A formal derivation of an executable Krivine machine

Wouter Swierstra
Universiteit Utrecht

IFIP WG 2.1
Meeting 68; Rome, Italy
\( \beta \) reduction

\[
(\lambda x \cdot t_0) \ t_1 \rightarrow t_0 \ \{ t_1/x \}
\]
Motivation

• Implementing $\beta$-reduction through substitutions is a terrible idea!

• Instead, modern compilers evaluate lambda terms using an abstract machine, such as Haskell’s STG or OCaml’s CAM.

• Such abstract machines are usually described as tail-recursive functions/finite state machines.
Who comes up with these things?
Olivier Danvy
and his many students and collaborators
Most of our implementations of the abstract machines raise compiler warnings about non-exhaustive matches. These are inherent to programming abstract machines in an ML-like language – Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, Jan Midtgaard
Laissez-faire vs nanny state
Swedish welfare state
Outline

1. A terminating small-step evaluator
2. A small-step abstract machine (refocusing)
3. A Krivine machine (inlining)
Small step evaluation
Types

data Ty : Set where
  0 : Ty
  _=>_  : Ty -> Ty -> Ty

Context : Set
Context = List Ty
Terms

data Term : Context -> Ty -> Set where
  Lam : Term (Cons u Γ) v
       -> Term Γ (u => v)
  App : Term Γ (u => v) -> Term Γ u
       -> Term Γ v
  Var : Ref Γ u -> Term Γ u
Closed terms only
**Reduction rules**

**LOOKUP**  \[ i [c_1, c_2, \ldots c_n] \rightarrow c_i \]

**APP**  \[(t_0 t_1) [env] \rightarrow (t_0 [env])(t_1 [env])\]

**BETA**  \[((\lambda t) [env]) x \rightarrow t [x \cdot env]\]

**LEFT**  if \( c_0 \rightarrow c'_0 \) then \( c_0 c_1 \rightarrow c'_0 c_1 \)
Closed terms

data Closed : Ty -> Set where
  Closure : Term Γ u -> Env Γ
            -> Closed u
  Clapp : Closed (u => v) -> Closed u
            -> Closed v

data Env : Context -> Set where
  Nil : Env Nil
  _·_ : Closed u -> Env Γ
        -> Env (Cons u Γ)
**Reduction rules**

\[ \text{LOOKUP} \quad i [c_1, c_2, \ldots, c_n] \rightarrow c_i \]

\[ \text{APP} \quad (t_0 \ t_1) [env] \rightarrow (t_0 [env]) \ (t_1 [env]) \]

\[ \text{BETA} \quad (\lambda t) [env]) \ x \rightarrow t [x \cdot env] \]

\[ \text{LEFT} \quad \text{if } c_0 \rightarrow c'_0 \text{ then } c_0 \ c_1 \rightarrow c'_0 \ c_1 \]
Reduction rules

**LOOKUP** \( i [c_1, c_2, \ldots, c_n] \rightarrow c_i \)

**APP** \((t_0 t_1) [env] \rightarrow (t_0 [env]) (t_1 [env])\)

**BETA** \(((\lambda t) [env]) x \rightarrow t [x \cdot env]\)

**LEFT** \( \text{if } c_0 \rightarrow c' \text{ then } c_0 \cdot c_1 \rightarrow c_0' c_1 \)
Reduction rules

\[ E ::= \square \mid E \mathit{t} \]

**Lookup** \[ E\{i [c_1, c_2, \ldots c_n]\} \rightarrow E\{c_i\} \]

**App** \[ E\{(t_0 t_1) [\mathit{env}]\} \rightarrow E\{(t_0 [\mathit{env}]) (t_1 [\mathit{env}])\} \]

**Beta** \[ E\{((\lambda t) [\mathit{env}]) c\} \rightarrow E\{t [c \cdot \mathit{env}]\} \]
Head reduction in three steps

• Decompose the term into a redex and evaluation context;
• Contract the redex;
• Plug the result back into the context.
data Redex : Ty -> Set where
  Lookup : Ref Γ u -> Env Γ -> Redex u
  App : Term Γ (u => v) -> Term Γ u
       -> Env Γ -> Redex v
  Beta : Term (Cons u Γ) v -> Env Γ
       -> Closed u -> Redex v
Contraction

\[
\text{contract} : \text{Redex } u \rightarrow \text{Closed } u
\]
\[
\text{contract} (\text{Lookup } i \text{ env}) = \text{ env } \downarrow i
\]
\[
\text{contract} (\text{App } f \text{ x env}) =
\]
\[
\text{Clapp} (\text{Closure } f \text{ env}) (\text{Closure } x \text{ env})
\]
\[
\text{contract} (\text{Beta } \text{ body env arg}) =
\]
\[
\text{Closure } \text{ body} (\text{arg } \cdot \text{ env})
\]
Decomposition as a view

- **Idea:** every closed term is:
  - a value;
  - or a redex in some evaluation context.

- Define a view on closed terms.
Views: example

• Natural numbers are typically defined using the Peano axioms.

• But sometimes you want to use the fact that every number is even or odd, e.g.
  • when converting to a binary representation;
  • or proving $\sqrt{2}$ is irrational.

• But why is that a valid proof principle?
• How can we derive even-odd induction from Peano induction?

• Define a data type
  • `EvenOdd : Nat -> Set`

• Define a covering function
  • `evenOdd : (n : Nat) -> EvenOdd n`
The view data type

data EvenOdd : Nat -> Set where
  IsEven : (k : Nat) -> EvenOdd (double k)
  IsOdd : (k : Nat) -> EvenOdd (Succ (double k))
Covering function

evenOdd : (n : Nat) -> EvenOdd n
evenOdd Zero = IsEven Zero
evenOdd (Succ Zero) = IsOdd Zero
evenOdd (Succ (Succ k)) with evenOdd k
... | IsEven k' = IsEven (Succ k')
... | IsOdd k' = IsOdd (Succ k')
Example

example: Nat -> ...

eexample n with evenOdd n
eexample .(double k) | IsEven k
= ...

eexample .(Succ (double k)) | IsOdd k
= ...
Decomposition as a view

• **Idea:** every closed term is:
  • a value;
  • or a redex in some evaluation context.
• Define a view on closed terms.
Evaluation contexts

data EvalContext : Ty -> Ty -> Set where
  MT : EvalContext u u
  ARG : Closed u -> EvalContext v w
       -> EvalContext (u => v) w
Plug

plug : EvalContext u v -> Closed u -> Closed v
plug MT f = f
plug (ARG x ctx) f = plug ctx (Clapp f x)
Decomposition

data Decomposition : Closed u -> Set where
  Val : (t : Closed u) -> isVal t
       -> Decomposition t

Decompose : (r : Redex v)
       -> (ctx : EvalContext v u)
       -> Decomposition (plug ctx (fromRedex r))
Decompose

decompose : (c : Closed u) -> Decomposition c
decompose c = load MT c
load : (ctx : EvalContext u v) (c : Closed u) ->
    Decomposition (plug ctx c)
load ctx (Closure (Lam body) env) =
    unload ctx body env
load ctx (Closure (App f x) env) =
    Decompose (App f x env) ctx
load ctx (Closure (Var i) env) =
    Decompose (Lookup i env) ctx
load ctx (Clapp f x) = load (ARG x ctx) f

unload : (ctx : EvalContext (u => v) w) ->
    (body : Term (Cons u G) v) (env : Env G) ->
    Decomposition (plug ctx (Closure (Lam body) env))
unload MT body env = Val body env
unload (ARG arg ctx) body env =
    Decompose (Beta body env arg) ctx
Head-reduction

headReduce : Closed u -> Closed u
headReduce c with decompose c
  ... | Val val p = val
  ... | Decompose redex ctx
       = plug ctx (contract redex)
Iterated head reduction

evaluate : Closed u -> Value u
evaluate c = iterate (decompose c)
  where
  iterate : Decomposition c -> Value u
  iterate (Val val p) = Val val p
  iterate (Decompose r ctx)
    = iterate (decompose (plug ctx (contract r)))
Iterated head reduction

evaluate : Closed u -> Value u
evaluate c = iterate (decompose c)
  where
  iterate : Decomposition c -> Value u
  iterate (Val val p) = Val val p
  iterate (Decompose r ctx)
    = iterate (decompose (plug ctx (contract r)))
Iterated head reduction

evaluate : Closed u -> Value u
\[\text{evaluate } c = \text{iterate } (\text{decompose } c)\]

where

iterate : Decomposition c -> Value u
iterate (Val val p) = Val val p
iterate (Decompose r ctx) = \text{iterate } (\text{decompose } (\text{plug } ctx (\text{contract } r)))
The Bove-Capretta method
terminates(ν)

t\rightarrow t' \quad \text{terminates}(t')

\text{terminates}(t)
data Trace : Decomposition c -> Set where
    Done : (val : Closed u) -> (p : isVal val) -> Trace (Val val p)
    Step : Trace (decompose (plug ctx (contract r))) -> Trace (Decompose r ctx)
Iterated head reduction, again

iterate : {u : Ty} {c : Closed u} ->
  (d : Decomposition c) -> Trace d -> Value u
iterate (Val val p) Done = Val val p
iterate (Decompose r ctx) (Step step) =
  let d' = decompose (plug ctx (contract r)) in
iterate d' step
Nearly done

We still need to find a trace for every term...

(c : Closed u) -> Trace (decompose c)
We still need to find a trace for every term...

\[(c : \text{Closed } u) \rightarrow \text{trace} \ (\text{decompose } c)\]
Nearly done

We still need to find a trace for every term...

(c : Closed u) ➝ Trace (decompose c)

Yet we know that the simply typed lambda calculus is strongly normalizing...
Logical relation

Reducible : (u : Ty) -> (t : Closed u) -> Set
Reducible 0 t = Trace (decompose t)
Reducible (u => v) t
  = Pair (Trace (decompose t))
    ((x : Closed u) -> Reducible u x
     -> Reducible (Clapp t x))
Required lemmas

lemma1 : (t : Closed u) -> Reducible (headReduce t) -> Reducible t

lemma2 : (t : Term G u) (env : Env G) -> ReducibleEnv env -> Reducible (Closure t env)
Result!

\[
\text{theorem} : (c : \text{Closed } u) \rightarrow \text{Reducible } c
\]
\[
\text{theorem} (\text{Closure } t \text{ env})
  = \text{lemma2 } t \text{ env } (\text{envTheorem } env)
\]
\[
\text{theorem} (\text{Clapp } f \text{ x})
  = \text{snd } (\text{theorem } f) \text{ x } (\text{theorem } x)
\]

\[
\text{termination} : (c : \text{Closed } u) \rightarrow
  \text{Trace } (\text{decompose } c)
\]

...an easy corollary
Finally, evaluation

\[
\text{evaluate} : \text{Closed } u \rightarrow \text{Value } u
\]
\[
\text{evaluate } t =
\]
\[
\text{iterate } (\text{decompose } t) (\text{termination } t)
\]
The story so far...

• Data types for terms, closed terms, values, redexes, evaluation contexts.

• Defined a three step head-reduction function: decompose, contract, plug.

• Proven that iterated head reduction yields a normal form...

• ... and used this to define a normalization function.
What’s next?

• Use Danvy & Nielsen’s refocusing transformation to define a *small-step abstract machine*

• Inline the *iterate* function (and one or two minor changes), yields the *Krivine abstract machine*.

• Prove that each transformation preserves the termination behaviour and semantics.
A term
A redex and an evaluation context
Contract the redex
Plug and repeat
The drawback

• To contract a single redex, we need to:
  • traverse the term to find a redex;
  • contract the redex;
  • traverse the context to plug back the contractum.
Refocusing

- The refocusing transformation (Danvy & Nielsen) avoids these traversals.
- Instead, given a decomposition, it navigates to the next redex immediately.
- Refocusing behaves just the same as decompose . plug
Refocus summary

\[ \text{refocus} : (\text{ctx} : \text{EvalContext u v}) \rightarrow (c : \text{Closed u}) \rightarrow \text{Decomposition (plug ctx c)} \]

\[ \text{refocusCorrect} : (\text{ctx} : \text{EvalContext u v}) \rightarrow (c : \text{Closed u}) \rightarrow \text{refocus ctx c == decompose (plug ctx c)} \]
Refocus, details

refocus : (ctx : EvalContext u v) (c : Closed u) -> Decomposition (plug ctx c)
refocus MT (Closure (Lam body) env) = Val body env
refocus (ARG x ctx) (Closure (Lam body) env)
  = Decompose (Beta body env x) ctx
refocus ctx (Closure (Var i) env)
  = Decompose (Lookup i env) ctx
refocus ctx (Closure (App f x) env)
  = Decompose (App f x env) ctx
refocus ctx (Clapp f x) = refocus (ARG x ctx) f
What else?

• It is easy to prove that iteratively refocusing and contracting redexes produces the same result as the small step evaluator.

• And that if the Trace data type is inhabited, then so is the corresponding data type for the refocussing evaluator.
The Krivine machine

• Now inline the iterate function;
• and disallow closed applications;
• and compress ‘corridor transitions’.
The Krivine machine

refocus :
(\text{ctx} : \text{EvalContext} \ u \ v) \rightarrow 
(\text{t} : \text{Term} \ \Gamma \ u) \rightarrow 
(\text{env} : \text{Env} \ \Gamma) \rightarrow \text{Value} \ v

\text{refocus} \ \text{ctx} \ (\text{Var} \ i) \ \text{env} = 
\begin{array}{l}
\text{let} \ \text{Closure} \ \text{t} \ \text{env}' = \text{lookup} \ i \ \text{env} \ q \ \text{in} \\
\text{refocus} \ \text{ctx} \ \text{t} \ \text{env}'
\end{array}

\text{refocus} \ \text{ctx} \ (\text{App} \ f \ x) \ \text{env} \\
= \text{refocus} \ (\text{ARG} \ (\text{Closure} \ x \ \text{env}) \ \text{ctx}) \ f \ \text{env}

\text{refocus} \ (\text{ARG} \ x \ \text{ctx}) \ (\text{Lam} \ \text{body}) \ \text{env} \\
= \text{refocus} \ \text{ctx} \ \text{body} \ (x \cdot \ \text{env})

\text{refocus} \ \text{MT} \ (\text{Lam} \ \text{body}) \ \text{env} \\
= \text{Val} \ (\text{Closure} \ (\text{Lam} \ \text{body}) \ \text{env})
Once again...

• We need to prove that this function terminates...

• ... by adapting the proof we saw for the refocusing evaluator.

• ... and show that it produces the same value as our previous evaluation functions.
Conclusions

Most of our implementations of the abstract machines raise compiler warnings about non-exhaustive matches. These are inherent to programming abstract machines in an ML-like language.
Conclusions

Most of our implementations of the abstract machines raise compiler warnings about non-exhaustive matches. These are inherent to programming abstract machines in an ML-like language.

Agda is not an ML-like language.
Conclusions

Most of our implementations of the abstract machines raise compiler warnings about non-exhaustive matches. These are inherent to programming abstract machines in an ML-like language.
Conclusions

Most of our implementations of the abstract machines raise compiler warnings about non-exhaustive matches. These are inherent to programming abstract machines in an ML-like language.

Using dependent types exposes structure that is not apparent in ML-like languages.