A formal derivation of an executable Krivine machine

Wouter Swierstra
Universiteit Utrecht

IFIP WG 2.1
Meeting 68; Rome, Italy

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 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
  O : 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

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

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

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

$$\textsc{Left} \qquad \text{if } c_0 \to c_0' \text{ then } c_0\ c_1 \to c_0'\ c_1$$

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)
```

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

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

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

$$\textsc{Left} \qquad \text{if } c_0 \to c_0' \text{ then } c_0\ c_1 \to c_0'\ c_1$$

Reduction rules

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

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

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

$$\cancel{\textsc{Left} \qquad \text{if } c_0 \to c_0' \text{ then } c_0\; c_1 \to c_0'\; c_1}$$

Reduction rules

$$E := \square \mid E\ t$$

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

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

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

Reduction rules

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)
```

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

```
example: Nat -> ...
example n with evenOdd n
example .(double k) | IsEven k
  = ...
example .(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
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
```

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))
```

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!

```
theorem : (c : Closed u) -> Reducible c
theorem (Closure t env)
  = lemma2 t env (envTheorem env)
theorem (Clapp f x)
  = snd (theorem f) x (theorem x)

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

...an easy corollary

Finally, evaluation

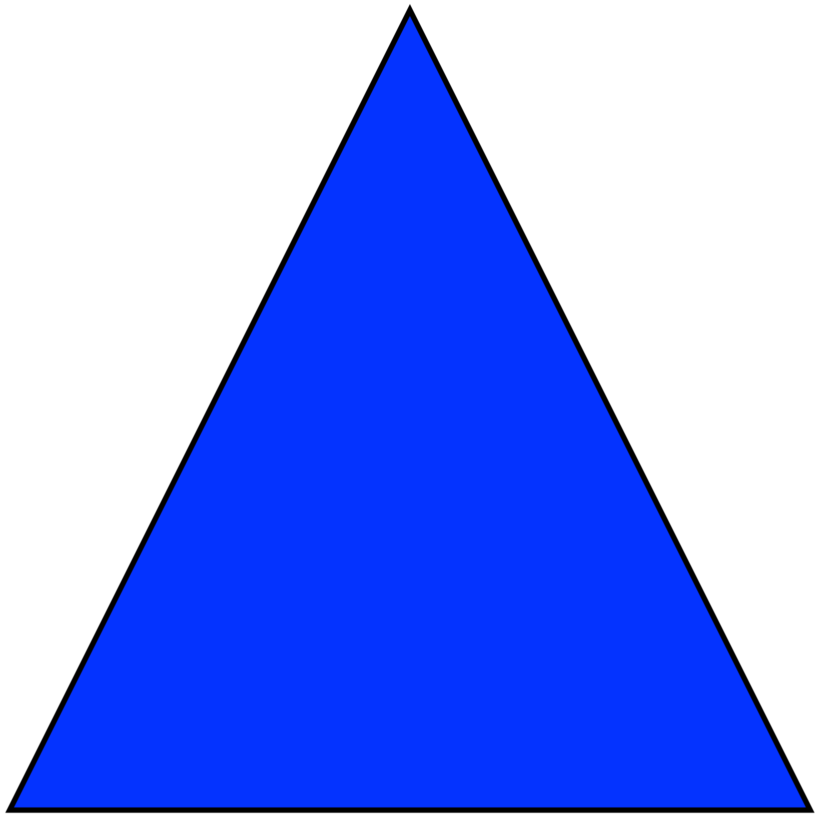
```
evaluate : Closed u -> Value u
evaluate t =
  iterate (decompose t) (termination t)
```

The story so far...

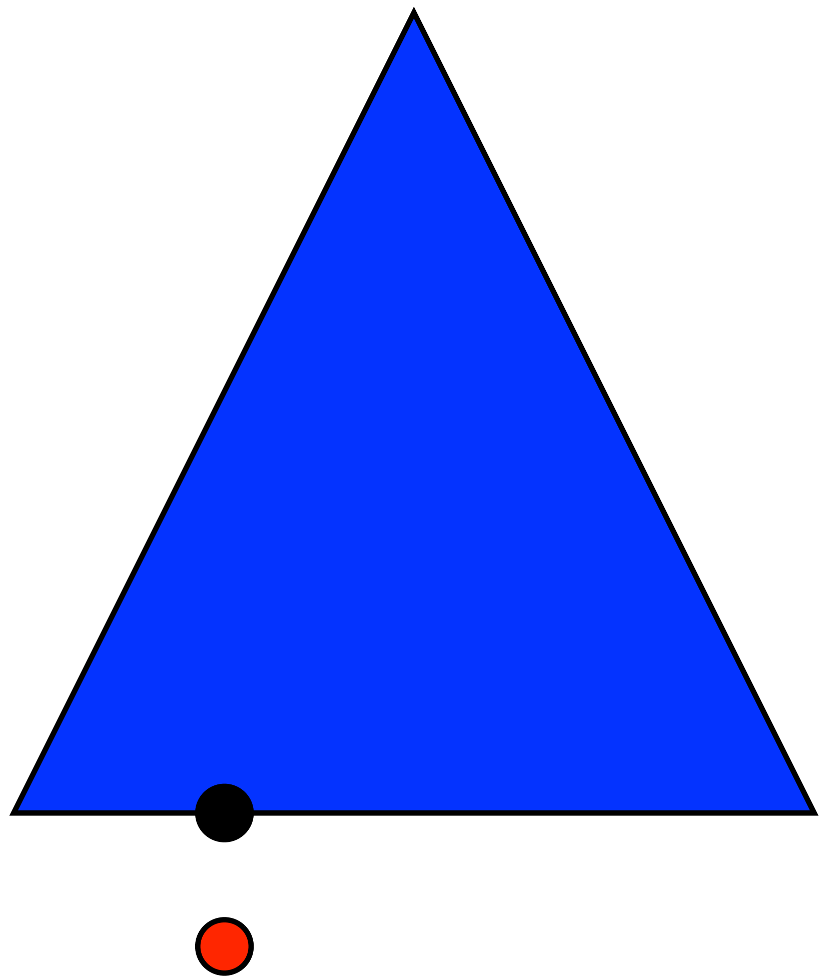
- Data types for terms, closed terms, values, redexes, evaluation contexts.
- Defined a three step head-reduction function: decompose, contract, plug.
- Proven that iterated head reduction yields a normal form...
- ... and used this to define a normalization function.

What's next?

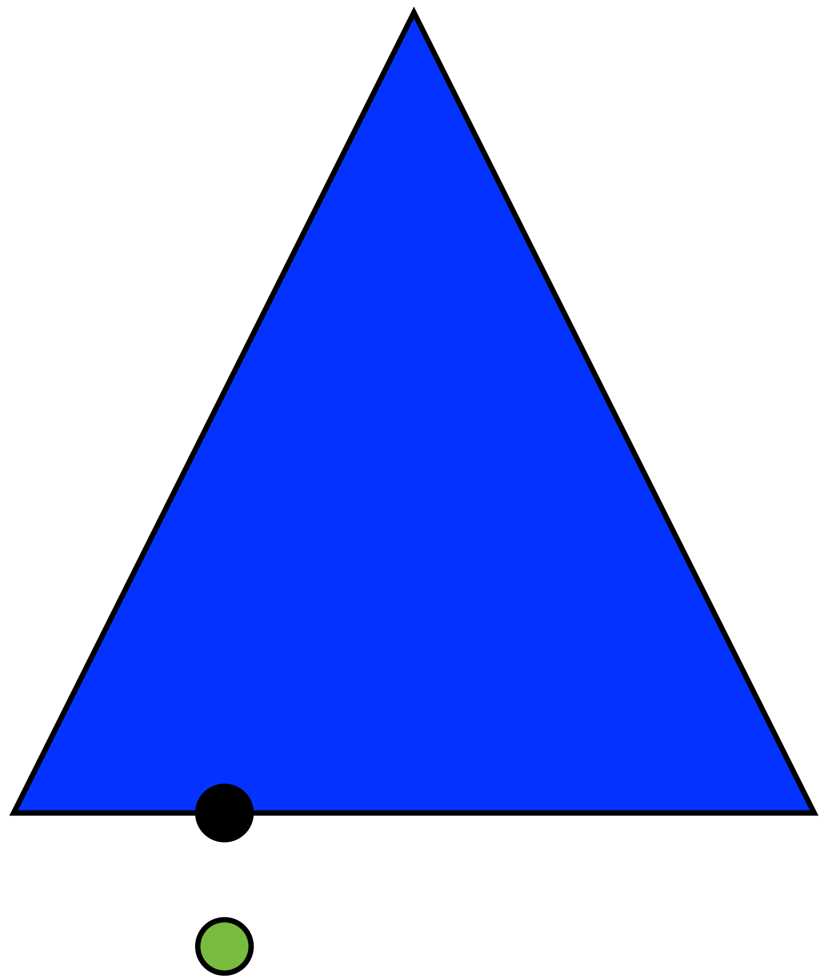
- Use Danvy & Nielsen's *refocusing* transformation to define a *small-step abstract machine*;
- Inline the `iterate` function (and one or two minor changes), yields the *Krivine abstract machine*.
- Prove that each transformation preserves the termination behaviour and semantics.



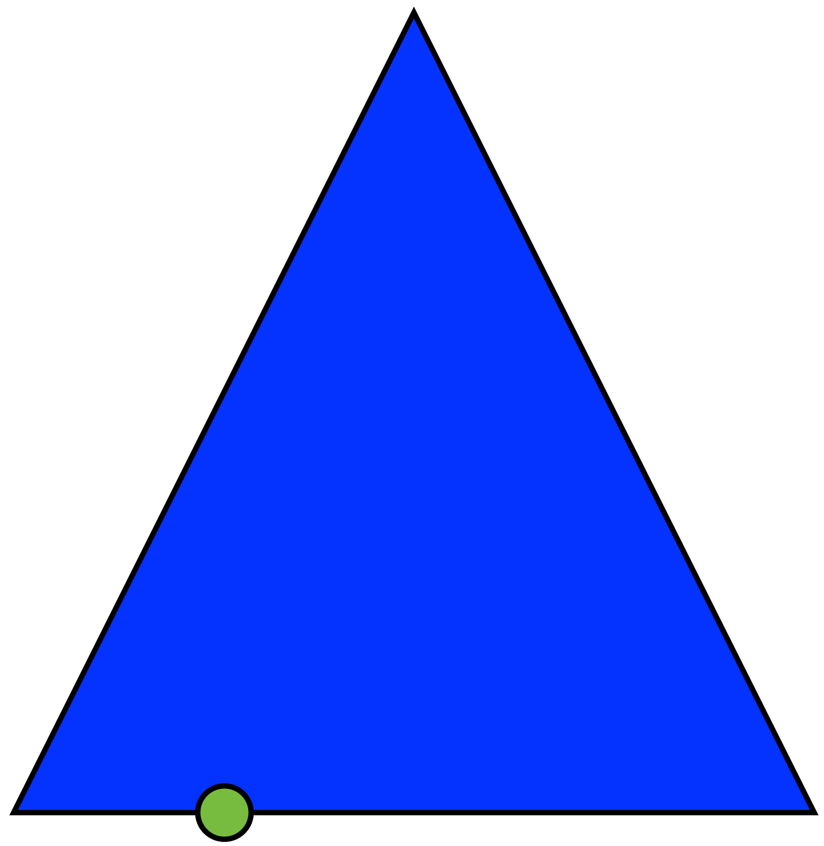
A term



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

```
refocus : (ctx : EvalContext u v) ->
   (c : Closed u) ->
   Decomposition (plug ctx c)

refocusCorrect : (ctx : EvalContext u v) ->
  (c : Closed u) ->
  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 :
 (ctx : EvalContext u v) ->
 (t : Term Γ u) ->
 (env : Env Γ) -> Value v
refocus ctx (Var i) env =
  let Closure t env' = lookup i env q in
  refocus ctx t env'
refocus ctx (App f x) env
  = refocus (ARG (Closure x env) ctx) f env
refocus (ARG x ctx) (Lam body) env
  = refocus ctx body (x · env)
refocus MT (Lam body) env
  = Val (Closure (Lam body) 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.