From Mathematics to Abstract Machine

Wouter Swierstra

MSFP 2012, Tallinn, Estonia

β reduction

$$(\lambda\ x\ .\ t_0)\ t_1 \longrightarrow t_0\ \{t_1/x\}$$

Motivation

- Implementing β-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~~ talk

- 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
  O : Ty
  _=>_ : Ty -> Ty -> Ty

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

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
```

Normalization-by-cheating

```
eval : Env Γ -> Term Γ u -> el u

eval env (Lam body)

  = \x -> eval (Cons x env) body
eval env (App f x)

  = (eval env f) (eval env x)

eval env (Var i)
  = lookup i env
```

Closed terms only

$$E := \Box \mid E\ t$$

$$\textsc{Lookup} \qquad E\{i\,[c_1, c_2, \ldots c_n]\} \to E\{c_i\}$$

$$\textsc{App} \qquad E\{(t_0\ t_1)\,[env]\} \to E\{(t_0\,[env])\ (t_1\,[env])\}$$

$$\textsc{Beta} \qquad E\{((\lambda t)\,[env])\ c\} \to E\{t\,[c \cdot env]\}$$

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

```
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)
```

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

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

Head reduction in three steps

- [ ] Decompose the term into a redex and evaluation context;
- [x] Contract the redex;
- [x] 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.

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
```

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

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

Plan of attack

- [x] Define one step of head reduction:
  - [x] Decompose the term into a redex and evaluation context;
  - [x] Contract the redex;
  - [x] 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

```
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)))
```

The Bove-Capretta method

Bove-Capretta

$$\frac{}{\mathsf{terminates}(v)}$$

$$\frac{t \rightarrow t' \quad \mathsf{terminates}(t')}{\mathsf{terminates}(t)}$$

Bove-Capretta

```
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 action

- ☑ 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 : Closed u) -> Trace (decompose c)
```

Nearly done

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

```
(c : Closed u) -> Trace (decompose c)
```

Fail

Nearly done

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

```
(c : Closed u) -> Trace (decompose c)
```

Fail

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

Logical relation

```
Reducible : (u : Ty) -> (t : Closed u) -> Set
Reducible O 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.

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.