A $\lambda$-calculus with Constants and Let-blocks

Arvind
Computer Science and Artificial Intelligence Laboratory
M.I.T.

September 15, 2010

Outline

• Big-step versus small-step semantics

• Recursion and the Y combinator

• The $\lambda_{let}$ Calculus
Big Step Semantics

• Consider the following deduction rule:

\[ \begin{align*}
E_1 & \vdash \lambda x. E_b \\
E_1 E_2 & \vdash E_b [E_2 / x]
\end{align*} \]

• Can we compute using this rule?
• In contrast to reduction rules such big step rules apply to the whole program and can subsume reduction strategy issues

Reduction rules aka small-step semantics

Big-step rules

• App1

\[ \begin{align*}
E_1 & \vdash \lambda x. E_b \\
E_1 E_2 & \vdash E_b [E_2 / x]
\end{align*} \]

• App2

\[ \begin{align*}
E_1 & \vdash x \\
E_1 E_2 & \vdash x E_2
\end{align*} \]

• App3

\[ \begin{align*}
E_1 & \vdash E_{11} E_{12} \\
E_1 E_2 & \vdash E_{11} E_{12} E_2
\end{align*} \]

• Abs

\[ \lambda x. E \vdash \lambda x. E \]

• Var

\[ x \vdash x \]
What can we compute using these Big-step rules?

- App1
  \[
  E_1 \vdash \lambda x.E_b \\
  E_1 E_2 \vdash E_b [E_2/ x]
  \]

- App2
  \[
  E_1 \vdash x \\
  E_1 E_2 \vdash x E_2
  \]

- App3
  \[
  E_1 \vdash E_{11} E_{12} \\
  E_1 E_2 \vdash E_{11} E_{12} E_2
  \]

- Abs
  \[
  \lambda x.E \vdash \lambda x.E
  \]

- Var
  \[
  x \vdash x
  \]

NF, HNF or WHNF?

Do these rules embody applicative or normal order reduction strategy?

Applicative-order Big-step rules

- App1
  \[
  E_1 \vdash \lambda x.E_b \\
  E_1 E_2 \vdash E_b [E_a/ x]
  \]

- App2
  \[
  E_1 \vdash x \\
  E_1 E_2 \vdash x E_a
  \]

- App3
  \[
  E_1 \vdash E_{11} E_{12} E_2 \vdash E_a \\
  E_1 E_2 \vdash E_{11} E_{12} E_a
  \]

- Abs
  \[
  \lambda x.E \vdash \lambda x.E
  \]

- Var
  \[
  x \vdash x
  \]

September 15, 2010  http://stellar.mit.edu/S/course/6/fa10/6.820

L03-5

cv([[x]]) = x
cv([[\lambda x.E]]) = \lambda x.E
\[
\begin{align*}
\text{cv}([[E_{11} E_{12}]]) &= \text{let } f = \text{cv}(E_{11}) \\
& \quad a = \text{cv}(E_{12}) \\
& \quad \text{in case } f \text{ of } \\
& \quad \lambda x.E_3 = \text{cv}(E_3[a/x]) \\
& \quad _\_ = f a
\end{align*}
\]
Outline

- Big-step versus small-step semantics
- Recursion and the Y combinator
- The $\lambda_{\text{let}}$ Calculus

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 \textit{fixed points} of \( f \) because if \( x_p \) is a solution then

\[ x_p = f(x_p) = f(f(x_p)) = f(f(f(x_p))) = ... \]

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

\textit{domain theory, Scottary, ...}

An example

Consider

\[ f n = \text{if } n=0 \text{ then } 1 \]
\[ \quad \text{else if } n=1 \text{ then } f 3 \text{ else } f (n-2) \]

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

Is there an \( f_p \) such that \( f_p = H f_p \)?
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 $\Omega : \Omega \rightarrow \Omega$

• Now to get $F (F (F (F ...)))$ we insert $F$ in $\Omega$:
  $\Omega_F = (\lambda x. F (x x)) (\lambda x. F (x x))$
  which $\beta$-reduces to:
  $$\Omega_F \rightarrow F(\lambda x. F(x x))(\lambda x. F(x x))$$
  $$\rightarrow F \Omega_F \rightarrow F(F \Omega_F) \rightarrow F(F(F \Omega_F)) \rightarrow ...$$
  • 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

$$Y F \rightarrow (\lambda x. F (x x))(\lambda x. F (x x))$$
Mutual Recursion

\[
\begin{align*}
\text{odd } n &= \text{if } n == 0 \text{ then False else even } (n-1) \\
\text{even } n &= \text{if } n == 0 \text{ then True else odd } (n-1)
\end{align*}
\]

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

Can we express odd using Y?

Self-application and Paradoxes

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

Suppose:

\[
u \equiv \lambda y. \text{if } (y y) = a \text{ then } b \text{ else } a
\]

What is \((u u)\)?
\(\lambda\)-calculus with Combinator Y

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 \(\lambda_{\text{let}}\) Calculus

Outline

- Big-step versus small-step semantics
- Recursion and the Y combinator
- The \(\lambda_{\text{let}}\) Calculus
\[\lambda\text{-calculus with Constants \\& Letrec}\]

\[E ::= x \mid \lambda x. E \mid E E \]
\[\mid \text{Cond } (E, E, E) \]
\[\mid \text{PF}_k(E_1, \ldots, E_k) \]
\[\mid \text{CN}_0 \]
\[\mid \text{CN}_k(E_1, \ldots, E_k) \mid \text{CN}_k(SE_1, \ldots, SE_k) \]
\[\mid \text{let } S \text{ in } E\]

\[\text{PF}_1 ::= \text{negate} \mid \text{not} \mid \ldots \mid \text{Prj}_1 \mid \text{Prj}_2 \mid \ldots\]
\[\text{PF}_2 ::= + \mid \ldots\]
\[\text{CN}_0 ::= \text{Number} \mid \text{Boolean}\]
\[\text{CN}_2 ::= \text{tuple2} \mid \text{cons} \mid \ldots\]

**Statements**

\[S ::= \varepsilon \mid x = E \mid S; S\]

*Variables on the LHS in a let expression must be pairwise distinct*

---

**Let-block Statements**

“;” is associative and commutative

\[S_1; S_2 \equiv S_2; S_1\]
\[S_1; (S_2; S_3) \equiv (S_1; S_2) ; S_3\]
\[\varepsilon; S \equiv S\]
\[let \varepsilon \text{ in } E \equiv E\]
Values and Simple Expressions

Values

\[ V ::= \lambda x.E \mid CN_0 \mid CN_k(SE_1, \ldots, SE_k) \]

Simple expressions

\[ SE ::= x \mid V \]

How to define the operational semantics of \( \lambda \text{let} \): Environments

\[ \text{Eval } [[e]] \rho \]

An environment-based interpreter.

- An environment where all the (variable name, value) bindings are kept and is passed around for expression evaluation
- When a let expression is encountered the environment is extended with all the let-bindings. Very complicated if the environment contains unevaluated expressions
- Not abstract enough – too many concrete data structures and associated functions for proper execution
How to define the operational semantics of $\lambda_{let}$: graphs

A let simply represents a wiring diagram, i.e., a graph

```
let
  f = \x.e₁
  y = e₂ e₃
in
(f y) + y
```

- Quite complicated to explain $\beta$-substitution in a graph based interpreter but good for showing sharing of terms

---

How to define the operational semantics of $\lambda_{let}$: via a calculus

- Rewrite rules
  - slightly more complicated than the $\lambda$-calculus
- Reduction Strategy
- Normal forms? Equivalences?
  - do the following terms have the same meaning

```
let x = 5
in  x
```

```
let x = 5
in  5
```

```
let x = 5
  y = 6
in  x
```

September 15, 2010  http://stellar.mit.edu/S/course/6/fa10/6.820
Operational semantics of $\lambda_{let}$: Issue #1

Creating redexes

$$((let \ S \ in \ \lambda x.e_1) \ e_2)$$

*How do we juxtapose* $$(\lambda x.e_1) \ e_2$$ ?

**Solution:** Lifting rules

Lifting Rules

$(let \ S' \ in \ e')$ is the $\alpha$-renamed $(let \ S \ in \ e)$ to avoid name conflicts in the following rules:

$$x = let \ S \ in \ e \quad \rightarrow \quad x = e'; \ S'$$

$$let \ S_1 \ in \ (let \ S \ in \ e) \quad \rightarrow \quad let \ S_1; \ S' \ in \ e'$$

$$(let \ S \ in \ e) \ e_1 \quad \rightarrow \quad let \ S' \ in \ e' \ e_1$$

$$\text{Cond}((let \ S \ in \ e), \ e_1, \ e_2) \quad \rightarrow \quad let \ S' \ in \ \text{Cond}(e', \ e_1, \ e_2)$$

$$\text{PF}_k(e_1, \ldots (let \ S \ in \ e), \ldots e_k) \quad \rightarrow \quad let \ S' \ in \ \text{PF}_k(e_1, \ldots e', \ldots e_k)$$
Operational semantics of $\lambda_{\text{let}}$:
Issue #2

How to refer to a variable binding

\[
\begin{align*}
\text{let} \\
f &= \lambda x. e_1 \\
y &= e_2 e_3 \\
in \\
(f \ y) + y
\end{align*}
\]

Solution: Instantiation rules

\[
((\lambda x. e_1) \ y) + y \ ?
\]

but first need to understand something about "contexts"

Contexts for Expressions

A context is an expression (or statement) with a “hole” such that if an expression is plugged in the hole the context becomes a legitimate expression:

\[
\begin{align*}
C[] &::= [ ] \\
& \mid \lambda x. C[] \\
& \mid C[] E \mid E C[] \\
& \mid \text{let } S \text{ in } C[] \\
& \mid \text{let } SC[] \text{ in } E
\end{align*}
\]

Statement Context for an expression

\[
\begin{align*}
SC[] &::= x = C[] \\
& \mid SC[] ; S \mid S ; SC[]
\end{align*}
\]
\( \lambda_{\text{let}} \) **Instantiation Rules**

A free variable in an expression can be instantiated by a *simple expression*

**Instantiation rule 1**
\[
(\text{let } x = a ; S \text{ in } C[x]) \to (\text{let } x = a ; S \text{ in } C'[a])
\]

| simple expression | free occurrence of \( x \) in some context \( C \) | renamed \( C[\ ] \) to avoid free-variable capture |

**Instantiation rule 2**
\[
(x = a ; S C[x]) \to (x = a ; S C'[a])
\]

**Instantiation rule 3**
\[
x = a \quad \text{→} \quad x = C'[C[x]]
\]

**where** \( a = C[x] \)

---

**The \( \beta \)-rule**

The normal \( \beta \)-rule
\[
(\lambda x.e) \ e_a \to e[e_a/x]
\]

is replaced the following \( \beta \)-rule
\[
(\lambda x.e) \ e_a \to \text{let } t = e_a \text{ in } e[t/x]
\]

where \( t \) is a new variable

*Instantiation rules* are used to refer to the value of variable \( t \)
Primitive Functions and Datastructures

\( \delta \)-rules
\[
\begin{align*}
+ & (n, m) & \rightarrow n + m \\
\ldots
\end{align*}
\]

Cond-rules
\[
\begin{align*}
\text{Cond}(\text{True}, e_1, e_2) & \rightarrow e_1 \\
\text{Cond}(\text{False}, e_1, e_2) & \rightarrow e_2
\end{align*}
\]

Data-structures
\[
\begin{align*}
\text{CN}_k(e_1, \ldots, e_k) & \rightarrow \\
& \text{let } t_1 = e_1; \ldots; t_k = e_k \\
& \text{in } \\
& \text{CN}_k(t_1, \ldots, t_k) \\
\text{Prj}_i(\text{CN}_k(a_1, \ldots, a_k)) & \rightarrow a_i
\end{align*}
\]

Strategy for computing WHNFs

- Conceptually just like normal-order reduction
- When we encounter a let-expression, we evaluate the term to be returned and instantiate variables in the term as necessary
  - exact specification can be given using a environment-based interpreter, graphs or big-step semantics (each has its advantages and disadvantages)

There are many semantics issues related to l-let and its relationship to l. We will discuss this later, Schedule permitting
Free Variables of an Expression

\[ \text{FV}(x) = \{x\} \]
\[ \text{FV}(E_1 E_2) = \text{FV}(E_1) \cup \text{FV}(E_2) \]
\[ \text{FV}(\lambda x.E) = \text{FV}(E) - \{x\} \]
\[ \text{FV(let S in E)} = \text{FVS}(S) \cup \text{FV}(E) - \text{BVS}(S) \]

\[ \text{FVS}(\varepsilon) = \{\} \]
\[ \text{FVS}(x = E; S) = \text{FV}(E) \cup \text{FVS}(S) \]

\[ \text{BVS}(\varepsilon) = \{\} \]
\[ \text{BVS}(x = E; S) = \{x\} \cup \text{BVS}(S) \]

\[ \alpha - \text{Renaming (to avoid free variable capture)} \]

Assuming \( t \) is a new variable, rename \( x \) to \( t \):
\[ \lambda x.e \equiv \lambda t.(e[t/x]) \]
\[ \text{let } x = e; S \text{ in } e_0 \equiv \text{let } t = e[t/x]; S[t/x] \text{ in } e_0[t/x] \]

where \([t/x]\) is defined as follows:
\[
\begin{align*}
    x[t/x] &= t \\
y[t/x] &= y & \text{if } x \neq y \\
(E_1 E_2)[t/x] &= (E_1[t/x] E_2[t/x]) \\
(\lambda x.E)[t/x] &= \lambda x.E \\
(\lambda y.E)[t/x] &= \lambda y.E[t/x] & \text{if } x \neq y \\
(\text{let } S \text{ in } E)[t/x] &= (\text{let } S \text{ in } E) & \text{if } x \notin \text{FV(let } S \text{ in } E) \\
&= (\text{let } S[t/x] \text{ in } E[t/x]) & \text{if } x \in \text{FV(let } S \text{ in } E) \\
(S_1; S_2)[t/x] &= (S_1[t/x]; S_2[t/x]) \\
(y = E)[t/x] &= (y = E[t/x]) \\
\varepsilon[t/x] &= \varepsilon
\end{align*}
\]