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 \( \lambda x. \lambda y. y \not= \lambda x. \lambda y. x \, y \not= \lambda x. \lambda y. x \, (x \, y) \)

- 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 \( \lambda \)-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

\[(\lambda q.\lambda p.p (q \ a)) (\lambda z.z)\]

\[\rightarrow \lambda p.p ((\lambda z.z) \ a))\]

\[\rightarrow \lambda p.p \ a\]

\(\beta\)-reductions monotonically increase information

The meaning of a term is the maximum information that can be obtained by \(\beta\)-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 \( \Omega = (\lambda x.x \ x) (\lambda x.x \ x) \)
  - It doesn’t have a normal form, and its meaning is ⊥
  - What about \( \lambda x.x \ \Omega \)?
  - It doesn’t have a normal form, but its meaning is \( \lambda x.x \ \perp \)

- What if a term has more than one normal form?
  - It would have two meanings. BAD
  - Not possible in the \( \lambda \)-calculus because of Confluence ...
Can the choice of redexes lead to different meaning?

• A term may have multiple redexes
  1. \(((\lambda x.M) A) ((\lambda x.N) B)\)
     \[
     \begin{array}{c}
     \text{-----} \\
     \rho_1 \text{-----} \\
     \rho_2 \text{-----}
     \end{array}
     \]
  2. \(((\lambda x.M) ((\lambda y.N)B))\)
     \[
     \begin{array}{c}
     \text{-----} \\
     \rho_2 \text{-----}
     \end{array}
     \]
     
     \[
     \begin{array}{c}
     \text{--------} \\
     \rho_1 \text{--------}
     \end{array}
     \]

• \(\rho_1\) followed by \(\rho_2\) does not necessary produce the same term as \(\rho_2\) followed by \(\rho_1\)
  – Notice in the second example \(\rho_1\) can destroy or duplicate \(\rho_2\).

• Can our choice of redexes lead us to produce terms that are clearly different (e.g., \(x\) versus \(\lambda y.y\))?
Church-Rosser Property

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

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 $\lambda$-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 $\lambda$-calculus is a program to reduce $\lambda$-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
  - ($\lambda x.E$) is in HNF if E is in HNF
  - (x E₁ ... Eₙ) 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
  - ($\lambda x.E$) is in WHNF
  - (x E₁ ... Eₙ) is in WHNF

  Practically most interesting ⇒ “Printable Answers”
Two Common Reduction Strategies

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

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

\[(\lambda x. y) ( ((\lambda x. x) x) (\lambda x. x)) \]

\[\rho_1 \quad \rho_2\]

*applicative order*

*normal order*
Computing a normal form

1. Every \( \lambda \)-expression does not have an answer \( i.e. \), a NF or HNF or WHNF

\[
(\lambda x.x \ x) \ (\lambda x.x \ x) = \Omega \\
\Omega \to \Omega \to \Omega \to \ldots
\]

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

\[
(\lambda x.\lambda y.y) \ \Omega
\]

leftmost redex: \( (\lambda x.\lambda y.y) \ \Omega \to \lambda y.y \)
innermost redex: \( (\lambda x.\lambda y.y) \ \Omega \to (\lambda x.\lambda y.y) \ \Omega \to \ldots \)
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 $\lambda$-calculus.
A Call-by-name Interpreter

**Answers:** WHNF

**Strategy:** leftmost redex

**cn(E):** Definition by cases on E

\[ E = x \mid \lambda x.E \mid E \ E \]

- \[ cn([[x]]) = x \]
- \[ cn([[\lambda x.E]]) = \lambda x.E \]
- \[ cn([[E_1 \ E_2]]) = \text{let } f = \text{cn}(E_1) \text{ in case } f \text{ of } \lambda x.E_3 = \text{cn}(E_3[E_2/x]) - f \ E_2 \]

Apply the function before evaluating the arguments

\[ [... \text{ represents syntax } ...] \text{ is my ppt approx} \]

Meta syntax
A Call-by-value Interpreter

Answers: WHNF

Strategy: rightmost-innermost redex but not inside a \( \lambda \)-abstraction

Evaluate the argument before applying the function

\( cv(E) \): Definition by cases on \( E \)
\( E = x \mid \lambda x.E \mid E \ E \)

\[
\begin{align*}
  cv([[x]]) & = x \\
  cv([[\lambda x.E]]) & = \lambda x.E \\
  cv([[E_1 \ E_2]]) & = let \ f = cv(E_1) \\
 & \quad a = cv(E_2) \\
 & in \ case \ f \ of \\
 & \lambda x.E_3 = cv(E_3[a/x]) \\
 & \_ \_ = f \ a
\end{align*}
\]
Coding this in Haskell
with
Algebraic Data Types
Algebraic types

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

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

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

```haskell
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
```

September 20, 2011
Constructors are functions

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

```haskell
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

```haskell
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
  - 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

\[
\begin{align*}
\langle c_1 \rangle \rightarrow r_1 & \quad \langle c_2 \rangle \rightarrow r_2 & \quad \ldots & \quad \langle c_k \rangle \rightarrow r_k \\
\hline
\langle configuration \rangle \rightarrow result
\end{align*}
\]
What evaluation order is this?

Call by Name

Do these semantics coincide?
What evaluation order is this?

Call by Value

\[
x \rightarrow x
\]

\[
\lambda x. e \rightarrow \lambda x. e
\]

\[
e_1 \rightarrow \lambda x. e'_1 \quad e_2 \rightarrow e'_2 \quad e'_1[\alpha(e'_2)/x] \rightarrow e_3
\]

\[
e_1 e_2 \rightarrow e_3
\]

- \(cv([[x]]) = x\)
- \(cv([[\lambda x. E]]) = \lambda x. E\)
- \(cv([[E_1 \ E_2]]) = \text{let } f = cv(E_1)\)
  \hspace{1cm} a = cv(E_2)\)
  \hspace{1cm} \text{in case } f \text{ of}\)
  \hspace{1cm} \lambda x. E_3 = cv(E_3[a/x])\)
  \hspace{1cm} _ = f a\)
Recursion and the Y Combinator
Recursion and Fixed Point Equations

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

\[ \text{fact} = \lambda n. \text{Cond} \ (\text{Zero?} \ n) \ 1 \ (\text{Mul} \ n \ (\text{fact} \ (\text{Sub} \ n \ 1))) \]

Suppose

\[ H = \lambda f. \lambda n. \text{Cond} \ (\text{Zero?} \ n) \ 1 \ (\text{Mul} \ n \ (f \ (\text{Sub} \ n \ 1))) \]

then

\[ \text{fact} = H \ \text{fact} \]

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

\[ f(n) = \begin{cases} 1 & \text{if } n = 0 \\ \text{else} & \begin{cases} f(3) & \text{if } n = 1 \\ f(n-2) & \text{else} \end{cases} \end{cases} \]

\[ H = \lambda f.\lambda n.\text{Cond}(n=0, 1, \text{Cond}(n=1, f 3, f(n-2))) \]

Is there an \( f_p \) such that \( f_p = H f_p \)?

\[
\begin{array}{ll}
\text{f1} \ n & = 1 & \text{if } n \text{ is even} \\
& = \bot & \text{otherwise}
\end{array}
\]

\[
\begin{array}{ll}
\text{f2} \ n & = 1 & \text{if } n \text{ is even} \\
& = 5 & \text{otherwise}
\end{array}
\]

\( f_1 \) contains no arbitrary information and is said to be the least fixed point (lfp)

Under the assumption of \textit{monotonicity} and \textit{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 \( \beta \)-reduction of \( \Omega \) leaves \( \Omega : \Omega \to \Omega \)
- Now to get \( F (F (F (F \ldots ))) \) we insert \( F \) in \( \Omega \):
  \[ \Omega_F = (\lambda x. F(x x)) (\lambda x. F(x x)) \]
  which \( \beta \)-reduces to:
  \[ \Omega_F \to F(\lambda x. F(x x))(\lambda x. F(x x)) \]
  \[ \to F \Omega_F \to F(F \Omega_F) \to F(F(F \Omega_F)) \to \ldots \]
- Now \( \lambda \)–abstract \( F \) to get a Fix-Point Combinator:
  \[ Y \equiv \lambda f. (\lambda x. (f (x x))) (\lambda x. (f (x x))) \]
**Y : A Fixed Point Operator**

\[ Y \equiv \lambda f. (\lambda x. (f (x x))) (\lambda x. (f (x x))) \]

Notice

\[ Y \, F \rightarrow (\lambda x. F (x x)) (\lambda x. F (x x)) \]
\[ \rightarrow F (\lambda x. F (x x)) (\lambda x. F (x x)) \]
\[ \rightarrow F (Y \, F) \]

\[ F (Y \, F) = Y \, F \]

(Y \, F) is a fixed point of \( F \)

Y computes the least fixed point of any function !

There are many different fixed point operators.

September 16, 2015
Mutual Recursion

odd \ n = \text{if } n == 0 \text{ then False else even (n-1)}
even \ n = \text{if } n == 0 \text{ then True else odd (n-1)}

\[
\begin{align*}
\text{odd} & = H_1 \text{ even} \\
\text{even} & = H_2 \text{ odd}
\end{align*}
\]

where
\[
\begin{align*}
H_1 & = \lambda f. \lambda n. \text{Cond}(n=0, \text{False}, f(n-1)) \\
H_2 & = \lambda f. \lambda n. \text{Cond}(n=0, \text{True}, f(n-1))
\end{align*}
\]

substituting “H2 odd” for even odd
\[
\begin{align*}
\text{odd} & = H_1 (H_2 \text{ odd}) \\
& = H \text{ odd} \text{ where } H = \lambda f. H_1 (H_2 f)
\end{align*}
\]

\[\Rightarrow \text{odd} = Y H\]

Can we express odd using Y?
Self-application and Paradoxes

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

Suppose:
\[
\begin{align*}
 u & \equiv \lambda y. \text{if } (y \ y) = a \text{ then } b \text{ else } a \\
\text{What is } (u \ u) \ ? \\
(u \ u) & \rightarrow \text{if } (u \ u) = a \text{ then } b \text{ else } a
\end{align*}
\]

\textit{Contradiction!!!}

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

Self application also violates \textit{every} type discipline.
Recursive programs can be translated into the \( \lambda \)-calculus with constants and combinator Y. However,

- Y violates every type discipline
- translation is messy in case of mutually recursive functions
  \[ \Rightarrow \]
  extend the \( \lambda \)-calculus with recursive let blocks.

The \( \lambda_{let} \) Calculus
6.820 Fundamentals of Program Analysis
Fall 2015

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.