From Mathematics to Abstract Machine

Wouter Swierstra
MSFP 2012, Tallinn, Estonia
β 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 languages evaluate lambda terms using an *abstract machine* (tail-recursive function)
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 of the paper

• Define well-typed lambda terms;
• Implement a small step evaluator;
• Prove that it terminates;
• Apply program transformations to derive the Krivine machine.
Outline of the paper

• Sketch how one might define a terminating evaluator for the simply typed lambda calculus in Agda.

• What are the problems?

• What ‘design patterns’ help solve them?
Types

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

el : U -> Set
el 0 = Unit
el (s => t) = el s -> el t

Context : Set
Context = List Ty
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
Normalization-by-cheating

\[\text{eval} : \text{Env} \Gamma \to \text{Term} \Gamma u \to \text{el} u\]

\[\text{eval env} (\text{Lam} \ \text{body}) = \lambda x \to \text{eval} (\text{Cons} x \ \text{env}) \ \text{body}\]

\[\text{eval env} (\text{App} f x) = (\text{eval env} f) (\text{eval env} x)\]

\[\text{eval env} (\text{Var} i) = \text{lookup} i \ \text{env}\]
Closed terms only
\[ E ::= \square \mid E \cdot t \]

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

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

**Beta** \( E\{((\lambda t) [env]) \cdot c\} \rightarrow E\{t [c \cdot env]\} \)

Reduction rules
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)
Plan of attack

- Define one step of head reduction:
  - Decompose the term into a redex and evaluation context;
  - Contract the redex;
  - Plug the result back into the context.

- Iterated head reduction yields an evaluator.
- Prove termination.
Head reduction in three steps

- Decompose the term into a redex and evaluation context;
- Contract the redex;
- Plug the result back into the context.
Redex

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

\[\text{contract} : \text{Redex } u \rightarrow \text{Closed } u\]
\[\text{contract} (\text{Lookup } i \text{ env}) = \text{env} ! i\]
\[\text{contract} (\text{App } f \text{ x env}) =\]
\[\text{Clapp} (\text{Closure } f \text{ env}) (\text{Closure } x \text{ env})\]
\[\text{contract} (\text{Beta } \text{body env arg}) =\]
\[\text{Closure } \text{body} (\text{arg} \cdot \text{env})\]
Head reduction in three steps

☐ Decompose the term into a redex and evaluation context;

☑ Contract the redex;

☐ Plug the result back into the context.
Evaluation contexts

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

\[\text{plug} : \text{EvalContext } u \rightarrow v \rightarrow \begin{cases} \text{Closed } u \rightarrow \text{Closed } v \\
\text{MT } f = f \\
\text{plug } (\text{ARG } x \text{ ctx}) f = \text{plug } \text{ctx} (\text{Clapp } f x)\end{cases}\]
Head reduction in three steps

☐ Decompose the term into a redex and evaluation context;

☑ Contract the redex;

☑ Plug the result back into the context.
Decomposition as a view

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

• Define a view on closed terms.
data Decomposition : Closed u -> Set where
  Val : (t : Closed u) -> isVal t -> Decomposition t
  Decompose : (r : Redex v) -> (ctx : EvalContext u v) -> Decomposition (plug ctx (fromRedex r))
Decompose

decompose : (c : Closed u) =>
           Decomposition c
Head reduction in three steps

- Decompose the term into a redex and evaluation context;
- Contract the redex;
- Plug the result back into the context.
Head-reduction

\[
\text{headReduce} : \text{Closed } u \rightarrow \text{Closed } u
\]

\[
\text{headReduce } c \text{ with decompose } c
\]

\[
\ldots \mid \text{Val } \text{val } p = \text{val}
\]

\[
\ldots \mid \text{Decompose redex ctx}
\]

\[
= \text{plug ctx} \ (\text{contract redex})
\]
Plan of attack

- Define one step of head reduction:
  - Decompose the term into a redex and evaluation context;
  - Contract the redex;
  - Plug the result back into the context.
- Iterated head reduction yields an evaluator.
- Prove termination.
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

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

\[
\begin{align*}
\text{terminates}(v) \\
\hline
\text{terminates}(t') \\
\hline
\text{terminates}(t)
\end{align*}
\]
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
Plan of attack

☐ Define one step of head reduction:

☐ Decompose the term into a redex and evaluation context;

☐ Contract the redex;

☐ Plug the result back into the context.

☐ Iterated head reduction yields an evaluator.

☐ Prove termination.
Nearly done

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

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

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

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

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

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

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))
Finally, evaluation

evaluate : Closed u -> Value u
evaluate t =
    iterate (decompose t) (termination t)
Plan of attack

- Define one step of head reduction:
  - Decompose the term into a redex and evaluation context;
  - Contract the redex;
  - Plug the result back into the context.
- Iterated head reduction yields an evaluator.
- Prove termination.
What happened?

- Using typical programming idioms of dependently typed programming...
- Precise data types;
- Views;
- Bove-Capretta.
- ...you can define programs with non-trivial termination behaviour.
The Krivine machine

• Formalizing Biernacka & Danvy’s derivation of the Krivine machine is not so hard.

• Having an executable definition helps.
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.