λ_S : A Lambda Calculus with Side-effects

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

Lecture 14

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 λ_S calculus
  - λ_C + side-effects and barriers

The λ_B Calculus : λ_C + Barriers

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

- Using λ_B as a stepping stone to λ_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 λ_C calculus: λ + letrecs
- Observable values
- The λ_B calculus: λ_C + barriers
- Garbage collection
- The λ_S calculus: λ_B + side-effects

λ + Let : A way to model sharing

Instead of the normal β-rule

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

use the following β_{let} *rule*

$(\lambda x.e)\ e_a \Rightarrow \{\ \textit{let}\ t = e_a\ \textit{in}\ \ e[t/x]\ \}$
where t is a new variable

and only allow the substitution of *values* and *variables* to preserve sharing

Previous work on Sharing

Differences are mainly regarding
- where variables can be instantiated
- the source language

	λ
or	λ + let
or	λ + letrec

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

λ_C Syntax

$$
\begin{aligned}
E ::= \; & x \;|\; \lambda x.E \;|\; E\ E \;|\; \{\, S \textit{ in } E \,\} \\
& |\ \text{Cond}\ (E, E, E) \\
& |\ PF_k(E_1,\ldots,E_k) \\
& |\ CN_0 \;|\; CN_k(E_1,\ldots,E_k) \;|\; \underline{CN}_k(SE_1,\ldots,SE_k)
\end{aligned}
$$

PF_1 ::= negate | not | ... | Prj_1 | Prj_2 | ...

...

CN_0 ::= Number | Boolean

CN_2 ::= Cons | ...

$S ::= \varepsilon \;|\; x = E \;|\; S; S$

Not in initial expressions

λ_C Syntax

Values

$V ::= \lambda x.E \;|\; CN_0 \;|\; \underline{CN}_k(SE_1,\ldots,SE_k)$

Simple expressions

$SE ::= x \;|\; V$

Equivalence Rules

- *α-renaming*

  $\lambda x.e \equiv \lambda x'.(e[x'/x])$

  $\{x=e;\ S\ \mathit{in}\ e_0\} \equiv \{x'=e;\ S\ \mathit{in}\ e_0\}[x'/x]$

- *Properties of* " ; "

  $\varepsilon\ ;\ S \equiv S$

  $S_1\ ;\ S_2 \equiv S_2\ ;\ S_1$

  $S_1\ ;\ (S_2\ ;\ S_3) \equiv (S_1\ ;\ S_2)\ ;\ S_3$

λ_{let} Instantiation Rules

a is a Simple Expression;
[x] is a free occurrence of x in C[x] or SC[x]

- Instantiation Rule 1

  $\{\ x = a\ ;\ S\ \text{in}\ C[x]\ \} \Rightarrow \{\ x = a\ ;\ S\ \text{in}\ C'[a]\ \}$

- 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

λ_C Rules

- *Cond-rules*

  Cond (True, e_1, e_2) $\Rightarrow$ e_1
  Cond (False, e_1, e_2) $\Rightarrow$ e_2

- *Constructors*

  $CN_k(e_1,\ldots,e_k) \Rightarrow \{t_1 = e_1 ;\ldots; t_k = e_k$ *in* $\underline{CN}_k(t_1,\ldots,t_k)\}$

- *δ-rules*

  $PF_k(v_1,\ldots,v_k) \Rightarrow pf_k(v_1,\ldots,v_k)$
  $Prj_i(\underline{CN}_k(x_1,\ldots,x_i,\ldots,x_k)) \Rightarrow x_i$

Need for Lifting Rules

{f = { S_1 *in* $\lambda x.e_1$ };
y = f a ;
in
({ S_2 *in* $\lambda x.e_2$ } e_3) }

How do we juxtapose

$(\lambda x.e_1)$ a

or

$(\lambda x.e_2)$ e_3 ?

λ_C Block Flattening and Lifting Rules

- *Block Flatten*
  x = { S *in* e } $\Rightarrow$ (x = e′ ; S′)

- *Lifting rules*
  { S_1 *in* { S_2 *in* e } } $\Rightarrow$ { S_1; S'_2 *in* e′ }
  { S *in* e } e_2 $\Rightarrow$ { S′ *in* e′ e_2 }
  Cond({ S *in* e }, e_1, e_2) $\Rightarrow$ { S′ *in* Cond (e′, e_1, e_2) }
  PF_k(e_1,...{ S *in* e },...e_k) $\Rightarrow$ { S′ *in* PF_k(e_1,...e′,...e_k) }

{ S′ *in* e′ } is the α*-renaming* of { S *in* e } to avoid name conflicts

Non-confluence

odd = λn.Cond(n=0, False, even (n-1)) ---- (M)
even = λn.Cond(n=0, True, odd (n -1))

substitute for even (n-1) in M
odd = λn.Cond(n=0, False,
Cond(n-1 = 0, True, odd ((n-1)-1))) ---- (M_1)
even = λn.Cond(n=0, True, odd (n -1))

substitute for odd (n-1) in M
odd = λn.Cond(n=0, False, even (n-1)) ---- (M_2)
even = λn.Cond(n=0, True,
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: $E \dashrightarrow T_P$ (trees)
Print: $E \dashrightarrow \{T_P\}$
(downward closed sets of trees)

where

T_P ::= $\perp$ | λ | CN_0 | $CN_k(T_{P1},\ldots,T_{Pk})$

$\perp \leq t$ *(bottom)*
$t \leq t$ *(reflexive)*
$CN_k(v_1,\ldots,v_i,\ldots,v_k) \leq CN_k(v_1,\ldots,v'_i,\ldots,v_k)$
if $v_i \leq v'_i$

Info Procedure

Info : $E \dashrightarrow T_P$

Info [{ S *in* E }] = Info [E]
Info [λx.E] = λ
Info [CN_0] = CN_0
Info [$\underline{CN}_k(a_1,\ldots,a_k)$] = $CN_k(\text{Info}[a_1],\ldots,\text{Info}[a_k])$
Info [E] = Ω
otherwise

Proposition Reduction is monotonic wrt Info:
If $e \twoheadrightarrow e_1$ then $\text{Info}[e] \leq \text{Info}[e_1]$.

Proposition Confluence wrt Info:
If $e \twoheadrightarrow e_1$ and $e \twoheadrightarrow e_2$ then
$\exists\, e_3$ s.t. $e_1 \twoheadrightarrow e_3$ and $\text{Info}[e_2] \leq \text{Info}[e_3]$.

Print Procedure

Print : E --> $\{T_P\}$

Print[e] = { i | i ≤ Info[e_1] and e $\xrightarrow{i}$>> e_1}.

$\xrightarrow{i}$> is simple instantiation:

let x = v ; S *in* C[x] $\xrightarrow{i}$> *let* x = v ; S *in* C[v]

Unwind the value as much as possible
Keep track of all the unwindings

Terms with infinite unwindings lead to infinite sets.

Print*: Maximum Printable Info

Print*[e] = { U_i Print[s_i] | s ε PRS(e) }

where

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

Definition: Progressive Reduction Sequence
PRS(e) = { s | s ε RS(e), and
$\exists i\ \forall j > i$. s_j ->> t ⇒ $\exists k$. Print[t] ≤ Print[s_k] }

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

λ_B Syntax

E ::= x | λx.E | E E | { S *in* E }
| Cond (E, E, E)
| $PF_k(E_1,\ldots,E_k)$
| CN_0 | $CN_k(E_1,\ldots,E_k)$ | $\underline{CN}_k(x_1,\ldots,x_k)$

PF_1 ::= negate | not | ... | Prj_1| Prj_2 | ...
...
CN_0 ::= Number | Boolean

S ::= ε | x = E | S; S | S >>> S

Not in initial expressions

Barriers

```
{ ( y = 1+7        { ( y = 8         { y = 8 ;         { y = 8 ;
    >>>                >>>
    z = 3 )    ⇒       z = 3 )   ⇒     ( z = 3 )   ⇒     z = 3
in                 in                in                in
        z }                z }               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 ->> e_1, Print[e] = Print[e_1]

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 ::= λx.E | CN_0 | $\underline{CN}_k(x_1,...,x_k)$

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*

  $((H ; S_1) >>> S_2) \equiv (H ; (S_1 >>> S_2))$

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

λ_C Versus λ_B

In λ_B termination of a term is observable.
Thus,

$$5 \underset{?}{\neq} \{x = \perp \textit{ in } 5\}$$

Consider the context:

```
{ (y = [■]
   >>>
   z = 3)
in
  z }
```

$\Rightarrow$ equality in λ_C does not imply equality in λ_B

However, barriers can only make a term less defined.

Properties of λ_B

Proposition Barriers are associative:

S1 >>> (S2 >>> S3) = (S1 >>> S2) >>> S3

in all contexts.

Proposition Barriers reduce results:

Every reduction in C[S1 >>> S2] can be modeled by a reduction in C[S1 ; S2].

Proposition Postregions can be postponed:

If C1[S1 >>> S2] ->> C3[S3 >>> S4] where the barrier is the same in both terms, there is a C2 such that:

C1[S1 >>> S2] ->> C2[S3 >>> S2] ->> C3[S3 >>> S4]

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, Print*(e) = Print*(GC(e))

λ_B Garbage Collection Rule

GC_0-rule
$\{ S_G ; S \textit{ in } e \} \Rightarrow \{ S \textit{ in } e \}$
if forall x, $x \in (FV(e) \cup FV(S))$ then $x \notin BV(S_G)$

GC_v-rule
$\{ H ; S \textit{ in } e \} \Rightarrow \{ S \textit{ in } e \}$
if forall x, $x \in (FV(e) \cup FV(S))$ then $x \notin BV(H)$

While both GC_0 and GC_v rules are correct for λ_{let}, only the GC_v-rule is correct for λ_B.

λ_S Syntax

E ::= x | λx.E | E E | { S *in* E }
| Cond (E, E, E)
| $PF_k(E_1,\ldots,E_k)$
| CN_0 | $CN_k(E_1,\ldots,E_k)$ | $\underline{CN}_k(x_1,\ldots,x_k)$
| allocate()
| o_i object descriptors

PF_1 ::= negate | not | ... | Prj_1| Prj_2 | ... | ifetch | mfetch
...
CN_0 ::= Number | Boolean | ()

S ::= ε | x = E | S; S
| S >>> S
| sstore(E,E)
| allocator | empty(o_i) | full(o_i,E) | error(o_i)

Not in initial expressions

Values and Heap Terms

Values

V ::= λx.E | CN_0 | $\underline{CN}_k(x_1,...,x_k)$ | o_i

Simple expressions

SE ::= x | V

Heap Terms

H ::= x = V | H; H | allocator
| empty(o_i) | full(o_i,V)

Terminal Expressions

E^T ::= V | *let* H *in* SE

Side-effect Rules

- *Allocation rule*
  (allocator; x=allocate())$\Rightarrow$allocator; x = o; empty(o))
  where o is a new object descriptor
- *Fetch and Take rules*
  (x=ifetch(o) ; full(o,v)) $\Rightarrow$ (x=v ; full(o,v))
  (x=mfetch(o) ; full(o,v)) $\Rightarrow$ (x=v ; empty(o))
- *Store rules*
  (sstore(o,v) ; empty(o)) $\Rightarrow$ full(o,v)
  (sstore(o,v) ; full(o,v')) $\Rightarrow$ (error(o); full(o,v'))
- *Lifting rules*
  sstore({ S *in* e }, e_2) $\Rightarrow$ (S ; sstore(e,e_2))
  sstore(e_1, { S *in* e }) $\Rightarrow$ (S ; sstore(e_1,e))

Nondeterministic Choice

```
choose =    λx. { m = allocate();
                  sstore(m, True);
                  (  y = mfetch(m)
                     >>>
                     sstore(m, False) );
                  (  z = mfetch(m)
                     >>>
                     sstore(m,True)  )
                  in  z }

choose 100    ?
```