\( \lambda_S \): A Lambda Calculus with Side-effects

Arvind

delivered by Jacob Schwartz
Laboratory for Computer Science
M.I.T.

Lecture 14

http://www.csg.lcs.mit.edu/6.827

M-Structures and Barriers

- Some problems cannot be expressed functionally
  - Input / Output
  - Gensym: Generate unique identifiers
  - Gathering statistics
  - Graph algorithms
  - Non-deterministic algorithms

- Once side-effects are introduced, barriers are needed to control the execution of some operations

- The \( \lambda_S \) calculus
  - \( \lambda_C \) + side-effects and barriers

http://www.csg.lcs.mit.edu/6.827
The $\lambda_B$ Calculus : $\lambda_C +$ Barriers

- Even adding barriers to a purely functional calculus (without side-effects) is significant
  - Observability of Termination

- Using $\lambda_B$ as a stepping stone to $\lambda_S$ allows us to analyze the semantic effects of barriers separate from side-effects, simplifying the analysis
  - $\lambda_S = \lambda_B +$ side-effects

Outline

- Background
- The $\lambda_C$ calculus: $\lambda +$ letrecs
- Observable values
- The $\lambda_B$ calculus: $\lambda_C +$ barriers
- Garbage collection
- The $\lambda_S$ calculus: $\lambda_B +$ side-effects
\[ \text{\(\lambda + \text{Let}: A \text{ way to model sharing}\)} \]

Instead of the normal \(\beta\)-rule

\[(\lambda x. e) \ e_a \Rightarrow e \ [e_a/x]\]

use the following \(\beta_{\text{let}}\) rule

\[(\lambda x. e) \ e_a \Rightarrow \{ \text{let} \ t = e_a \ \text{in} \ e[t/x] \}\]

where \(t\) is a new variable

and only allow the substitution of \textit{values} and \textit{variables} to preserve sharing

---

\[ \text{http://www.csg.lcs.mit.edu/6.827} \]

---

\[ \text{Previous work on Sharing} \]

Differences are mainly regarding
- where variables can be instantiated
- the source language
  \[\lambda\]
  \[\lambda + \text{let}\]
  \[\lambda + \text{letrec}\]

- Graph reduction and lazy evaluation
  \textit{Wadsworth (71), Launchbury (POPL93)}
- Environments and Explicit Substitution
  \textit{Abadi, Cardelli, Curien & Levy (POPL 92, JFP)}
- Letrecs but no reductions inside \(\lambda\)-abstractions
  \textit{Ariola, Felleisen, Wadler, ... (POPL 95)}
- Letrecs
  \textit{Ariola et al. (96)}

\[ \text{http://www.csg.lcs.mit.edu/6.827} \]
\[ \lambda_C \text{ Syntax} \]

\[
E ::= \ x \mid \lambda x. E \mid E \ E \mid \{ S \ in \ E \} \mid \text{Cond (E, E, E)} \mid PF_k(E_1,...,E_k) \mid CN_0 \mid CN_k(E_1,...,E_k) \mid CN_k(SE_1,...,SE_k)
\]

\[
PF_1 ::= \text{negate} \mid \text{not} \mid \text{...} \mid \text{Prj}_1 \mid \text{Prj}_2 \mid \text{...}
\]

\[
CN_0 ::= \text{Number} \mid \text{Boolean}
\]

\[
CN_2 ::= \text{Cons} \mid \text{...}
\]

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

\[ \lambda_C \text{ Syntax} \]

\[ Values \]

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

\[ Simple \ expressions \]

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

http://www.csg.lcs.mit.edu/6.827
Equivalence Rules

- **α-renaming**
  \[ \lambda x. e \equiv \lambda x'. (e[x'/ x]) \]
  \[ \{ x=e ; S \ in \ e_0 \} \equiv \{ x'=e ; S \ in \ e_0 \}[x'/x] \]

- **Properties of " ; "**
  \[ \epsilon ; S \equiv S \]
  \[ S_1 ; S_2 \equiv S_2 ; S_1 \]
  \[ S_1 ; (S_2 ; S_3) \equiv (S_1 ; S_2 ) ; S_3 \]

\[ \lambda \text{let Instantiation Rules} \]

\( \lambda \) is a Simple Expression;
\( [x] \) is a free occurrence of \( x \) in \( C[x] \) or \( SC[x] \)

- **Instantiation Rule 1**
  \[ \{ x = a ; S \ in \ C[x] \} \Rightarrow \{ x = a ; S \ in \ C'[a] \} \]

- **Instantiation Rule 2**
  \[ (x = a ; SC[x]) \Rightarrow (x = a ; SC'[a]) \]

- **Instantiation Rule 3**
  \[ x = C[x] \Rightarrow x = C'[C[x]] \]
  where \( C[x] \) is simple

http://www.csg.lcs.mit.edu/6.827
\( \lambda_C \) Rules

- **Cond-rules**
  
  \[
  \text{Cond (True, } e_1, e_2) \Rightarrow e_1 \\
  \text{Cond (False, } e_1, e_2) \Rightarrow e_2
  \]

- **Constructors**
  
  \[
  \text{CN}_k(e_1, \ldots, e_k) \Rightarrow \{ t_1 = e_1 ; \ldots ; t_k = e_k \text{ in } \text{CN}_k(t_1, \ldots, t_k) \}
  \]

- **\( \delta \)-rules**
  
  \[
  \text{PF}_k(v_1, \ldots, v_k) \Rightarrow \text{pf}_k(v_1, \ldots, v_k) \\
  \text{Pr}_{ji}(CN_k(x_1, \ldots, x_i, \ldots, x_k)) \Rightarrow x_i
  \]

---

Need for Lifting Rules

\[
\{ f = \{ S_1 \text{ in } \lambda x.e_1 \}; \ \\
y = f a ; \ \\
in \ \\
(\{ S_2 \text{ in } \lambda x.e_2 \} e_3 ) \}
\]

*How do we juxtapose*

\[\text{(} \lambda x.e_1 \text{)} a\]

or

\[\text{(} \lambda x.e_2 \text{)} e_3\]

?
\[\lambda_c\] Block Flattening and Lifting Rules

- **Block Flatten**
  \[x = \{ S \text{ in } e \}\Rightarrow (x = e'; S')\]

- **Lifting rules**
  \[\{ S_1 \text{ in } \{ S_2 \text{ in } e \}\}\Rightarrow \{ S_1; S_2' \text{ in } e' \}\]
  \[\{ S \text{ in } e \}\Rightarrow \{ S' \text{ in } e' e_2 \}\]
  \[\text{Cond}(\{ S \text{ in } e \}, e_1, e_2) \Rightarrow \{ S' \text{ in } \text{Cond} (e', e_1, e_2) \}\]
  \[\text{PF}_k(e_1, \ldots \{ S \text{ in } e \}, \ldots e_k) \Rightarrow \{ S' \text{ in } \text{PF}_k(e_1, \ldots e', \ldots e_k) \}\]

\{ S' \text{ in } e' \} is the \(\alpha\)-renaming of \{ S \text{ in } e \} to avoid name conflicts

Non-confluence

\[
\text{odd} = \lambda n.\text{Cond}(n=0, \text{False}, \text{even (n-1)}) \quad ---- (M)
\]
\[
\text{even} = \lambda n.\text{Cond}(n=0, \text{True}, \text{odd (n-1)})
\]

**substitute for even (n-1) in M**
\[
\text{odd} = \lambda n.\text{Cond}(n=0, \text{False},
\text{Cond(n-1 = 0, True, odd ((n-1)-1)))} \quad ---- (M_1)
\]
\[
\text{even} = \lambda n.\text{Cond}(n=0, \text{True}, \text{odd (n-1)})
\]

**substitute for odd (n-1) in M**
\[
\text{odd} = \lambda n.\text{Cond}(n=0, \text{False}, \text{even (n-1)}) \quad ---- (M_2)
\]
\[
\text{even} = \lambda n.\text{Cond}(n=0, \text{True},
\text{Cond(n-1 = 0, False, even ((n-1)-1)))}
\]

\(M_1\) and \(M_2\) cannot be reduced to the same expression!

Ariola & Klop (LICS 94)
Printable Values

Printable values are trees and can be infinite.

We will compute the printable value of a term in 2 steps:

Info: \[ E \rightarrow T_p \] (trees)
Print: \[ E \rightarrow \{ T_p \} \]

(downward closed sets of trees)

where

\[ T_p ::= \bot \mid \lambda x.E \mid CN_0 \mid CN_k(T_{p1},...,T_{pk}) \]

\[ \bot \leq t \quad (bottom) \]
\[ t \leq t \quad (reflexive) \]
\[ CN_k(v_1,...,v_i,...,v_k) \leq CN_k(v'_1,...,v'_i,...,v_k) \]
\[ if \ v_i \leq v'_i \]

Info Procedure

**Info Procedure**

**Info** : \[ E \rightarrow T_p \]

\[
\begin{align*}
\text{Info} \left[ \{ \text{S in } E \} \right] &= \text{Info}[E] \\
\text{Info} \left[ \lambda x.E \right] &= \lambda \\
\text{Info} \left[ CN_0 \right] &= CN_0 \\
\text{Info} \left[ CN_k(a_1,...,a_k) \right] &= CN_k(\text{Info}[a_1],...,\text{Info}[a_k]) \\
\text{Info} \left[ E \right] &= \Omega \\
\text{otherwise}
\end{align*}
\]

**Proposition** Reduction is monotonic wrt Info:

If \( e \rightarrow e_1 \) then \( \text{Info}[e] \leq \text{Info}[e_1] \).

**Proposition** Confluence wrt Info:

If \( e \rightarrow e_1 \) and \( e \rightarrow e_2 \) then

\( \exists e_3 \text{ s.t. } e_1 \rightarrow e_3 \) and \( \text{Info}[e_2] \leq \text{Info}[e_3] \).
**Print Procedure**

Print : E --> \{T_p\}

Print[e] = \{ i | i \leq \text{Info}[e_1] \text{ and } e \triangleright e_1 \}\n
\triangleright \text{ is simple instantiation:}

\text{let } x = v ; S \text{ in } C[x] \triangleright \text{ let } x = v ; S \text{ in } C[v] \n
Unwind the value as much as possible
Keep track of all the unw windings

Terms with infinite unw windings lead to infinite sets.

---

**Print*: Maximum Printable Info**

Print*[e] = \{ U_{i \geq 0} \text{Print[s_i]} | s \epsilon PRS(e) \}  

\text{where}

*Definition*: Reduction Sequence
RS(e) = \{ s | s_0 = e, s_{i-1} \rightarrow s_i, 0 < i < |s| \}  

*Definition*: Progressive Reduction Sequence
PRS(e) = \{ s | s \epsilon RS(e), and
\exists i \forall j > i . s_j \rightarrow t \Rightarrow \exists k . \text{Print}[t] \leq \text{Print}[s_k] \}  

*Proposition:*
if e \rightarrow e_1 then Print*[e] = Print*[e_1].
Print*[e] has precisely one element.

---

http://www.csg.lcs.mit.edu/6.827
\( \lambda_B \) Syntax

\[
E ::= \ x \mid \lambda x.E \mid E \ E \mid \{ S \ in \ E \} \\
\mid \text{Cond } (E, E, E) \\
\mid \text{PF}_k(E_1, \ldots, E_k) \\
\mid \text{CN}_0 \mid \text{CN}_k(E_1, \ldots, E_k) \mid \text{CN}_k(x_1, \ldots, x_k)
\]

\[
\text{PF}_1 ::= \text{negate} \mid \text{not} \mid \ldots \mid \text{Prj}_1 \mid \text{Prj}_2 \mid \ldots
\]

\[
\text{CN}_0 ::= \text{Number} \mid \text{Boolean}
\]

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

Not in initial expressions

Barriers

\[
\{ \ y = 1+7 \} \Rightarrow \{ \ y = 8 \} \Rightarrow \{ \ y = 8 \;
\mid z = 3 \} \Rightarrow \{ \ z = 3 \} \Rightarrow \{ \ z = 3 \}
\]

Barriers discharge when all the bindings in the pre-region terminate, i.e., all expressions become values.
Stability and Termination

Definition: Expression e is said to be stable if when e ->> e₁, Print[e] = Print[e₁]

In general, an expression cannot be tested for stability.

Terminated Terms

\[ E^T ::= \ V \ | \ \{H \ in \ SE\} \]
\[ H ::= \ x = V \ | \ H; H \]

Proposition: All terminated terms are stable.

Values and Heap Terms

Values

\[ V ::= \ \lambda x.E \ | \ CN_0 \ | \ CN_k(x_1,...,x_k) \]

Simple expressions

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

Terminated Terms

\[ E^T ::= \ V \ | \ \{H \ in \ SE\} \]
\[ H ::= \ x = V \ | \ H; H \]
Barrier Rules

- **Barrier discharge**
  \((\varepsilon >> S) \Rightarrow S\)

- **Barrier equivalence**
  \(((H ; S_1) >> S_2) = (H ; (S_1 >> S_2))\)

\((H >> S) \Rightarrow (H ; S)\) (derivable)

---

\(\lambda_C\) Versus \(\lambda_B\)

In \(\lambda_B\) termination of a term is observable. Thus,

\[5 \not\in \{x = \bot\ \text{in} \ 5\}\]

Consider the context:

\[
\begin{align*}
\{ \ (y & = [\_] \\
& >>>
\ z & = 3) \\
& \text{in} \\
& z \}
\end{align*}
\]

⇒ equality in \(\lambda_C\) does not imply equality in \(\lambda_B\)

However, barriers can only make a term less defined.
Properties of $\lambda_B$

**Proposition** Barriers are associative:

$$S_1 >>> (S_2 >>> S_3) = (S_1 >>> S_2) >>> S_3$$

in all contexts.

**Proposition** Barriers reduce results:

Every reduction in $C[S_1 >>> S_2]$ can be modeled by a reduction in $C[S_1 ; S_2]$.

**Proposition** Postregions can be postponed:

If $C_1[S_1 >>> S_2] ->> C_3[S_3 >>> S_4]$ where the barrier is the same in both terms, there is a $C_2$ such that:

$$C_1[S_1 >>> S_2] ->> C_2[S_3 >>> S_2] ->> C_3[S_3 >>> S_4]$$

Garbage Collection

A Garbage collection rule erases part of a term.

**Definition:**

A garbage collection rule, GC, is said to be correct if for all $e$, $\text{Print}^*(e) = \text{Print}^*(\text{GC}(e))$
$\lambda_B$ Garbage Collection Rule

**GC$_0$-rule**
$$\{S_G ; S \in e\} \Rightarrow \{S \in e\}$$
if for all $x$, $x \in (FV(e) \cup FV(S))$ then $x \notin BV(S_G)$

**GC$_v$-rule**
$$\{H ; S \in e\} \Rightarrow \{S \in e\}$$
if for all $x$, $x \in (FV(e) \cup FV(S))$ then $x \notin BV(H)$

While both GC$_0$ and GC$_v$ rules are correct for $\lambda_{let}$, only the GC$_v$-rule is correct for $\lambda_B$.

$\lambda_S$ Syntax

$$E ::= x \mid \lambda x.E \mid E.E \mid \{S \in E\}$$
$$\mid \text{Cond}(E,E,E)$$
$$\mid \text{PF}_k(E_1,\ldots,E_k)$$
$$\mid \text{CN}_0 \mid \text{CN}_k(E_1,\ldots,E_k)$$
$$\mid \text{allocate}()$$
$$\mid \text{allocate}$$

PF$_1 ::= \text{negate} \mid \text{not} \mid \ldots \mid \text{Prj}_1 \mid \text{Prj}_2 \mid \ldots \mid \text{ifetch} \mid \text{mfetch} \ldots$

CN$_0 ::= \text{Number} \mid \text{Boolean} \mid ()$

$$S ::= \varepsilon \mid x = E \mid S; S$$
$$\mid S >>> S$$
$$\mid \text{sstore}(E,E)$$
$$\mid \text{allocator} \mid \text{empty}(o_i) \mid \text{full}(o_i,E) \mid \text{error}(o_i)$$

Not in initial expressions
Values and Heap Terms

**Values**

\[ V ::= \lambda x. E | CN_0 | CN_k(x_1, ..., x_k) | o_i \]

**Simple expressions**

\[ SE ::= x | V \]

**Heap Terms**

\[ H ::= x = V | H; H | allocator \]
\[ \quad | empty(o_i) | full(o_i, V) \]

**Terminal Expressions**

\[ ET ::= V | let \ H \ in \ SE \]

Side-effect Rules

- **Allocation rule**
  
  \[(\text{allocator}; x = \text{allocate}()) \Rightarrow (\text{allocator}; x = o; \text{empty}(o))\]
  
  where \(o\) is a new object descriptor

- **Fetch and Take rules**
  
  \[(x = \text{ifetch}(o) ; \text{full}(o,v)) \Rightarrow (x = v ; \text{full}(o,v))\]
  \[(x = \text{mfetch}(o) ; \text{full}(o,v)) \Rightarrow (x = v ; \text{empty}(o))\]

- **Store rules**
  
  \[(\text{sstore}(o,v) ; \text{empty}(o)) \Rightarrow \text{full}(o,v)\]
  \[(\text{sstore}(o,v) ; \text{full}(o,v')) \Rightarrow (\text{error}(o); \text{full}(o,v'))\]

- **Lifting rules**
  
  \[\text{sstore}(\{ S \ in \ e \}, e_2) \Rightarrow (S ; \text{sstore}(e, e_2))\]
  \[\text{sstore}(e_1, \{S \ in \ e \}) \Rightarrow (S ; \text{sstore}(e_1, e))\]
Nondeterministic Choice

choose = \lambda x. \{ m = allocate();
  sstore(m, True);
  (y = mfetch(m)
    >>>
    sstore(m, False));
  (z = mfetch(m)
    >>>
    sstore(m,True))
  in z \}

choose 100  ?