From Math to Machine
A formal derivation of an executable Krivine Machine

Wouter Swierstra
Brouwer Seminar
\[ (\lambda x . t_0) \ t_1 \rightarrow t_0 \{ t_1 / x \} \]
Substitution

• Realizing $\beta$-reduction through substitution 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.
Abstract machines

- Abstract machines have many applications:
  - trace and debug evaluation;
  - study operational behaviour of new language constructs (continuations, threads, etc.);
  - a framework for program analysis.
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
Outline

1. Define a small step evaluator for the simply typed lambda calculus in Agda;

2. Show that this evaluator terminates;

3. From this evaluator, derive a Krivine machine in two short steps;

4. Prove that these preserve the semantics and termination behaviour of the original.
Small step evaluation
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

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)
Values

isVal : Closed u -> Set

isVal ( Closure ( Lam body ) env ) = Unit

isVal _ = Empty

data Value ( u : Ty ) : Set where

Val : ( c : Closed u ) -> isVal c

-> Value u
Reduction rules

\textbf{Lookup} \quad i [c_1, c_2, \ldots, c_n] \rightarrow c_i

\textbf{App} \quad (t_0 \; t_1) [env] \rightarrow (t_0 [env]) \; (t_1 [env])

\textbf{Beta} \quad ((\lambda t) [env]) \; x \rightarrow t [x \cdot env]

\textbf{Left} \quad \text{if } c_0 \rightarrow c'_0 \text{ then } c_0 \; c_1 \rightarrow c'_0 \; c_1
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

contract : Redex u -> Closed u
contract (Lookup i env) = env ! i
contract (App f x env) =
  Clapp (Closure f env) (Closure x env)
contract (Beta body env arg) =
  Closure body (arg · 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?
Views: example

• 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

def example: Nat -> Bin

example n with evenOdd n

def example n (double k) | IsEven k
  = ...

example n (Succ (double k)) | IsOdd k
  = ...

Friday, October 7, 2011
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 : Context -> Set where
  MT : EvalContext Nil
  ARG : Closed u -> EvalContext Δ
       -> EvalContext (Cons u Δ)

_===>_ : List Ty -> Ty -> Ty
Nil ==> u = u
(Cons v Δ) ==> u = v => (Δ ==> u)
plug : EvalContext $\Delta$ $\rightarrow$ Closed ($\Delta \Longrightarrow u$)
  $\rightarrow$ Closed $u$
plug $\text{MT}$ $f = f$
plug ($\text{ARG}$ $x$ $\text{ctx}$) $f =$ plug $\text{ctx}$ ($\text{Clapp}$ $f$ $x$)
data Decomposition : Closed u -> Set where
  Val : (t : Closed u) -> isVal t
       -> Decomposition t
 Decompose : (r : Redex (Δ ==> v))
            -> (ctx : EvalContext Δ)
            -> Decomposition (plug ctx (fromRedex r))
Decompose

deAcc : (ctx : EvalContext Δ) (c : Closed (Δ ==> u)) ->
   Decomposition (plug ctx c)
decAcc MT (Closure (Lam body) env)
   = Val (Closure (Lam body) env) unit
decAcc (ARG x ctx) (Closure (Lam body) env)
   = Decompose (Beta body env x) ctx
decAcc ctx (Closure (App f x) env)
   = Decompose (App f x env) ctx
decAcc ctx (Closure (Var i) env)
   = Decompose (Lookup i env) ctx
decAcc ctx (Clapp f x) = decAcc (ARG x ctx) f

decompose : (c : Closed u) -> Decomposition c
decompose c = decAcc MT c
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

\[
evaluate : \text{Closed } u \rightarrow \text{Value } u
\]
\[
\text{evaluate } c = \text{iterate } (\text{decompose } c)
\]
\[
\text{where}
\]
\[
\text{iterate} : \text{Decomposition } c \rightarrow \text{Value } u
\]
\[
\text{iterate } (\text{Val } \text{val } p) = \text{Val } \text{val } p
\]
\[
\text{iterate } (\text{Decompose } r \text{ ctx})
\]
\[
= \text{iterate } (\text{decompose } (\text{plug } \text{ctx } (\text{contract } r)))
\]

Does not pass the termination check!
The Bove-Capretta method
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)
Nearly done

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

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

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

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

Yet we know that the simply typed lambda calculus is strongly normalizing...
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 : \((t : \text{Closed } u) \rightarrow \text{Reducible } (\text{headReduce } t) \rightarrow \text{Reducible } t\)

lemma2 : \((t : \text{Term } G u) (\text{env} : \text{Env } G) \rightarrow \text{ReducibleEnv } \text{env} \rightarrow \text{Reducible } (\text{Closure } t \text{ env})\)
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 pre-abstract machine;

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

• Prove that each transformation preserves the termination behaviour and semantics.
A term
A redex
Contract
Plug
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.

• Intuitively, refocusing behaves just the same as decompose . plug
Refocus summary

refocus : (ctx : EvalContext Δ) ->
  (c : Closed (Δ ==> u)) ->
  Decomposition (plug ctx c)

refocusCorrect : (ctx : EvalContext Δ) ->
  (c : Closed (Δ ==> u)) ->
  refocus ctx c == decompose (plug ctx c)
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} \; \Delta) \rightarrow
(t : \text{Term} \; \Gamma \; (\Delta \Rightarrow u)) \rightarrow
(env : \text{Env} \; \Gamma) \rightarrow \text{Value} \; u

\text{refocus} \; \text{ctx} \; (\text{Var} \; i) \; \text{env} =
\quad \text{let} \; \text{Closure} \; t \; \text{env}' = \text{lookup} \; i \; \text{env} \; q \; \text{in}
\quad \text{refocus} \; \text{ctx} \; t \; \text{env}'

\text{refocus} \; \text{ctx} \; (\text{App} \; f \; x) \; \text{env}
\quad = \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}
\quad = \text{refocus} \; \text{ctx} \; \text{body} \; (x \; \cdot \; \text{env})

\text{refocus} \; \text{MT} \; (\text{Lam} \; \text{body}) \; \text{env}
\quad = \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.
Proofs?

\[
\text{decomposePlug : (ctx : EvalContext us) ->} \\
\quad (c : \text{Closed (us ==> u)}) -> \\
\quad \text{decompose (plug ctx c) == decAcc ctx c} \\
\text{decomposePlug MT c = Refl} \\
\text{decomposePlug (ARG x ctx) t} \\
\quad \text{rewrite decomposePlug ctx (Clapp t x)} \\
\quad \mid \text{decomposeClapp ctx t x = Refl}
\]
refocusCorrect : (ctx : EvalContext D) (c : Closed (D ==> u)) ->
  refocus ctx c == decompose (plug ctx c)
refocusCorrect MT (Closure (Lam body) env) = Refl
refocusCorrect (ARG x ctx) (Closure (Lam body) env)
  rewrite decomposePlug ctx (Clapp (Closure (Lam body) env ) x)
  | decomposeClapp ctx (Closure (Lam body) env) x = Refl
refocusCorrect MT (Closure (Var i) env) = Refl
refocusCorrect (ARG x ctx) (Closure (Var i) env)
  rewrite decomposePlug ctx (Clapp (Closure (Var i) env) x)
  | decomposeClapp ctx (Closure (Var i) env) x = Refl
refocusCorrect MT (Closure (App f x) env) = Refl
refocusCorrect (ARG arg ctx) (Closure (App f x) env)
  rewrite decomposePlug ctx (Clapp (Closure (App f x) env) arg)
  | decomposeClapp ctx (Closure (App f x) env) arg = Refl
refocusCorrect MT (Clapp f x) = refocusCorrect (ARG x MT) f
refocusCorrect (ARG arg ctx) (Clapp f x)
  = refocusCorrect (ARG x (ARG arg ctx)) f
Proofs???
Lines of code vs. lines of proof

- In total, 585 lines of Agda (288/297)
- 50 lines Prelude (50/0)
- 100 lines data types & basics (100/0)
- 100 lines lemmas (0/100)
- 100 lines reducibility proof (33/67)
- 100 lines pre-abstract machine (25/75)
- 135 lines Krivine machine (80/55)
Conclusions

• You *can* reason about functions with non-trivial recursive behaviour in type theory.

• You can repeat the same trick on many other derivations of abstract machines.

• All our evaluators are *executable*; we do not need any additional postulates.