A Functional Correspondence between Parametric Evaluators and which Abstract Machines?

Álvaro García Pérez

Oxford, June 2009

Outline

Motivation

- Definitional interpreters for higher-order languages [3].
- Abstract machines can be obtained from definitional interpreters [1].
- Definitional interpreters, evaluation order, ‘reducers’.
- Focus: untyped lambda calculus. Free variables, no constants, no closures, no environments.
- There is a higher-order, parametrised reducer. Particular reducers (AOR, NOR, CBV, CBN) defined as fixed points.
- Abstract plugable machine from reducer?
- Related definitional interpreter (from reducer if possible)?
- Abstract plugable machine from definitional interpreter?

Abstract Syntax of Untyped λ-terms

$$\begin{aligned}\textbf{data}\ \mathit{SLTerm} = {} & \mathit{Var}\ \mathit{String} \\ \mid {} & \mathit{App}\ \mathit{SLTerm}\ \mathit{SLTerm} \\ \mid {} & \mathit{Lam}\ \mathit{String}\ \mathit{SLTerm}\end{aligned}$$

Assume capture-avoiding substitution.

We are not interested in substitution issues (DeBruijn, nested rep., HOAS, etc),

Some definitions

Definition

A term t is in normal form (NF) iff:

- $t = \lambda x.N$ where N is a normal form.
- $t = ((\ldots(xN_1)N_2)\ldots N_n)$ where x is a free variable and N_i are in normal form, with $n \geqslant 0$.

Definition

A term t is in weak normal form (WNF) iff:

- $t = \lambda x.B$ whatever the B.
- $t = ((\ldots(xW_1)W_2)\ldots W_n)$ where x is a free variable and W_i are in weak normal form.

Some definitions (cont'd)

Evaluation orders:

Name	**Normal Form**	**Strategy**
Applicative Order (AOR)	NF	leftmost innermost
Normal Order (NOR)	NF	leftmost outermost
Call-by-value (CBV)	WNF	leftmost innermost
Call-by-name (CBN)	WNF	leftmost outermost

Naive AOR Interpreter

$$
\begin{aligned}
&\mathit{reduce} && :: \mathit{SLTerm} \to \mathit{SLTerm} \\
&\mathit{reduce}\ v@(\mathit{Var}\ x) && = v \\
&\mathit{reduce}\ (\mathit{Lam}\ v\ b) && = \mathit{Lam}\ v\ (\mathit{reduce}\ b) \\
&\mathit{reduce}\ (\mathit{App}\ m\ n) && = \mathbf{let}\ m' = \mathit{reduce}\ m \\
& && \qquad\quad n' = \mathit{reduce}\ n \\
& && \quad \mathbf{in\ case}\ m'\ \mathbf{of} \\
& && \qquad (\mathit{Lam}\ v\ b) \to \mathit{reduce}\ (\mathit{subst}\ v\ b\ n') \\
& && \qquad \_ \qquad\quad \to \mathit{App}\ m'\ n'
\end{aligned}
$$

Ancillary *apply* obviated by case expression.

Error: $(\lambda x.\lambda y.y)\omega$, where $\omega = (\lambda x.xx)(\lambda x.xx)$

Reflecting the Evaluation Order

Forcing evaluation in Haskell:

- Pattern matching.
- Strict application ($!).
- Continuation Passing Style (CPS) [3, 2].

We can use monads to encapsulate the forcing strategy [4].

- We can force the evaluation in the implementation of the monad.
- CPS expressed by the Continuation Monad.

AOR Using ($!)

$$
\begin{aligned}
&\mathit{reduce} && :: \mathit{SLTerm} \to \mathit{SLTerm} \\
&\mathit{reduce}\ v@(\mathit{Var}\ x) && = v \\
&\mathit{reduce}\ (\mathit{Lam}\ v\ b) && = \mathit{Lam}\ v\ (\mathit{reduce}\ b) \\
&\mathit{reduce}\ (\mathit{App}\ m\ n) && = \mathbf{let}\ m' = \mathit{reduce}\ m \\
&&&\qquad\quad n' = \mathit{reduce}\ n \\
&&&\quad \mathbf{in\ case}\ m'\ \mathbf{of} \\
&&&\qquad (\mathit{Lam}\ v\ b) \to \mathit{reduce}\ (\mathit{subst}\ v\ b\ \$!\ n') \\
&&&\qquad \_ \qquad\quad \to \mathit{App}\ m'\ \$!\ n'
\end{aligned}
$$

NOR Using ($!)

$$
\begin{aligned}
&\mathit{reduce} && :: \mathit{SLTerm} \to \mathit{SLTerm} \\
&\mathit{reduce}\ v@(\mathit{Var}\ x) && = v \\
&\mathit{reduce}\ (\mathit{Lam}\ v\ b) && = \mathit{Lam}\ v\ (\mathit{reduce}\ b) \\
&\mathit{reduce}\ (\mathit{App}\ m\ n) && = \mathbf{let}\ m' = \mathit{reduce}\ m \\
&&&\qquad\quad n' = \mathit{reduce}\ n \\
&&&\quad \mathbf{in\ case}\ m'\ \mathbf{of} \\
&&&\qquad (\mathit{Lam}\ v\ b) \to \mathit{reduce}\ (\mathit{subst}\ v\ b\ \$\ n) \\
&&&\qquad \_ \qquad\quad \to \mathit{App}\ m'\ \$\ n'
\end{aligned}
$$

CBV Using ($!)

$$
\begin{aligned}
&\mathit{reduce} && :: \mathit{SLTerm} \to \mathit{SLTerm} \\
&\mathit{reduce}\ v@(\mathit{Var}\ \_) && = v \\
&\mathit{reduce}\ l@(\mathit{Lam}\ v\ b) && = l \\
&\mathit{reduce}\ (\mathit{App}\ m\ n) && = \mathbf{let}\ m' = \mathit{reduce}\ m \\
&&&\qquad\quad n' = \mathit{reduce}\ n \\
&&&\quad \mathbf{in\ case}\ m'\ \mathbf{of} \\
&&&\qquad (\mathit{Lam}\ x\ b) \to \mathit{reduce}\ (\mathit{subst}\ x\ b\ \$!\ n') \\
&&&\qquad \_ \qquad\qquad \to \mathit{App}\ m'\ \$!\ n'
\end{aligned}
$$

CBN Using ($!)

$$
\begin{aligned}
&\mathit{reduce} && :: \mathit{SLTerm} \to \mathit{SLTerm} \\
&\mathit{reduce}\ v@(\mathit{Var}\ x) && = v \\
&\mathit{reduce}\ l@(\mathit{Lam}\ v\ b) && = l \\
&\mathit{reduce}\ (\mathit{App}\ m\ n) && = \mathbf{let}\ m' = \mathit{reduce}\ m \\
&&&\qquad\quad n' = \mathit{reduce}\ n \\
&&&\quad \mathbf{in\ case}\ m'\ \mathbf{of} \\
&&&\qquad (\mathit{Lam}\ v\ b) \to \mathit{reduce}\ (\mathit{subst}\ v\ b\ \$\ n) \\
&&&\qquad \_ \qquad\qquad \to \mathit{App}\ m'\ \$\ n'
\end{aligned}
$$

Using Monads

Monads encapsulating the forcing strategy could be written.

- Strict application ($!):

$$
\begin{aligned}
&\textbf{data}\ \mathit{StrictApp}\ a = \mathit{StrictApp}\{\mathit{out} :: a\}\\
&\textbf{instance}\ \mathit{Monad}\ \mathit{StrictApp}\ \textbf{where}\\
&\quad \mathit{return} \quad = \mathit{StrictApp}\\
&\quad m \gg\!= f = f\ \$!\ (\mathit{out}\ m)
\end{aligned}
$$

- Continuations:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Cont}\ r\ a = \mathit{Cont}\{\mathit{runCont} :: (a \to r) \to r\}\\
&\textbf{instance}\ \mathit{Monad}\ (\mathit{Cont}\ r)\ \textbf{where}\\
&\quad \mathit{return}\ a = \mathit{Cont}\ (\lambda c \to c\ a)\\
&\quad m \gg\!= k = \mathit{Cont}\ (\lambda c \to (\mathit{runCont}\ m)\\
&\qquad\qquad\qquad\qquad (\lambda a \to (\mathit{runCont}\ (k\ a))\ c))
\end{aligned}
$$

AOR Using Monads

$$
\begin{aligned}
&\mathit{reduce} && :: \mathit{Monad}\ m \Rightarrow \mathit{SLTerm} \rightarrow m\ \mathit{SLTerm} \\
&\mathit{reduce}\ v@(\mathit{Var}\ \_) && = \mathit{return}\ v \\
&\mathit{reduce}\ (\mathit{Lam}\ v\ b) && = \mathbf{do}\ b' \leftarrow \mathit{reduce}\ b \\
&&&\qquad\quad \mathit{return}\ (\mathit{Lam}\ v\ b') \\
&\mathit{reduce}\ (\mathit{App}\ m\ n) && = \mathbf{do}\ m' \leftarrow \mathit{reduce}\ m \\
&&&\qquad\quad n' \leftarrow \mathit{reduce}\ n \\
&&&\qquad\quad \mathbf{case}\ m'\ \mathbf{of} \\
&&&\qquad\qquad (\mathit{Lam}\ v\ b) \rightarrow \mathit{reduce}\ (\mathit{subst}\ v\ b\ n') \\
&&&\qquad\qquad \_ \qquad\qquad \rightarrow \mathit{return}\ (\mathit{App}\ m'\ n')
\end{aligned}
$$

NOR Using Monads

$$
\begin{aligned}
&\mathit{reduce} && :: \mathit{Monad}\ m \Rightarrow \mathit{SLTerm} \rightarrow m\ \mathit{SLTerm} \\
&\mathit{reduce}\ v@(\mathit{Var}\ \_) && = \mathit{return}\ v \\
&\mathit{reduce}\ (\mathit{Lam}\ v\ b) && = \mathbf{do}\ b' \leftarrow \mathit{reduce}\ b \\
&&& \quad\ \mathit{return}\ (\mathit{Lam}\ v\ b') \\
&\mathit{reduce}\ (\mathit{App}\ m\ n) && = \mathbf{do}\ m' \leftarrow \mathit{reduce}\ m \\
&&& \quad\ n' \leftarrow \mathit{reduce}\ n \\
&&& \quad\ \mathbf{case}\ m'\ \mathbf{of} \\
&&& \quad\quad (\mathit{Lam}\ v\ b) \rightarrow \mathit{reduce}\ (\mathit{subst}\ v\ b\ n) \\
&&& \quad\quad \_ \qquad\quad \rightarrow \mathit{return}\ (\mathit{App}\ m'\ n')
\end{aligned}
$$

CBV Using Monads

$$
\begin{aligned}
&\mathit{reduce} && :: \mathit{Monad}\ m \Rightarrow \mathit{SLTerm} \rightarrow m\ \mathit{SLTerm} \\
&\mathit{reduce}\ v@(\mathit{Var}\ \_) && = \mathit{return}\ v \\
&\mathit{reduce}\ l@(\mathit{Lam}\ v\ b) && = \mathit{return}\ l \\
&\mathit{reduce}\ (\mathit{App}\ m\ n) && = \mathbf{do}\ m' \leftarrow \mathit{reduce}\ m \\
&&&\qquad\quad n' \leftarrow \mathit{reduce}\ n \\
&&&\qquad\quad \mathbf{case}\ m'\ \mathbf{of} \\
&&&\qquad\qquad (\mathit{Lam}\ v\ b) \rightarrow \mathit{reduce}\ (\mathit{subst}\ v\ b\ n') \\
&&&\qquad\qquad \_ \rightarrow \mathit{return}\ (\mathit{App}\ m'\ n')
\end{aligned}
$$

CBN Using Monads

$$
\begin{aligned}
&\mathit{reduce} && :: \mathit{Monad}\ m \Rightarrow \mathit{SLTerm} \rightarrow m\ \mathit{SLTerm} \\
&\mathit{reduce}\ v@(\mathit{Var}\ \_) && = \mathit{return}\ v \\
&\mathit{reduce}\ l@(\mathit{Lam}\ v\ b) && = \mathit{return}\ l \\
&\mathit{reduce}\ (\mathit{App}\ m\ n) && = \mathbf{do}\ m' \leftarrow \mathit{reduce}\ m \\
&&&\qquad\quad n' \leftarrow \mathit{reduce}\ n \\
&&&\qquad\quad \mathbf{case}\ m'\ \mathbf{of} \\
&&&\qquad\qquad (\mathit{Lam}\ v\ b) \rightarrow \mathit{reduce}\ (\mathit{subst}\ v\ b\ n) \\
&&&\qquad\qquad \_ \rightarrow \mathit{return}\ (\mathit{App}\ m'\ n')
\end{aligned}
$$

Parametric Reducer

All the given reducers follow the same pattern

$$
\begin{aligned}
&\textbf{type } \mathit{Red} = \mathit{SLTerm} \to \mathit{SLTerm} \\
&\mathit{reduce} :: \mathit{Red} \to \mathit{Red} \to \mathit{Red} \to \mathit{Red} \\
&\mathit{reduce}\ l\ ar\ ap\ v@(\mathit{Var}\ x) = v \\
&\mathit{reduce}\ l\ ar\ ap\ (\mathit{Lam}\ v\ b) = (\mathit{Lam}\ v\ (l\ b)) \\
&\mathit{reduce}\ l\ ar\ ap\ (\mathit{App}\ m\ n) = \\
&\quad \textbf{let } m' = \mathit{reduce}\ l\ ar\ ap\ m \\
&\qquad\ \ n' = ar\ n \\
&\quad \textbf{in case } m' \textbf{ of} \\
&\qquad (\mathit{Lam}\ v\ b) \to \mathit{reduce}\ l\ ar\ ap\ (\mathit{subst}\ v\ b\ \$!\ n') \\
&\qquad \_ \qquad\qquad \to \textbf{let } n'' = ap\ n' \\
&\qquad\qquad\qquad\ \ \textbf{in } \mathit{App}\ m'\ \$!\ n'
\end{aligned}
$$

Parametric Reducer (cont'd)

We can define reducers for every evaluation order as fix points of the parametric interpreter:

$$
\begin{aligned}
aor &= reduce\ aor\ aor\ id \\
nor &= reduce\ nor\ id\ \ nor \\
cbv &= reduce\ id\ \ cbv\ id \\
cbn &= reduce\ id\ \ id\ \ cbn
\end{aligned}
$$

Monadic Parametric Reducer

```
type Red m = SLTerm → m SLTerm
reduce :: Monad m ⇒ Red m → Red m → Red m → Red m
reduce l ar ap v@(Var x) = return v
reduce l ar ap (Lam v b) = do b' ← l b
                              return (Lam v b')
reduce l ar ap (App m n) =
  do m' ← reduce l ar ap m
     n' ← ar n
     case m' of
       (Lam v b) → reduce l ar ap (subst v b n')
       _         → do n'' ← ap n'
                      return (App m' n'')
```

Monadic Parametric Reducer (cont'd)

$$
\begin{aligned}
aor &= reduce\ aor\ \ aor\ \ return \\
nor &= reduce\ nor\ \ return\ nor \\
cbv &= reduce\ return\ cbv\ \ return \\
cbn &= reduce\ return\ return\ cbn
\end{aligned}
$$

Alternative Parametric Reducer

Why not use n instead of n' when applying ap?

```
type Red m = SLTerm → m SLTerm
reduce :: Monad m ⇒ Red m → Red m → Red m → Red m
reduce l ar ap v@(Var x) = return v
reduce l ar ap (Lam v b) = do b' ← l b
                              return (Lam v b')
reduce l ar ap (App m n) =
  do m' ← reduce l ar ap m
     n' ← ar n
     case m' of
       (Lam v b) → reduce l ar ap (subst v b n')
       _         → do n'' ← ap n
                      return (App m' n'')
```

Alternatives to Parametric Reducer (cont'd)

The fixpoint definition will change accordingly.

$$
\begin{array}{lllll}
aor & = & reduce & aor & aor \quad aor \\
nor & = & reduce & nor & return \quad nor \\
cbv & = & reduce & return & cbv \quad cbv \\
cbn & = & reduce & return & return \quad cbn
\end{array}
$$

Reducers Using Primitive Recursion

Four other reducers can be expressed using primitive recursion.

$$
\begin{aligned}
\mathit{triv} &= \mathit{reduce}\ \mathit{return}\ \mathit{return}\ \mathit{return} \\
\mathit{rlab} &= \mathit{reduce}\ \mathit{rlab}\ \ \mathit{return}\ \mathit{return} \\
\mathit{cbn} &= \mathit{reduce}\ \mathit{return}\ \mathit{cbn}\ \ \mathit{return} \\
\mathit{nor} &= \mathit{reduce}\ \mathit{nor}\ \ \mathit{nor}\ \ \mathit{return} \\
\mathit{rop} &= \mathit{reduce}\ \mathit{return}\ \mathit{return}\ \mathit{rop} \\
\mathit{rlop} &= \mathit{reduce}\ \mathit{rlop}\ \ \mathit{return}\ \mathit{rlop} \\
\mathit{cbv} &= \mathit{reduce}\ \mathit{return}\ \mathit{cbv}\ \ \mathit{cbv} \\
\mathit{aor} &= \mathit{reduce}\ \mathit{aor}\ \ \mathit{aor}\ \ \mathit{aor}
\end{aligned}
$$

triv: β-reduction (enforces reducing operators in applications).
rlab: β-reduction plus lambda abstraction bodies.
rop: β-reduction plus operands of lambda abstractions.
rlab: β-reduction plus lambda abstraction bodies and operands of lambda abstractions.

Reducers Using Primitive Recursion

This defines a model of inheritance where we have a basic reducer plus three orthogonal behaviours. *triv* is the ancestor and *aor* the successor of every reducer.

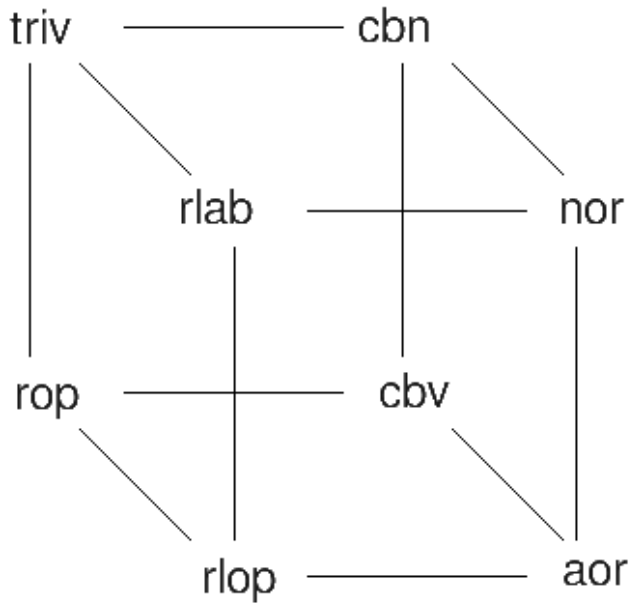

Figure: Reducers cube

Reducers Using Mutual Recursion

An infinite number of reducers can be specified using mutual recursion:

$$cbn' = reduce\ cbn \quad return\ cbn$$
$$cbn = reduce\ return\ return\ cbn$$

Call-by-name, but reducing the first nesting level of lambda abstractions.

The depth at which reduction strategies work can be specified.

More Generic Reducers

Why not considering more parameters in the generic reducer?

- Moving the reduction of the operands of applications.
- Considering the *subst* case.
- Considering the *App* case.

More Generic Reducers (cont'd)

```
type Red m = SLTerm → m SLTerm
reduce :: Monad m ⇒ Red m → Red m → Red m → Red m →
    Red m → Red m
reduce l ar1 ar2 su ap v@(Var x) = return v
reduce l ar1 ar2 su ap (Lam v b) = do b' ← l b
                                      return (Lam v b')
reduce l ar1 ar2 su ap (App m n) =
    do m' ← reduce l ar1 ar2 su ap m
       case m' of
         (Lam v b) → do n' ← ar1 n
                        su (subst v b n')
         _         → do n' ← ar2 n
                        ap (App m' n')
```

Conclusions and Future Work

Work in progress:

- Generic reducers with different number of parameters.
- Mutual recursion when defining fix points.
- Inheritance models and relationship with *mixins*.
- Evaluators and 'reducers' correspondence to abstract machines.
- Definitional interpreter from reducer.
- Higher-order plugable abstract machine, if possible.


M. S. Ager, D. Biernacki, O. Danvy, and Midtgaard J.
A functional correspondence between evaluators and abstract machines.
Technical Report BRICS RS-03-35, Department of Computer Science, University of Aarhus, Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark, March 2003.


John Hatcliff and Olivier Danvy.
A generic account of continuation-passing styles.
In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 458–471, New York, NY, USA, 1994. ACM.


John C. Reynolds.
Definitional interpreters for higher-order programming languages.
In *ACM '72: Proceedings of the ACM annual conference*, pages 717–740, New York, NY, USA, 1972. ACM.


Philip Wadler.
The essence of functional programming.

In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, New York, NY, USA, 1992. ACM.