

From Math to Machine

A formal derivation of an executable Krivine Machine

Wouter Swierstra
Brouwer Seminar

β reduction

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

Substitution

- Realizing β -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

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  $\Gamma$ ) v
        -> Term  $\Gamma$  (u => v)
  App : Term  $\Gamma$  (u => v) -> Term  $\Gamma$  u
        -> Term  $\Gamma$  v
  Var : Ref  $\Gamma$  u -> Term  $\Gamma$  u
```

Closed terms

```
data Closed : Ty -> Set where
```

```
  Closure : Term  $\Gamma$  u -> Env  $\Gamma$   
          -> Closed u
```

```
  Clapp : Closed (u ==> v) -> Closed u  
        -> Closed v
```

```
data Env : Context -> Set where
```

```
  Nil : Env Nil
```

```
  _·_ : Closed u -> Env  $\Gamma$   
      -> Env (Cons u  $\Gamma$ )
```

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
```

LOOKUP $i [c_1, c_2, \dots, c_n] \rightarrow c_i$

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 $c_0 c_1 \rightarrow c'_0 c_1$

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  $\Gamma$  u -> Env  $\Gamma$  -> Redex u
  App : Term  $\Gamma$  (u => v) -> Term  $\Gamma$  u
        -> Env  $\Gamma$  -> Redex v
  Beta : Term (Cons u  $\Gamma$ ) v -> Env  $\Gamma$ 
        -> 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 -> Bin
```

```
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 : 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

```
plug : EvalContext  $\Delta$  -> Closed ( $\Delta \implies u$ )  
      -> Closed u
```

```
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 ( $\Delta \implies v$ ))
              -> (ctx : EvalContext  $\Delta$ )
              -> Decomposition (plug ctx (fromRedex r))
```

Decompose

```
decAcc : (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` : `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`)))



Does not pass the termination check!



The Bove-Capretta method

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...

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

Nearly done

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

(c : Closed u) → Tr c (decompose c)

Fail

Nearly done

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

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

Fail

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

Logical relation

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

`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 =`

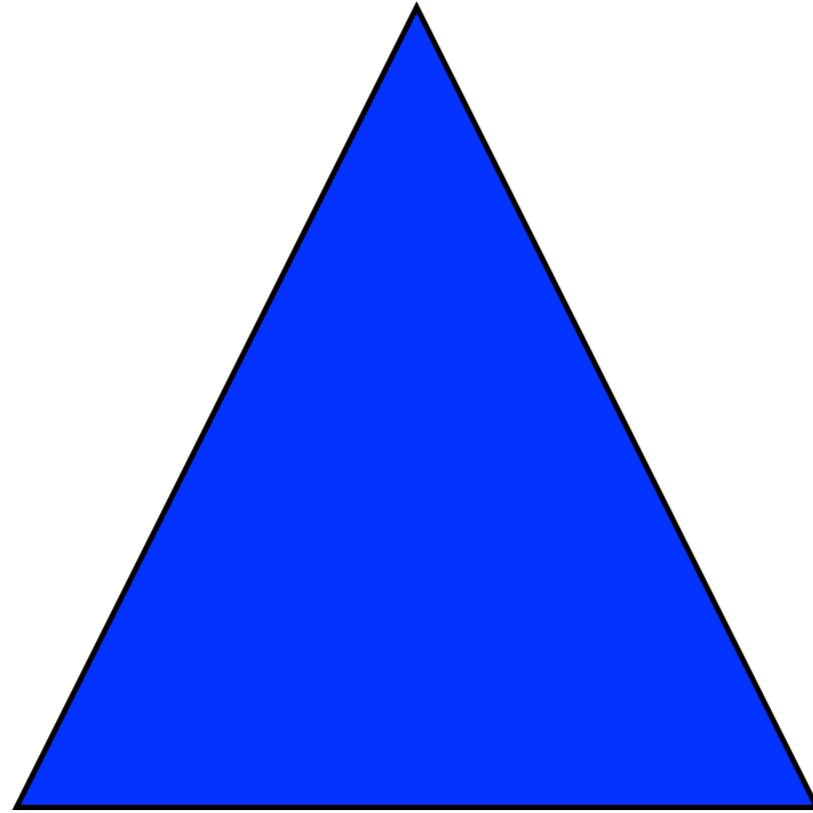
`iterate (decompose t) (termination t)`

The story so far...

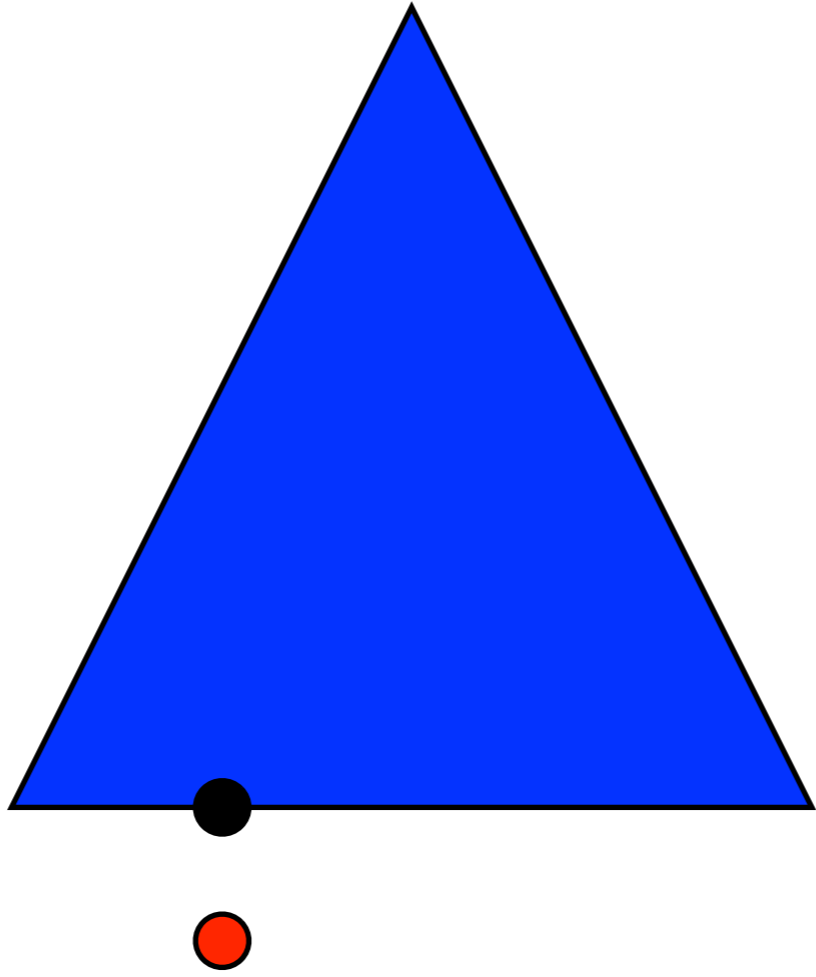
- Data types for terms, closed terms, values, redexes, evaluation contexts.
- Defined a three step head-reduction function: decompose, contract, plug.
- Proven that iterated head reduction yields a normal form...
- ... and used this to define a normalization function.

What's next?

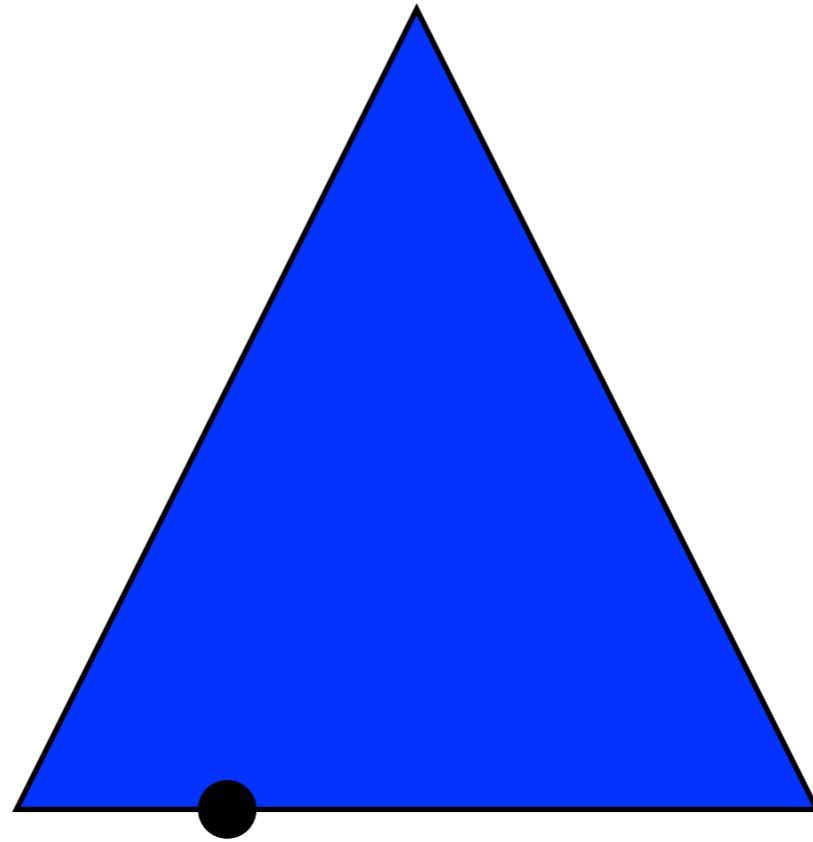
- Use Danvy & Nielsen's *refocusing* transformation to define a pre-abstract machine;
- Inline the *iterate* function (and one or two minor changes), yields the Krivine machine.
- Prove that each transformation preserves the termination behaviour and semantics.



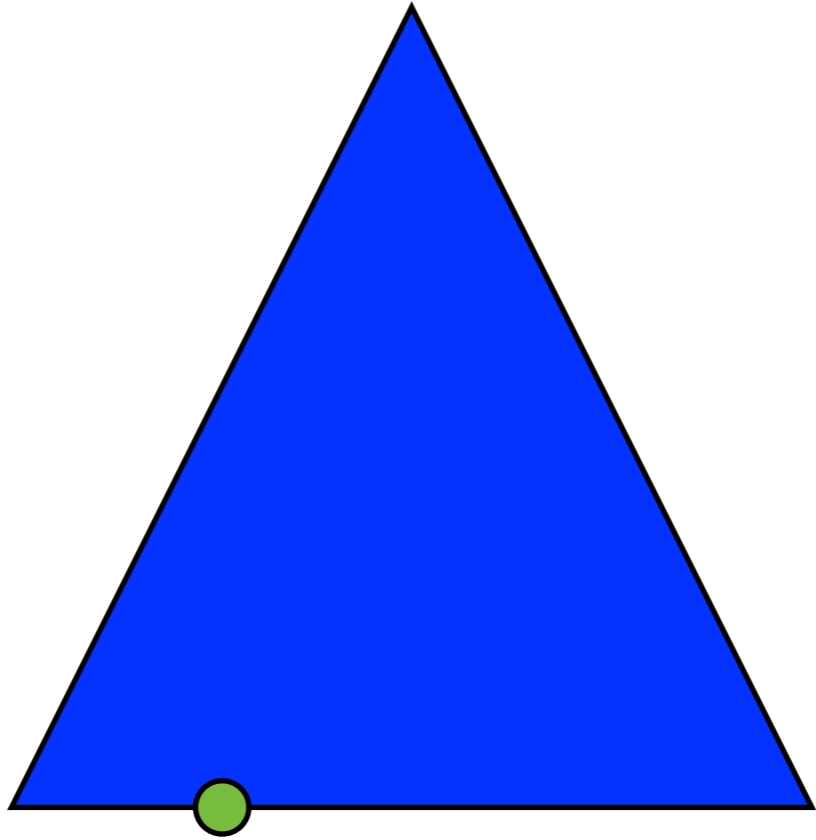
A term



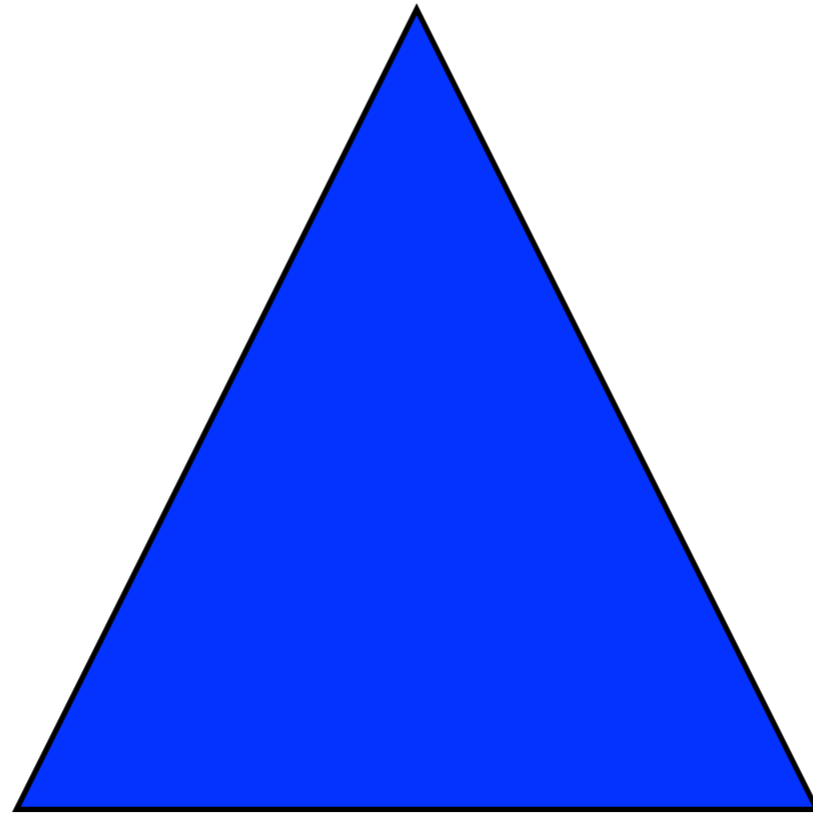
A redex



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  $\Delta$ ) ->  
  (c : Closed ( $\Delta \implies u$ )) ->  
  Decomposition (plug ctx c)
```

```
refocusCorrect : (ctx : EvalContext  $\Delta$ ) ->  
  (c : Closed ( $\Delta \implies u$ )) ->  
  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 :  
  (ctx : EvalContext Δ) ->  
  (t : Term Γ (Δ ==> u)) ->  
  (env : Env Γ) -> Value u  
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.

Proofs?

```
decomposePlug : (ctx : EvalContext us) ->
  (c : Closed (us ==> u)) ->
    decompose (plug ctx c) == decAcc ctx c
decomposePlug MT c = Refl
decomposePlug (ARG x ctx) t
  rewrite decomposePlug ctx (Clapp t x)
  | decomposeClapp ctx t x = Refl
```

Proofs??

```
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???

```
termination : (t : Term Nil u) -> Trace t Nil MT
termination t = lemma MT t Nil (unit , unit) (Section5.termination (Closure t Nil))
where
lemma : (ctx : EvalContext D) (t : Term G (D ==> u)) (env : Env G) -> (invariant ctx env) ->
  Section5.Trace (Section5.refocus ctx (Closure t env)) -> Trace t env ctx
lemma MT (Lam body) env p (Section5.Done .(Closure (Lam body) env) .(record { }))) = Done body
lemma MT (App f x) env p (Section5.Step y0) = App f x (lemma (ARG (Closure x env) MT) f env ((fst p) , p) y0)
lemma MT (Var i) env (p1 , p2) (Section5.Step y') rewrite lookupClosure env i p1
  = Lookup i p1 (lemma MT (getTerm (lookup i env p1)) (getEnv (lookup i env p1)) (lookupLemma env i p1 , p2) y')
lemma (ARG (Closure arg env') ctx) (Lam body) env (p1 , (p2 , p3)) (Section5.Step step)
  = Beta ctx arg env' body (lemma ctx body ((Closure arg env') · env) ((p2 , p1) , p3) step)
lemma (ARG (Clapp y y') ctx) (Lam y0) env (y1 , ()) (Section5.Step y3)
lemma (ARG arg ctx) (App (Lam y) x) env p (Section5.Step step)
  = App ((Lam y)) x (lemma (ARG (Closure x env) (ARG arg ctx)) (Lam y) env ((fst p) , p) step)
lemma (ARG arg ctx) (App (App f y) x) env p (Section5.Step step)
  = App (App f y) x (lemma (ARG (Closure x env) (ARG arg ctx)) (App f y) env ((fst p) , p) step)
lemma (ARG arg ctx) (App (Var i) x) env p (Section5.Step step)
  rewrite lookupClosure env i (fst p)
  = let c = lookup i env (fst p) in
    App (Var i) x (lemma (ARG (Closure x env) (ARG arg ctx)) (Var i) env ((fst p) , p) step)
lemma (ARG arg ctx) (Var i) env p (Section5.Step step)
  rewrite lookupClosure env i (fst p)
  = let c = lookup i env (fst p) in
    Lookup i (fst p) (lemma (ARG arg ctx) (getTerm c) (getEnv c) ((lookupLemma env i (fst p)) , (snd p)) step)
```

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.