Abstract Machines for the Multi-return $\lambda$-calculus

Principles of Programming Languages (G711 ’05)
Northeastern University
13 December 2005

David Van Horn
<dvanhorn@cs.brandeis.edu>
http://www.cs.brandeis.edu/~dvanhorn/

Matthew Goldfield
<mvg@cs.brandeis.edu>
http://www.cs.brandeis.edu/~mvg/
Abstract Machines for the Multi-return $\lambda$-calculus

ignore
Abstract Machines for the Multi-return $\lambda$-calculus

Outline

- the original goal (Matt)
- the background (what is it all about)
- the chosen method to accomplish it
- a description of how you went about your task and what you accomplished (David)
- what you learned from this activity
- what you didn’t learn from this activity
- how you could refine/reformulate/enhance the first problem statement
Abstract Machines for the Multi-return $\lambda$-calculus

The Original Goal

Posit implementation strategies for this calculus, realized as a series of interpreters for the applicative order, call-by-value fragment of the calculus.

The interpreters range from a big-step operational semantics to a CK-style abstract machine.
Abstract Machines for the Multi-return $\lambda$-calculus

The Background (What it is all about)

- Multi-return Function Call, Shivers and Fisher (under review for JFP)
Abstract Machines for the Multi-return $\lambda$-calculus

The Multi-return $\lambda$-calculus

Abstractions \[ l ::= \lambda x.e \quad l \in \text{Lam} \]

Expressions \[ e ::= x \mid n \mid l \mid (e \ e) \mid \langle e \ r \ldots r \rangle \quad e \in \text{Exp} \]

Return points \[ r ::= l \mid \#i \quad r \in \text{RP} \]

Values \[ v ::= n \mid l \quad v \in \text{Val} \subseteq \text{Exp} \]
Abstract Machines for the Multi-return $\lambda$-calculus

The Multi-return $\lambda$-calculus

funapp $\frac{(\lambda x.e) \ e_1 \leadsto [x \mapsto e_1]e}{\text{retlam} \ \frac{\langle v \ l \rangle \leadsto (l \ v)}{\text{rpsel} \ \frac{\langle v \ r_1 \ldots r_m \rangle \leadsto \langle v \ r_1 \rangle}{m > 1}}$

retail $\frac{\langle\langle v \ #i\rangle r_1 \ldots r_m \rangle \leadsto \langle v \ r_i \rangle}{1 < i \leq m}$

funprog $\frac{e_0 \leadsto e'_0}{(e_0 \ e_1) \leadsto (e'_0 \ e_1)}$

argprog $\frac{e_1 \leadsto e'_1}{(e_0 \ e_1) \leadsto (e_0 \ e'_1)}$

retprog $\frac{e \leadsto e'}{\langle e \ r_1 \ldots r_m \rangle \leadsto \langle e' \ r_1 \ldots r_m \rangle}$

bodyprog $\frac{e \leadsto e'}{\lambda x.e \leadsto \lambda x.e'}$

rpprog $\frac{l \leadsto l'}{\langle e \ r_1 \ldots l \ldots r_m \rangle \leadsto \langle e \ r_1 \ldots l' \ldots r_m \rangle}$
The Multi-return $\lambda$-calculus (CbV fragment)

funapp $\langle \lambda x.e \rangle v_1 \rightsquigarrow [x \mapsto v_1]e$

retlam $\langle v \; l \rangle \rightsquigarrow (l \; v)$

rpsel $\langle v \; r_1 \ldots r_m \rangle \rightsquigarrow \langle v \; r_1 \rangle$

$\quad m > 1$

ret1 $\langle v \; \#1 \rangle \rightsquigarrow v$

retail $\langle \langle v \; \#i \rangle \; r_1 \ldots r_m \rangle \rightsquigarrow \langle v \; r_i \rangle$

$\quad 1 < i \leq m$

funprog $e_0 \rightsquigarrow e'_0$

$(e_0 \; e_1) \rightsquigarrow (e'_0 \; e_1)$

argprog $e_1 \rightsquigarrow e'_1$

$(v_0 \; e_1) \rightsquigarrow (v_0 \; e'_1)$

retprog $e \rightsquigarrow e'$

$\langle e \; r_1 \ldots r_m \rangle \rightsquigarrow \langle e' \; r_1 \ldots r_m \rangle$

bodyprog $\lambda x.e \rightsquigarrow \lambda x.e'$

rpprog $l \rightsquigarrow l'$

$\langle e \; r_1 \ldots l \ldots r_m \rangle \rightsquigarrow \langle e \; r_1 \ldots l' \ldots r_m \rangle$
Abstract Machines for the Multi-return λ-calculus

The Chosen Method

- *Refocusing in Reduction Semantics*, Danvy and Nielsen
- *A Syntactic Correspondence between Context-Sensitive Calculi and Abstract Machines*, Biernacka and Danvy
- *Programming Languages and Lambda Calculi*, Felleisen and Flatt
Abstract Machines for the Multi-return $\lambda$-calculus

Going About the Task

We take a derivational approach:

- We develop a *standard reduction relation* for the CbV $\lambda_{MR}$
- Give a reduction-based interpreter
- Then *refocus* based on Danvy
- Then derive a CK-style machine
- Refunctionalize to obtain CPS semantics
- Direct-style transformation to get a big-step operational semantics
Abstract Machines for the Multi-return $\lambda$-calculus

The Doggie-bag: What to take home

I want people to come away with at least a cursory familiarity of the tools we employ in the derivational approach:

- Standard reduction
- Refocusing
- defunctionalization and refunction-alization
- CPS and direct-style

And not the details of the machines we produce, or the $\lambda_{MR}$-calculus.
Abstract Machines for the Multi-return λ-calculus

Contributions (The Danvy hammer hits this thumb)

- Standard reduction relation
- Reduction-based evaluator
- Refocused evaluator
- Pre-abstract machine
- Eval/Apply machine
- Eval/Apply in defunctionalized form
- CPS semantics
- Big-step operational semantics
Correspondences (The Danvy hammer hits this thumb)

- Standard reduction relation $\iff \rightsquigarrow_v \iff$
- Reduction-based evaluator $\iff$
- Refocused evaluator $\iff$
- Pre-abstract machine $\iff$
- Eval/Apply machine $\iff$
- Eval/Apply in defunctionalized form $\iff$
- CPS semantics $\iff$
- Big-step operational semantics
Standard reduction

Standard reduction employs an explicit representation of a term’s context.

Evaluation is defined as the transitive closure of single reductions consisting of:

1. decomposing a term into a context and a potential redex
2. contracting the redex
3. plugging the contractum into the context

If steps 1 or 2 fail, the program is stuck. For evaluation to be deterministic, decomposition must be unique.
Abstract Machines for the Multi-return $\lambda$-calculus

Standard approach to Standard reduction

Grammar of reduction contexts $C$ given by *progress* rules. Place hole in place of term making progress.

Standard reduction relation $\rightsquigarrow$ given by *redex* rules where redex is in the hole of a context.

For example:

\[
\begin{align*}
\text{argprog} & : \quad e \rightsquigarrow e' \\
 & \Rightarrow \quad (v \; e) \rightsquigarrow (v \; e') \\
& \Rightarrow \quad C ::= \ldots \mid C[(v \; [\; ])]
\end{align*}
\]

\[
\begin{align*}
\text{funapp} & : \quad ((\lambda x. e) \; v) \rightsquigarrow [x \mapsto v]e \\
& \Rightarrow \quad C[((\lambda x. e) \; v)] \mapsto C[[x \mapsto v]e]
\end{align*}
\]
Abstract Machines for the Multi-return $\lambda$-calculus

The Multi-return $\lambda$-calculus (CbV fragment)

\[
\begin{align*}
\text{funapp}_v & : (\lambda x.e) v \leadsto_v [x \mapsto v]e \\
\text{retlam} & : \langle v \; l \rangle \leadsto_v (l \; v) \\
\text{rpsel} & : \langle v \; r_1 \ldots r_m \rangle \leadsto_v \langle v \; r_1 \rangle \quad m > 1 \\
\text{ret1} & : \langle v \; \#1 \rangle \leadsto_v v \\
\text{rettail} & : \langle\langle v \; \#i\rangle \; r_1 \ldots r_m \rangle \leadsto_v \langle v \; r_i \rangle \quad 1 < i \leq m \\
\text{funprog} & : e_0 \leadsto_v e'_0 \\
& \quad (e_0 \; e_1) \leadsto_v (e'_0 \; e_1) \\
\text{argprog}_v & : e \leadsto_v e' \\
& \quad (v \; e) \leadsto_v (v \; e') \\
\text{retprog} & : e \leadsto_v e' \\
& \quad \langle e \; r_1 \ldots r_m \rangle \leadsto_v \langle e' \; r_1 \ldots r_m \rangle
\end{align*}
\]
Standard approach to Standard reduction

Reduction contexts and potential redexes:

\[
C ::= \begin{array}{l}
[ ] \\
C[([e][])] \text{ argprog} \\
C[[[v]]] \text{ funprog}_v \\
C[<[v] r \ldots r>] \text{ retprog}
\end{array}
\]

\[
p ::= (v \ v) \\
\begin{array}{l}
<[v \ r \ldots r]> \\
<[<[v \ r \ldots r]> r \ldots r]>
\end{array}
\]
Problem: Grammatical ambiguity

Unique decomposition does not hold.

\[
\text{decompose}(\langle v \ r_1 \rangle \ r_1') = [\ ], \ \langle v \ r_1 \rangle \ r_1'
\]

\[
\text{decompose}(\langle v \ r_1 \rangle \ r_1') = [\langle \ ] \ r_1', \ \langle v \ r_1 \rangle
\]
Fix 1: Tighter characterization of potential redexes

We can refine the grammar of potential redexes starting with the observation that \( \langle v \ #i \rangle, i > 1 \) is never a redex, and therefore not a potential redex.

(due to Matthias)
Fix 2: Context-sensitive standard reduction

We can simplify the grammar of potential redexes:

\[
p ::= (v v) \\
    | \langle v \ r \ldots \ r \rangle \\
    | \langle \langle v \ r \ldots \ r \rangle \ r \ldots \ r \rangle \\
\]

⇒

\[
p ::= (v v) \\
    | \langle v \ r \ldots \ r \rangle \\
\]

And make contraction context-sensitive, i.e. contracting a redex depends upon the context in which it appears.
Fix 2: Context-sensitive standard reduction

Reduction contexts \( C \) ::= [ ]
| \( C[(e \ [ \ ])] \)
| \( C[[\ ] v]) \)
| \( C[\langle[\ ] r\ldots r\rangle] \)

Potential redexes \( p \) ::= (\( v \ v \))
| \( \langle v \ r\ldots r\rangle \)

\[
\begin{align*}
C[(\lambda x.e) v] & \longrightarrow C[[x \mapsto v]e] \\
C[\langle v \ r_1\ldots r_m\rangle] & \longrightarrow C[\langle v \ r_1\rangle] \quad m > 1 \\
C[\langle v \ l\rangle] & \longrightarrow C[(l \ v)] \\
C[\langle v \ #1\rangle] & \longrightarrow C[v] \\
(C[C[\langle[\ ] r_1\ldots r_m\rangle][\langle v \ #i\rangle]] & \longrightarrow C[\langle v \ r_i\rangle] \quad 1 < i \leq m
\end{align*}
\]
Fix 2: Context-sensitive standard standard reduction

Reduction contexts \( C \) ::= [ ]
| \( C[(e [ ])] \)
| \( C[[ ] v] \)
| \( C[\langle[ ] r...r\rangle] \)

Potential redexes \( p \) ::= (v v)
| \( \langle v r...r\rangle \)

\[
\begin{align*}
C[((\lambda x.e) v)] & \longrightarrow C[[x \mapsto v]e] \\
C[\langle v r_1...r_m\rangle] & \longrightarrow C[\langle v r_1\rangle] \quad m > 1 \\
C[\langle v l\rangle] & \longrightarrow C[[l v]] \\
C[\langle v \#1\rangle] & \longrightarrow C[v] \\
(C[\langle[ ] r_1...r_m\rangle][\langle v \#i\rangle] & \longrightarrow C[\langle v r_i\rangle] \quad 1 \leq i \leq m
\end{align*}
\]
Results

Lemma 1 (Unique decomposition) For any expression $e$, either $e \in \text{Val}$ or there exists a unique reduction context $C$ and potential redex $p$, such that $e = C[p]$.

Lemma 2 (Correspondence with $\lambda_{MR}$) $e \xrightarrow{v} e' \iff e \xrightarrow{} e'$.

These results are easy to prove and get us “off the ground” for producing interpreters that correspond with the original CbV fragment of $\lambda_{MR}$.
Reduction-based evaluation

We can now define our first interpreter based on the rules we saw before, modified to be context-sensitive.

Evaluation is defined as the transitive closure of single reductions consisting of:

1. decomposing a term into a context and a potential redex
2. contracting the redex, together with its context
3. plugging the contractum into the potentially modified context
Abstract Machines for the Multi-return λ-calculus

Reduction-based evaluation

evaluate : Exp → Val + (RedCont × StuckRedex)
evaluate(e) = iterate(decompose(e))

iterate : Val + (RedCont × PotRedex) → Val + (RedCont × StuckRedex)
iterate(v) = v
iterate(C, ((λx.e) v)) = evaluate(plug([x → v]e, C))
iterate(C, <v l>) = evaluate(plug((l v), C'))
iterate(C, <v #1>) = evaluate(plug(v, C'))
iterate(C, <v r1...rm>) = evaluate(plug(<v r1>, C'))  m > 1
iterate(C'[<[ ] r1...rm>], <v #i>) = evaluate(plug(<v ri>, C'))  1 < i ≤ m
iterate(C'[<[ ] r1...rm>], <v #i>) = (C'[<[ ] r1...rm>], <v #i>)  i > m > 1
iterate(C', <v #i>) = (C', <v #i>)
                  C ≠ C'[<[ ] r1...rm>] and i > 1
iterate(C', (n v)) = (C', (n v))
Abstract Machines for the Multi-return $\lambda$-calculus

Refocused evaluation

Iterative decomposition is not efficient. So we rewrite the interpreter so that \emph{decompose} is always called on the result of \emph{plug}:

\[
evaluate(\plug(e, C)) \Rightarrow \iterate(\decompose(\plug(e, C)))
\]

\[
\decompose(e) \Rightarrow \decompose(\plug(e, []))
\]

The first transformation is obtained by inlining \emph{evaluate}, i.e. \emph{iterate(\decompose(e))}. The second is an obvious equivalence.

We rewrite the interpreter using \emph{refocus} = \emph{decompose} \circ \emph{plug} and are now free to use any function extensionally equivalent to \emph{refocus}.
Abstract Machines for the Multi-return λ-calculus

Pre-abstract machine

Danvy and Nielsen provide a construction for an efficient \textit{refocus} focus from the standard reduction specification.

By construction, it is extensionally equivalent to \textit{decompose} \text{circ} \textit{plug}.

The \textit{refocus} function is itself an abstract machine (state transition system). Evaluation then uses an abstract machine and a trampoline function computing its transitive closure.
Pre-abstract machine

The \textit{refocus} function is defined by cases on the grammar of expressions, \textit{refocus}_{aux} by cases on the top most context:

\begin{align*}
\text{refocus} : \text{Exp} \times \text{RedCont} & \rightarrow \text{Val} + (\text{RedCont} \times \text{StuckRedex}) \\
\text{refocus}(v, C) & = \text{refocus}_{aux}(C, v) \\
\text{refocus}((e_0 \ e_1), C) & = \text{refocus}(e_0, C[[ \ ] \ e_1]) \\
\text{refocus}(<e \ r_1 \ldots r_m>, C) & = \text{refocus}(e, C[<[ ] \ r_1 \ldots r_m>]) \\
\text{refocus}_{aux} : \text{RedCont} \times \text{Val} & \rightarrow \text{Val} + (\text{RedCont} \times \text{StuckRedex}) \\
\text{refocus}_{aux}([ \ ], v) & = v \\
\text{refocus}_{aux}(C[[ \ ] \ e], v) & = \text{refocus}(e, C \circ (v[ \ ])) \\
\text{refocus}_{aux}(C[[v' [ ]]], v) & = (C, (v' v)) \\
\text{refocus}_{aux}(C[<[ ] \ r_1 \ldots r_m>], v) & = (C, <v \ r_1 \ldots r_m>)
\end{align*}
Staged abstract machine

In the pre-abstract machine *iterate* is always called on the result of *refocus*.

We can rewrite *iterate* and *evaluate* to call *refocus* tail-recursively and rewrite *refocus* to call *iterate* on it’s result.

The result is a state-transition system, aka an *abstract machine*.

The machine transitions are partitioned into context transitions and redex transitions, hence it is *staged*.
Abstract Machines for the Multi-return $\lambda$-calculus

CK abstract machine

Inlining the iterate function gives a CK abstract machine.

CPS semantics

Refunctionalizing (Church-encoding the reduction context datatype) the CK machine gives a CPS semantics.

Big-step operational semantics

Direct-style transformation of the CPS semantics yields the big-step operational semantics.
Abstract Machines for the Multi-return λ-calculus

Acknowledgements

Olivier Danvy

Matthias Felleisen
Abstract Machines for the Multi-return λ-calculus

Acknowledgements

Olivier Danvy

Matthias Felleisen