f = cv($E_1$)
                            a = cv($E_2$)
                       in case f of
                          $\lambda x.E_3$  = cv($E_3$[a/x])
                            _      = f a
```

Coding this in Haskell
with
Algebraic Data Types

Algebraic types

- Algebraic types are *tagged unions of products*
- Example



```
data Shape = Line       Pnt  Pnt
           | Triangle   Pnt  Pnt  Pnt
           | Quad       Pnt  Pnt  Pnt  Pnt
```

keyword

new type

"union"

"products" (fields)

- new "constructors" (a.k.a. "tags", "disjuncts", "summands")
- a k-ary constructor is applied to k type expressions

Examples of Algebraic types

```
data Bool = False | True

data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat

data Maybe a  = Nothing | Just a

data List a = Nil | Cons a (List a)

data Tree a = Leaf a | Node (Tree a) (Tree a)

data Tree′ a b = Leaf′ a
               | Nonleaf′ b (Tree′ a b) (Tree′ a b)

data Course  = Course String Int String (List Course)
```

name number description pre-reqs

Constructors are functions

- Constructors can be used as functions to *create* values of the type

```
let
    l1 :: Shape
    l1 = Line  e1  e2

    t1 :: Shape = Triangle  e3  e4  e5
    q1 :: Shape = Quad  e6  e7  e8  e9
in
    ...
```

where each "eJ" is an expression of type "Pnt"

Pattern-matching on algebraic types

- *Pattern-matching* is used to examine values of an algebraic type

```
anchorPnt :: Shape -> Pnt
anchorPnt s = case s of
                Line     p1 p2       -> p1
                Triangle p3 p4 p5    -> p3
                Quad     p6 p7 p8 p9 -> p6
```

- A pattern-match has two roles:
  - A test: "does the given value match this pattern?"
  - Binding ("if the given value matches the pattern, *bind* the variables in the pattern to the corresponding parts of the value")
- Clauses are examined top-to-bottom and left-to-right for pattern matching

Big Step Semantics

Big Step Operational Semantics

- Model the execution in an abstract machine
- Basic Notation: Judgments

Some alternative notations:
$\Downarrow, \longrightarrow, \leadsto, \Rightarrow, etc$ ...

$$\langle configuration \rangle \rightarrow result$$

  - describe how a program configuration is evaluated into a result
  - the configuration is usually a program fragment together with any state.
- Basic Notation: Inference rules
  - define how to derive judgments for an arbitrary program
  - also called derivation rules or evaluation rules
  - usually defined recursively

$$\frac{\langle c_1 \rangle \rightarrow r_1 \quad \langle c_2 \rangle \rightarrow r_2 \ \ldots \ \langle c_k \rangle \rightarrow r_k}{\langle configuration \rangle \rightarrow result}$$

What evaluation order is this?

$$\frac{}{x \to x}$$

$$\frac{}{\lambda x.\, e \to \lambda x.\, e}$$

$$\frac{e_1 \to \lambda x.\, e_1' \qquad e_1'[\,\alpha(e_2)/x] \to e_3}{e_1\, e_2 \to e_3}$$

Call by Name

```
cn([[x]])       =  x
cn([[λx.E]])  =  λx.E
cn([[E1 E2]])=  let  f = cn(E1)
                in case f of
                   λx.E3  = cn(E3[E2/x])
                   _        = f E2
```

Do these semantics coincide?

What evaluation order is this?

$$\frac{}{x \to x}$$

$$\frac{}{\lambda x. e \to \lambda x. e}$$

$$\frac{e_1 \to \lambda x. e_1' \quad e_2 \to e_2' \quad e_1'[\alpha(e_2')/x] \to e_3}{e_1 e_2 \to e_3}$$

Call by Value

```
cv([[x]])      =  x
cv([[λx.E]]) =  λx.E
cv([[E1 E2]])=  let  f = cv(E1)
                     a = cv(E2)
                in case f of
                     λx.E3  = cv(E3[a/x])
                     _      = f a
```

Recursion and the Y Combinator

Recursion and Fixed Point Equations

Recursive functions can be thought of as solutions of fixed point equations:

fact = λn. Cond (Zero? n) 1 (Mul n (fact (Sub n 1)))

Suppose

H = λf.λn.Cond (Zero? n) 1 (Mul n (f (Sub n 1)))

then

fact = H fact

fact is a *fixed point* of function H!

Fixed Point Equations

$$f : D \rightarrow D$$

A fixed point equation has the form

$$f(x) = x$$

Its solutions are called the *fixed points* of f because if x_p is a solution then

$$x_p = f(x_p) = f(f(x_p)) = f(f(f(x_p))) = \ldots$$

We want to consider fixed-point equations whose solutions are functions, i.e., sets that contain their function spaces

domain theory, Scottary, ...

An example

Consider

f n = *if* n=0 *then* 1
else (*if* n=1 *then* f 3 *else* f (n-2))

$H = \lambda f.\lambda n.\mathrm{Cond}(n=0 , 1, \mathrm{Cond}(n=1, f\ 3, f\ (n-2))$

Is there an f_p such that $f_p = H\ f_p$?

f1 n	= 1	if n is even
	= ⊥	otherwise

f2 n	= 1	if n is even
	= 5	otherwise

f1 contains no arbitrary information and is said to be the least fixed point (lfp)

Under the assumption of *monotonicity* and *continuity* least fixed points are unique and computable

Computing a Fixed Point

- Recursion requires repeated application of a function
- Self application allows us to recreate the original term
  - Consider: $\Omega \;=\; (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$
  - Notice β-reduction of Ω leaves Ω : $\Omega \rightarrow \Omega$
- Now to get $F\ (F\ (F\ (F\ \ldots)))$ we insert F in Ω:

$$\Omega_F \;=\; (\lambda x.F\ (x\ x))\ (\lambda x.F\ (x\ x))$$

which β-reduces to:

$$\Omega_F \rightarrow F(\lambda x.\ F(x\ x))(\lambda x.\ F(x\ x))$$

$$\rightarrow F\ \Omega_F \rightarrow F(F\ \Omega_F) \rightarrow F(F(F\ \Omega_F)) \rightarrow \ldots$$

- Now λ –abstract F to get a Fix-Point Combinator:

$$\boxed{Y \;\equiv\; \lambda f.(\lambda x.\ (f\ (x\ x)))\ (\lambda x.(f\ (x\ x)))}$$

Y : A Fixed Point Operator

$$\boxed{Y \equiv \lambda f.(\lambda x.\ (f\ (x\ x)))\ (\lambda x.(f\ (x\ x)))}$$

Notice

$$\begin{aligned} Y\ F &\to (\lambda x.F\ (x\ x))\ (\lambda x.F\ (x\ x)) \\ &\to F\ (\lambda x.F\ (x\ x))\ (\lambda x.F\ (x\ x)) \\ &\to F\ (Y\ F) \end{aligned}$$

$F\ (Y\ F) = Y\ F$ (Y F) is a fixed point of F

Y computes the least fixed point of any function !

There are many different fixed point operators.

Mutual Recursion

```
odd  n = if n==0 then False else even (n-1)
even n = if n==0 then True  else odd  (n-1)
```

odd = H_1 even
even = H_2 odd
where
$H_1 = \lambda f.\lambda n.\mathrm{Cond}(n=0, \mathrm{False}, f(n-1))$
$H_2 = \lambda f.\lambda n.\mathrm{Cond}(n=0, \mathrm{True}, f(n-1))$

substituting "H_2 odd" for even
odd = H_1 (H_2 odd)
= H odd *where* $H = \lambda f.\ H_1\ (H_2\ f)$
⇒ odd = Y H

Can we express odd using Y ?

Self-application and Paradoxes

Self application, i.e., (x x) is dangerous.

Suppose:

$u \equiv \lambda y.\ \textit{if}\ (y\ y) = a\ \textit{then}\ b\ \textit{else}\ a$

What is (u u) ?

$(u\ u) \rightarrow \textit{if}\ (u\ u) = a\ \textit{then}\ b\ \textit{else}\ a$

Contradiction!!!

Any semantics of λ-calculus has to make sure that functions such as u have the meaning ⊥, i.e. “totally undefined” or “no information”.

Self application also violates *every* type discipline.

λ-calculus with Combinator Y

Recursive programs can be translated into the λ-calculus with constants and combinator Y. However,

- Y violates every type discipline
- translation is messy in case of mutually recursive functions

$\Rightarrow$

extend the λ-calculus with *recursive let blocks.*

The λ_{let} Calculus

6.820 Fundamentals of Program Analysis
Fall 2015