Abstract Machines for the Multi-return λ-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/

ignore

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

The Original Goal

Posit implementation strategies for this calculus, realized as a series of a 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.

The Background (What it is all about)

- *Multi-return Function Call*, Shivers and Fisher (under review for JFP)

The Multi-return λ-calculus

Abstractions	$l ::= \lambda x.e$	$l \in \mathsf{Lam}$
Expressions	$e ::= x \mid n \mid l \mid (e\ e) \mid \triangleleft e\ r \ldots r \triangleright$	$e \in \mathsf{Exp}$
Return points	$r ::= l \mid \#i$	$r \in \mathsf{RP}$
Values	$v ::= n \mid l$	$v \in \mathsf{Val} \subseteq \mathsf{Exp}$

The Multi-return λ-calculus

$$\text{funapp}\ \frac{}{(\lambda x.e)\ e_1 \rightsquigarrow [x \mapsto e_1]e} \qquad \text{retlam}\ \frac{}{\lhd v\ \ l \rhd \rightsquigarrow (l\ \ v)}$$

$$\text{rpsel}\ \frac{}{\lhd v\ \ r_1 \ldots r_m \rhd \rightsquigarrow \lhd v\ \ r_1 \rhd}\ m > 1 \qquad \text{ret1}\ \frac{}{\lhd v\ \ \#1 \rhd \rightsquigarrow v}$$

$$\text{rettail}\ \frac{}{\lhd \lhd v\ \ \#i \rhd\, r_1 \ldots r_m \rhd \rightsquigarrow \lhd v\ \ r_i \rhd}\ 1 < i \leq m$$

$$\text{funprog}\ \frac{e_0 \rightsquigarrow e_0'}{(e_0\ \ e_1) \rightsquigarrow (e_0'\ \ e_1)} \qquad \text{argprog}\ \frac{e_1 \rightsquigarrow e_1'}{(e_0\ \ e_1) \rightsquigarrow (e_0\ \ e_1')}$$

$$\text{retprog}\ \frac{e \rightsquigarrow e'}{\lhd e\ \ r_1 \ldots r_m \rhd \rightsquigarrow \lhd e'\ \ r_1 \ldots r_m \rhd} \qquad \text{bodyprog}\ \frac{e \rightsquigarrow e'}{\lambda x.e \rightsquigarrow \lambda x.e'}$$

$$\text{rpprog}\ \frac{l \rightsquigarrow l'}{\lhd e\ \ r_1 \ldots l \ldots r_m \rhd \rightsquigarrow \lhd e\ \ r_1 \ldots l' \ldots r_m \rhd}$$

The Multi-return λ-calculus (CbV fragment)

$$\text{funapp}\ \frac{}{(\lambda x.e)\ v_1 \leadsto [x \mapsto v_1]e} \qquad \text{retlam}\ \frac{}{\lhd v\ l \rhd \leadsto (l\ v)}$$

$$\text{rpsel}\ \frac{}{\lhd v\ r_1 \ldots r_m \rhd \leadsto \lhd v\ r_1 \rhd}\ m > 1 \qquad \text{ret1}\ \frac{}{\lhd v\ \#1 \rhd \leadsto v}$$

$$\text{rettail}\ \frac{}{\lhd \lhd v\ \#i \rhd r_1 \ldots r_m \rhd \leadsto \lhd v\ r_i \rhd}\ 1 < i \leq m$$

$$\text{funprog}\ \frac{e_0 \leadsto e_0'}{(e_0\ e_1) \leadsto (e_0'\ e_1)} \qquad \text{argprog}\ \frac{e_1 \leadsto e_1'}{(v_0\ e_1) \leadsto (v_0\ e_1')}$$

$$\text{retprog}\ \frac{e \leadsto e'}{\lhd e\ r_1 \ldots r_m \rhd \leadsto \lhd e'\ r_1 \ldots r_m \rhd} \qquad \text{bodyprog}\ \frac{e \leadsto e'}{\lambda x.e \leadsto \lambda x.e'}$$

$$\text{rpprog}\ \frac{l \leadsto l'}{\lhd e\ r_1 \ldots l \ldots r_m \rhd \leadsto \lhd e\ r_1 \ldots l' \ldots r_m \rhd}$$

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

Going About the Task

We take a derivational approach:

- We develop a *standard reduction relation* for the CbV λ_{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

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 refunctionalization
- CPS and direct-style

And *not* the details of the machines we produce, or the λ_{MR}-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 $\Longleftrightarrow$ $\leadsto_v$ $\Longleftrightarrow$
- Reduction-based evaluator $\Longleftrightarrow$
- Refocused evaluator $\Longleftrightarrow$
- Pre-abstract machine $\Longleftrightarrow$
- Eval/Apply machine $\Longleftrightarrow$
- Eval/Apply in defunctionalized form $\Longleftrightarrow$
- CPS semantics $\Longleftrightarrow$
- 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*.

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 $\longmapsto$ given by redex rules where redex is in the hole of a context.

For example:

$$\text{argprog}\frac{e \rightsquigarrow e'}{(v\ e) \rightsquigarrow (v\ e')} \quad \Rightarrow \quad C ::= \ldots \mid C[(v\ [\ ])]$$

$$\text{funapp}\frac{}{((\lambda x.e)\ v) \rightsquigarrow [x \mapsto v]e} \quad \Rightarrow \quad C[((\lambda x.e)\ v)] \longmapsto C[[x \mapsto v]e]$$

The Multi-return λ-calculus (CbV fragment)

$$\text{funapp}_v \; \frac{}{(\lambda x.e)\ v \rightsquigarrow_v [x \mapsto v]e} \qquad \text{retlam} \; \frac{}{\lhd v\ l \rhd \rightsquigarrow_v (l\ v)}$$

$$\text{rpsel} \; \frac{}{\lhd v\ r_1 \ldots r_m \rhd \rightsquigarrow_v \lhd v\ r_1 \rhd}\ m > 1 \qquad \text{ret1} \; \frac{}{\lhd v\ \#1 \rhd \rightsquigarrow_v v}$$

$$\text{rettail} \; \frac{}{\lhd \lhd v\ \#i \rhd r_1 \ldots r_m \rhd \rightsquigarrow_v \lhd v\ r_i \rhd}\ 1 < i \leq m$$

$$\text{funprog} \; \frac{e_0 \rightsquigarrow_v e_0'}{(e_0\ e_1) \rightsquigarrow_v (e_0'\ e_1)} \qquad \text{argprog}_v \; \frac{e \rightsquigarrow_v e'}{(v\ e) \rightsquigarrow_v (v\ e')}$$

$$\text{retprog} \; \frac{e \rightsquigarrow_v e'}{\lhd e\ r_1 \ldots r_m \rhd \rightsquigarrow_v \lhd e'\ r_1 \ldots r_m \rhd}$$

Standard approach to Standard reduction

Reduction contexts and potential redexes:

$$
\begin{array}{rcll}
C & ::= & [\,] & \\
& | & C[(e\ [\,])] & \text{argprog} \\
& | & C[([\,]\ v)] & \text{funprog}_v \\
& | & C[\lhd [\,]\ r \ldots r \rhd] & \text{retprog}
\end{array}
$$

$$
\begin{array}{rcl}
p & ::= & (v\ v) \\
& | & \lhd v\ r \ldots r \rhd \\
& | & \lhd \lhd v\ r \ldots r \rhd\, r \ldots r \rhd
\end{array}
$$

Problem: Grammatical ambiguity

Unique decomposition does not hold.

$$
\begin{aligned}
decompose(\lhd\lhd v\ r_1\rhd r'_1\rhd) &= [\ ], \quad \lhd\lhd v\ r_1\rhd r'_1\rhd \\
decompose(\lhd\lhd v\ r_1\rhd r'_1\rhd) &= [\lhd[\ ]\rhd\ \ r'_1], \quad \lhd v\ r_1\rhd
\end{aligned}
$$

Fix 1: Tighter characterization of potential redexes

We can refine the grammar of potential redexes starting with the observation that $\triangleleft v \; \#i \triangleright, i > 1$ 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:

$$
\begin{array}{rcl}
p & ::= & (v\ v) \\
 & | & \lhd v\ r \ldots r \rhd \\
 & | & \lhd\lhd v\ r \ldots r \rhd r \ldots r \rhd
\end{array}
$$

$$\Rightarrow$$

$$
\begin{array}{rcl}
p & ::= & (v\ v) \\
 & | & \lhd v\ r \ldots r \rhd
\end{array}
$$

And make contraction context-sensitive, i.e. contracting a redex depends upon the context in which it appears.

Fix 2: Context-sensitive standard reduction

$$\begin{array}{lrcl} \text{Reduction contexts} & C & ::= & [\;] \\ & & \mid & C[(e\ [\;])] \\ & & \mid & C[([\;]\ v)] \\ & & \mid & C[\lhd[\;]\ r \ldots r\rhd] \end{array}$$

$$\begin{array}{lrcl} \text{Potential redexes} & p & ::= & (v\ v) \\ & & \mid & \lhd v\ r \ldots r\rhd \end{array}$$

$$\begin{array}{rcll} C[((\lambda x.e)\ v)] & \longmapsto & C[[x \mapsto v]e] & \\ C[\lhd v\ r_1 \ldots r_m\rhd] & \longmapsto & C[\lhd v\ r_1\rhd] & m > 1 \\ C[\lhd v\ l\rhd] & \longmapsto & C[(l\ v)] & \\ C[\lhd v\ \#1\rhd] & \longmapsto & C[v] & \\ (C[\lhd[\;]\ r_1 \ldots r_m\rhd])[\lhd v\ \#i\rhd] & \longmapsto & C[\lhd v\ r_i\rhd] & 1 < i \leq m \end{array}$$

Fix 2: Context-sensitive standard reduction

$$\text{Reduction contexts} \quad C \ ::= \ [\,] \ \mid \ C[(e\ [\,])] \ \mid \ C[([\,]\ v)] \ \mid \ C[\lhd[\,]\ r \ldots r\rhd]$$

$$\text{Potential redexes} \quad p \ ::= \ (v\ v) \ \mid \ \lhd v\ r \ldots r\rhd$$

$$\begin{array}{rcll}
C[((\lambda x.e)\ v)] & \longmapsto & C[[x \mapsto v]e] & \\
C[\lhd v\ r_1 \ldots r_m\rhd] & \longmapsto & C[\lhd v\ r_1\rhd] & m > 1 \\
C[\lhd v\ l\rhd] & \longmapsto & C[(l\ v)] & \\
C[\lhd v\ \#1\rhd] & \longmapsto & C[v] & \\
(C[\lhd[\,]\ r_1 \ldots r_m\rhd])[\lhd v\ \#i\rhd] & \longmapsto & C[\lhd v\ r_i\rhd] & 1 < i \leq m
\end{array}$$

Results

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

Lemma 2 (Correspondence with λ_{MR}) $e \leadsto_v e' \Longleftrightarrow e \longmapsto e'$.

These results are easy to prove and get us “off the ground” for producing interpreters that correspond with the original CbV fragment of λ_{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

Reduction-based evaluation

$$
\begin{array}{rcl}
\mathit{evaluate} : \mathsf{Exp} & \rightarrow & \mathsf{Val} + (\mathsf{RedCont} \times \mathsf{StuckRedex}) \\
\mathit{evaluate}(e) & = & \mathit{iterate}(\mathit{decompose}(e))
\end{array}
$$

$$
\begin{array}{rcll}
\mathit{iterate} : \mathsf{Val} + (\mathsf{RedCont} \times \mathsf{PotRedex}) & \rightarrow & \mathsf{Val} + (\mathsf{RedCont} \times \mathsf{StuckRedex}) & \\
\mathit{iterate}(v) & = & v & \\
\mathit{iterate}(C, ((\lambda x.e)\ v)) & = & \mathit{evaluate}(\mathit{plug}([x \mapsto v]e, C)) & \\
\mathit{iterate}(C, \triangleleft v\ l \triangleright) & = & \mathit{evaluate}(\mathit{plug}((l\ v), C)) & \\
\mathit{iterate}(C, \triangleleft v\ \#1 \triangleright) & = & \mathit{evaluate}(\mathit{plug}(v, C)) & \\
\mathit{iterate}(C, \triangleleft v\ r_1 \ldots r_m \triangleright) & = & \mathit{evaluate}(\mathit{plug}(\triangleleft v\ r_1 \triangleright, C)) & m > 1 \\
\mathit{iterate}(C[\triangleleft[\ ]\ r_1 \ldots r_m \triangleright], \triangleleft v\ \#i \triangleright) & = & \mathit{evaluate}(\mathit{plug}(\triangleleft v\ r_i \triangleright, C)) & 1 < i \leq m \\
\mathit{iterate}(C[\triangleleft[\ ]\ r_1 \ldots r_m \triangleright], \triangleleft v\ \#i \triangleright) & = & (C[\triangleleft[\ ]\ r_1 \ldots r_m \triangleright], \triangleleft v\ \#i \triangleright) & i > m > 1 \\
\mathit{iterate}(C, \triangleleft v\ \#i \triangleright) & = & (C, \triangleleft v\ \#i \triangleright) & \\
& & \quad C \neq C'[\triangleleft[\ ]\ r_1 \ldots r_m \triangleright] \text{ and } i > 1 & \\
\mathit{iterate}(C, (n\ v)) & = & (C, (n\ v)) &
\end{array}
$$

Refocused evaluation

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

$$evaluate(plug(e, C)) \;\Rightarrow\; iterate(decompose(plug(e, C)))$$

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

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

We rewrite the interpreter using $refocus = decompose \circ plug$ and are now free use any function extensionally equivalent to $refocus$.

Pre-abstract machine

Danvy and Nielsen provide a construction for an efficient $refocus$ focus from the standard reduction specification.

By construction, it is extensionally equivalent to $decompose \circ plug$.

The $refocus$ 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 *refocus* function is defined by cases on the grammar of expressions, $refocus_{aux}$ by cases on the top most context:

$$
\begin{array}{rcl}
refocus : \mathsf{Exp} \times \mathsf{RedCont} & \rightarrow & \mathsf{Val} + (\mathsf{RedCont} \times \mathsf{StuckRedex}) \\
refocus(v, C) & = & refocus_{aux}(C, v) \\
refocus((e_0\ e_1), C) & = & refocus(e_0, C[([\ ]\ e_1)]) \\
refocus(\triangleleft e\ r_1 \ldots r_m \triangleright, C) & = & refocus(e, C[\triangleleft[\ ]\ r_1 \ldots r_m \triangleright])
\end{array}
$$

$$
\begin{array}{rcl}
refocus_{aux} : \mathsf{RedCont} \times \mathsf{Val} & \rightarrow & \mathsf{Val} + (\mathsf{RedCont} \times \mathsf{StuckRedex}) \\
refocus_{aux}([\ ], v) & = & v \\
refocus_{aux}(C[([\ ]\ e)], v) & = & refocus(e, C \circ (v\ [\ ])) \\
refocus_{aux}(C[(v'\ [\ ])], v) & = & (C, (v'\ v)) \\
refocus_{aux}(C[\triangleleft[\ ]\ r_1 \ldots r_m \triangleright], v) & = & (C, \triangleleft v\ r_1 \ldots r_m \triangleright)
\end{array}
$$

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*.

CK abstract machine

Inlining the $iterate$ function gives a CK abstract machine.

CPS semantics

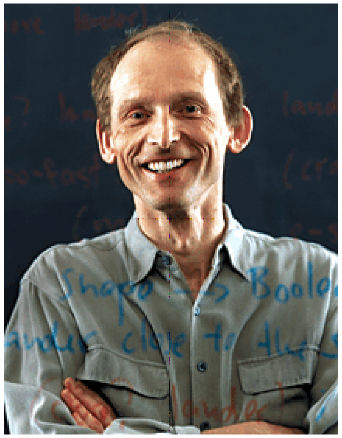
Refunctionalizing (Church-encoding the reduction context datatype) the CK machine gives a CPS semantics.

Big-step operational semantics

Direct-style transformation of the CPS semantics yields the big-step operational semantics.

Acknowledgements

Olivier Danvy



Matthias Felleisen

Acknowledgements

Olivier Danvy

Matthias Felleisen