Formally verified derivation of an executable and terminating CEK machine from call-by-value $\lambda\hat{p}$-calculus

Wojciech Rozowski

Programming Principles, Logic and Verification Group,
University College London

January 12, 2022

Overview

Motivation

- Formal semantics of functional programming languages like Haskell/Lisp/OCaml are based on the variants of λ-calculus.
- Practical implementation of interpreters for those languages are **based on abstract machines**, rather than higher-order functions implementing the operational semantics.
- Abstract machines are mathematical models used to describe formal semantics of programming languages, as **first-order transition systems**
- Known examples of abstract machines include Krivine machine, **CEK**, STG or SECD

CEK machine

Code + **E**nvironment + **K**ontinuation

$$\varsigma \longmapsto_{CEK} \varsigma'$$

$\langle x, \rho, \kappa \rangle$	$\langle v, \rho', \kappa \rangle$ where $\rho(x) = (v, \rho')$
$\langle (e_0 e_1), \rho, \kappa \rangle$	$\langle e_0, \rho, \mathbf{ar}(e_1, \rho, \kappa) \rangle$
$\langle v, \rho, \mathbf{ar}(e, \rho', \kappa) \rangle$	$\langle e, \rho', \mathbf{fn}(v, \rho, \kappa) \rangle$
$\langle v, \rho, \mathbf{fn}((\lambda x.e), \rho', \kappa) \rangle$	$\langle e, \rho'[x \mapsto (v, \rho)], \kappa \rangle$

Figure 1. The CEK machine.

Deriving abstract machines

- Abstract machines can be derived, rather than invented - Biernacka & Danvy obtained Krivine machine and CEK using Danvy's refocusing transform
- Curien's $\lambda\rho$-calculus has closures, similarly to SECD machine
- Biernacka & Danvy introduced $\lambda\hat{\rho}$-calculus - a more expressive variant of $\lambda\rho$ that can encompass small-step reduction

Proofs-as-programs and dependent types

- By Curry-Howard correspondence types correspond to logic - for example: tuples correspond to conjunction, and function types correspond to implication
- **Dependent types** are a more expressive system in which types can depend on values, and therefore types correspond to quantifiers known from propositional logic
- Languages with **dependent types** are expressive enough to perform **internal verification** and write code which is correct by specification
- **Agda** is an example of such a language

Agda example

```
data Nat : Set where
  Zero : Nat
  Suc : Nat -> Nat

_+_ : Nat -> Nat -> Nat
Zero + y = y
Suc x + y = Suc (x + y)

id : ∀ (x : Nat) -> x + Zero ≡ x
id Zero = refl
id (Suc x) = cong Suc (id x)

assoc : ∀ (x : Nat) (y : Nat) (z : Nat) -> x + (y + z) ≡ (x + y) + z
assoc Zero y z = refl
assoc (Suc x) y z = cong Suc (assoc x y z)
```

Formalising abstract machines

- It's valuable to have the derivation of abstract machine, checked by a computer
- We can do this using dependently-typed languages, like Agda or Coq
- Biernacka, Sieczkowski and Zielińska formalised the refocusing transform and showed the derivation of multiple machines
- They shown that refocusing leads to correct specifications, however their machines are not executable
- Independently from that Wouter Swierstra formalised call-by-name $\lambda\hat{\rho}$ and obtained executable and terminating Krivine machine for STLC
- Can this research by adapted to obtain other machines than Krivine machine?

Our contributions

- We extend Swierstra's formalisation of $\lambda\hat{p}$ to call-by-value case, including the properties of head reduction.
- We provide a proof of a Strong Normalisation property for call-by-value $\lambda\hat{p}$-calculus using Tait-style logical relation.
- We provide a constructive proof of equivalence of the obtained CEK machine with call-by-value $\lambda\hat{p}$.

Simply Typed λ-calculus with De Brujin indices

Abstraction (λ) $\dfrac{(\sigma :: \Gamma) \vdash \tau}{\Gamma \vdash (\sigma \Rightarrow \tau)}$

Application (∘) $\dfrac{\Gamma \vdash (\sigma \Rightarrow \tau) \quad \Gamma \vdash \sigma}{\Gamma \vdash \tau}$

Variable (‘) $\dfrac{\Gamma \ni \sigma}{\Gamma \vdash \sigma}$

Unit type $\dfrac{}{\bullet : \mathit{Type}}$

Arrow type $\dfrac{a : \mathit{Type} \quad b : \mathit{Type}}{a \Rightarrow b : \mathit{Type}}$

Zero (Z) $\dfrac{}{(\sigma :: \Gamma) \ni \sigma}$

Successor (S) $\dfrac{\Gamma \ni \sigma}{(\tau :: \Gamma) \ni \sigma}$

$\lambda\hat{p}$-calculus

- There is a close correspondence between **abstract machines** and **calculi with explicit subsitutions**

`Closed u` Closed term of type u

`Env Γ` Substitution environment for type context Γ

$$\frac{\texttt{t} : \Gamma \vdash u \quad \texttt{e} : \texttt{Env}\ \Gamma}{\texttt{Closure t e} : \texttt{Closed u}} \quad (1)$$

$$\frac{\texttt{f} : \texttt{Closed (u} \Rightarrow \texttt{v)} \quad \texttt{x} : \texttt{Closed u}}{\texttt{Clapp f x} : \texttt{Closed v}} \quad (2)$$

$$\frac{\texttt{c} : \texttt{Closed u} \quad \texttt{e} : \texttt{Env}\ \Gamma}{c \cdot e : \texttt{Env}\ (u :: \Gamma)} \quad (3)$$

$$\frac{}{\texttt{Nil} : \texttt{Env []}} \quad (4)$$

Biernacka & Danvy three step reduction

1. Traverse the AST to find redex (decompose)
2. Contract redex
3. Plug result of contraction to the original AST

Call-by-value redexes

$$\text{Lookup}\ \frac{\Gamma \ni \sigma \quad \texttt{Env}\ \Gamma}{\texttt{Redex}\ \sigma}$$

$$\text{LOOKUP}\ \frac{}{i[c_1 \ldots c_m] \to c_i}$$

$$\text{App}\ \frac{\Gamma \vdash (\sigma \Rightarrow \tau) \quad \Gamma \vdash \sigma \quad \texttt{Env}\ \Gamma}{\texttt{Redex}\ \tau}$$

$$\text{APP}\ \frac{}{(t_0\, t_1)[s] \to (t_0[s])(t_1[s])}$$

$$\text{Beta}\ \frac{(\sigma :: \Gamma) \vdash \tau \quad \texttt{Env}\ \Gamma \quad \texttt{Value}\ \sigma}{\texttt{Redex}\ \tau}$$

$$\text{BETA}\ \frac{}{((\lambda t)[s])\mathbf{v} \to t[\mathbf{v} \cdot s]}$$

Contraction

```
contract : ∀ {u}
         → 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 (Val c x)) = Closure body (c · env)
```

Evaluation contexts

$$\text{MT}\;\frac{}{\texttt{EvalContext u u}}$$

$$\text{ARG}\;\frac{\texttt{Closed u} \qquad \texttt{EvalContext v w}}{\texttt{EvalContext (u} \Rightarrow \texttt{v) w}}$$

$$\text{FN}\;\frac{\texttt{Value (a} \Rightarrow \texttt{b)} \qquad \texttt{EvalContext b c}}{\texttt{EvalContext a c}}$$

$$\text{LEFT}\;\frac{c_0 \to c_0'}{(c_0 c_1) \to (c_0' c_1)}$$

$$\text{RIGHT}\;\frac{c_1 \to c_1'}{(v c_1) \to (v c_1')}$$

Example traversal

$$\left(\underset{\text{v} \quad \text{x}}{\text{Clapp}}, \textbf{MT}\right) \longrightarrow \left(\text{v}, \textbf{ARG x MT}\right) \longrightarrow \left(\text{x}, \textbf{FN v MT}\right)$$

Figure: Visiting the left hand side first and then switching to the right side

Plugging

```
plug : ∀ {u v}
       → EvalContext u v
       → Closed u
       → Closed v

plug MT f = f
plug (ARG x ctx) f = plug ctx (Clapp f x)
plug (FN (Val closed isval) ctx) x = plug ctx (Clapp closed x)
```

Decomposition type - sourced from Swierstra (2012)

$$\text{Val}\ \frac{\texttt{(body : (u :: }\Gamma\texttt{)} \vdash \texttt{v)} \quad \texttt{(env : Env }\Gamma\texttt{)}}{\texttt{Decomposition (Closure (}\lambda\texttt{ body) env)}}$$

$$\text{Redex}\times\text{Context}\ \frac{\texttt{(r : Redex u)} \quad \texttt{(ctx : EvalContext u v)}}{\texttt{Decomposition (plug ctx (fromRedex r))}}$$

Figure: Valid decompositions of a closed term

Decomposition function

```
decompose' : ∀ { u v}
           → (ctx : EvalContext u v)
           → (c : Closed u)
           → Decomposition (plug ctx c)

decompose' ctx (Closure (` i) env) =
  RedexxContext (Lookup i env) ctx

decompose' ctx (Closure (ƛ body) env) =
  decompose'_aux ctx (body) env
decompose' ctx (Closure (f ∘ x) env) =
  RedexxContext (App f x env) ctx
decompose' ctx (Clapp f x) =
  decompose' (ARG x ctx) f
```

Decomposition function

```
-- The auxillary function peels of the lambda closure basing on the continuation frame
decompose'_aux : ∀ { a b w Γ}
               → (ctx : EvalContext (a ⇒ b) w)
               → (body : (a :: Γ) ⊢  b)
               → (env : Env Γ)
               → Decomposition (plug ctx (Closure (ƛ body) env))

decompose'_aux MT body env = Val body env
decompose'_aux (ARG arg ctx) body env = decompose' (FN (Val (Closure (ƛ body) env) tt) ctx) arg
decompose'_aux (FN (Val (Closure (ƛ x) env₂) proof) ctx) body env =
  RedexxContext (Beta x env₂ (Val (Closure (ƛ body) env) tt)) ctx
```

Decomposition function

Kick off with an empty evaluation context

```
decompose : ∀ {u}
          → (c : Closed u)
          → Decomposition c

decompose c = decompose' MT c
```

Small-step reduction

decompose → contract → plug

```
headReduce : ∀ {u}
           → Closed u
           → Closed u

headReduce c with decompose c
headReduce .(Closure (ƛ body) env) | Val body env =
  Closure (ƛ body) env
headReduce .(plug ctx (fromRedex redex)) | RedexxContext redex ctx =
  plug ctx (contract redex)
```

Refocusing theorem

Theorem (Refocusing theorem)

For any types `u` and `v` let `c` denote closed term of type `u` and let `ctx` denote an EvalContext parametrised by types `u` and `v`. We have `decompose (plug ctx c)` $\equiv$ `decompose' ctx c`

Let call `refocus = decompose'` Instead of:

$$\texttt{decompose} \longrightarrow \texttt{contract} \longrightarrow \texttt{plug}$$

We have:

$$\texttt{refocus} \longrightarrow \texttt{contract}$$

Showing termination for well-typed terms

- In general, programming languages can be diverging - type theory formalisations of evaluators as executable functions come at a price of showing termination
- Even after we restrict ourselves to well-behaved subset which is strongly normalising, showing termination is still quite daunting task, as we deal with **non-structurally recursive functions**
- To convince termination checker, we use **Bove-Capretta method** of presenting the execution trace
- We obtain the trace as the witness of the Strong Normalisation, which is proved using Tait-style logical relation.

Bove-Capretta trace - sourced from Swierstra (2012)

$$\text{Done}\,\frac{\begin{array}{c}\texttt{(body : (u :: }\Gamma\texttt{)} \vdash \texttt{v)}\\ \texttt{(env : Env }\Gamma\texttt{)}\end{array}}{\texttt{Trace (Val body env)}}$$

$$\text{Step}\,\frac{\begin{array}{c}\texttt{\{r : Redex u\}}\\ \texttt{\{ctx : EvalContext u v\}}\\ \texttt{Trace (decompose (plug ctx (contract r)))}\end{array}}{\texttt{Trace (Redex}\times\texttt{Context r ctx)}}$$

Figure: Definition of Bove-Capretta trace for repeated head reduction evaluator

Can we obtain trace for any well-typed term?

Q

Do we have a straightforward implication $\forall_{\mathtt{u}}(\mathtt{c} : \mathtt{Closed\ u}) \to \mathtt{Trace\ (decompose\ c)}$?

A

No, straightforward induction is too weak. We neeed to prove something stronger than that.

Tait (1967) strikes again

- There is a classic answer due to Tait (1967) to that coming from the problem of normalisation of proofs. Use logical relation instead of going for induction.
- Successfully worked for STLC- see Girard (1989)
- Formalised by Altenkirch and Chapman (2009) for System-T - they used Bove-Capretta too
- Swierstra (2012) did it for call-by-name $\lambda\hat{p}$-calculus
- How do we adapt it to call-by-value $\lambda\hat{p}$-calculus?

Logical relation for $\lambda\hat{p}$

Reducibility relation

We define a set `Reducible u` (reducible closed terms of type `u`) by induction on the types.

- For `c` of type $\bullet$, c belongs to `Reducible u`, if c is strongly normalising
- For `c` of type $a \Rightarrow b$, is reducible, if for any closed term `d` of type `a` which belongs to `Reducible a`, `Clapp c d` belongs to `Reducible b`

Environment reducibility relation

For the `Nil` constructor, an environment trivially belongs to `RedEnv` For the constructor case, an environment is reducible if the closure in the head position belongs to the `Reducible` relation of the appropriate type and the tail of the enviroment belongs to `RedEnv`

Reduction preserves types

Remark: $\rightsquigarrow$ means single step, not a transitive closure

Preservation lemma

If $e \rightsquigarrow e'$ and e' is reducible, then e is reducible

Backwards preservation lemma

If $e \rightsquigarrow e'$ and e is reducible, then e' is reducible

The first one allows us to build up the trace from bottom to top, by single-step increments. In case of arrow type, the second part of cartesian product inductively appeals to themselves

Reducing a closure of well-typed term

Call-by-value closure reducibility lemma

Closure of a well-typed term with a reducible environment is always reducible

Proof

- Variable lookup case - after reducing it becomes a closure from the environment. Use preservation lemma, and the witness that environment is reducible.
- Application case - closure of application becomes application of two closures. Inductively obtain reducibility of lhs and rhs and then use preservation lemma.
- Lambda abstraction, nothing much for the trace as we are done. However, this is a function type - we need to show that given a trace of rhs - application of lambda to rhs is still reducible. It is trivial in call-by-name case. We prove this property on the next slide.

Reducing right hand side

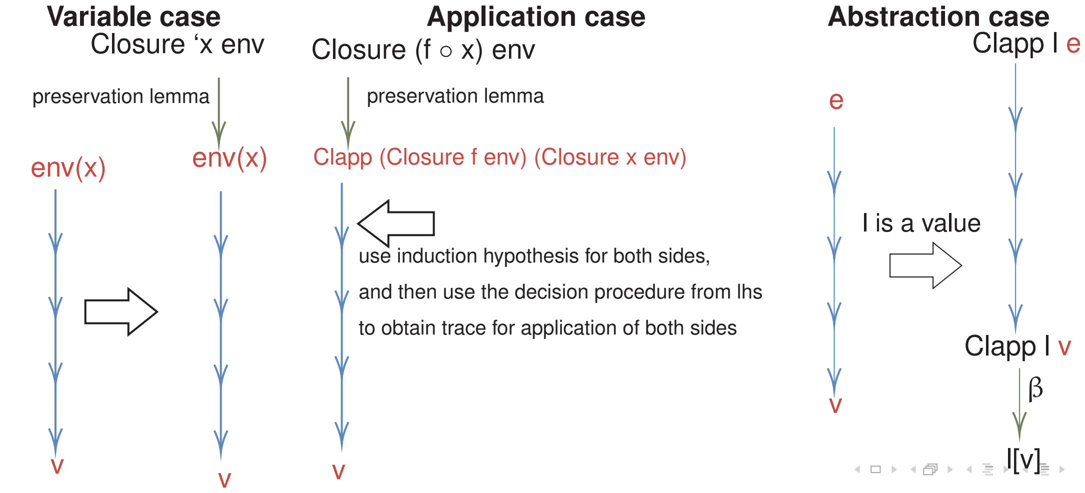
Right hand side reducibility lemma

For all Γ, σ and τ, let `body` denote a term $(\sigma :: \Gamma) \vdash \tau$ and let `env` denote a substitution environment for typing context Γ, which is reducible. Let `x` denote a closed term of type σ. Finally let `trace` denote a Bove-Capretta trace of decomposition of `x`. If `x` is reducible, then so is `(headReduce (Clapp (Closure (λ body) env) x)`

Proof

- x is a value - appeal to preservation lemma - it is the only case in call-by-name situation as we always perform β-reduction
- x is not a value - inductive call using backwards substitution lemma. But in case of `Clapp v x` how do we show that reducibility of x implies reducibility of `Clapp v x`?

Strong Normalisation of call-by-value $\lambda\hat{p}$-calculus



Basically done

- Empty substitution environment is trivially reducible
- So, we can obtain a reducibility predicate for any term without free variables
- From reducibility predicate, we can obtain Bove-Capretta trace
- Given Bove-Capretta trace - we obtain a terminating evaluator
- Because of refocusing lemma - we can rewrite the trace and simplify evalautor

Leftmost innermost head reduction properties

Left hand side head reduction lemma

For any types `u` and `v` let `f` denote a closed term of type `u` $\Rightarrow$ `v`, let `x` denote a closed term of type `u` and `fx` denote a closed term of type `v` such that `Clapp f x` $\equiv$ `fx` and `f` is not a value. We have the equality `headReduce fx` $\equiv$ `Clapp (headReduce f) x`

Right hand side head reduction lemma

For any types `u` and `v` let `f` denote a closed term of type `u` $\Rightarrow$ `v`, let `x` denote a closed term of type `u` and `fx` denote a closed term of type `v` such that `Clapp f x` $\equiv$ `fx` and `f` is a value. If x is not a value then `headReduce fx` $\equiv$ `Clapp f (headReduce x)`

Proof: decomposing one of the sides yields the same redex and the only difference in the contexts is the last non-MT frame.

Inlining the contract function gives the CEK transition function

```
refocus kont .(` i) env (Lookup i p trace) =
  let c = (lookup i env p) in refocus kont (getTerm c) (getEnv c) trace
  refocus .MT .(λ body) env (Done body) = Val (Closure (λ body) env) tt
refocus kont .(f ∘ x) env (Left f x trace) =
  refocus (ARG (Closure x env) kont) f env trace
refocus (ARG (Closure x argEnv) kont) .(λ body) env (Right .env body x trace) =
  refocus (FN (Val (Closure (λ body) env) tt) kont) x argEnv trace
refocus (FN (Val (Closure (λ body) env₂) tt) ctx) .(λ argBody) env (Beta ctx argBody .env body trace) =
  refocus ctx body (Closure (λ argBody) env · env₂) trace
```

Bove-Capretta trace for the CEK machine

$$\text{Done}\ \frac{\{\texttt{env : Env}\ \Gamma\}\quad (\texttt{body : }(\texttt{v :: }\Gamma) \vdash \texttt{u})}{\texttt{Trace }(\lambda\texttt{ body) env MT}}$$

$$\text{Lookup}\ \frac{\{\texttt{ctx : EvalContext u v}\}\{\texttt{env : Env}\ \Gamma\}\quad (\texttt{i : }\Gamma \ni \texttt{u})(\texttt{p : isValidEnv env})\quad \texttt{Trace (getTerm (lookup i env p)) (getEnv (lookup i env p)) ctx}}{\texttt{Trace (` i) env ctx}}$$

$$\text{Left}\ \frac{\{\texttt{env : Env}\ \Gamma\}\{\texttt{ctx : EvalContext v w}\}\quad (\texttt{f : }\Gamma \vdash (\texttt{u} \Rightarrow \texttt{v}))(\texttt{x : }\Gamma \vdash \texttt{u})\quad \texttt{Trace f env (ARG (Closure x env) ctx)}}{\texttt{Trace (f}\circ\texttt{x) env ctx}}$$

Bove-Capretta trace for the CEK machine - continued

$$\text{Right}\ \frac{\begin{array}{c}\{\texttt{env} : \texttt{Env}\ \Gamma\}\{\texttt{ctx} : \texttt{EvalContext v w}\}\\ (\texttt{env2} : \texttt{Env}\ \Delta)(\texttt{body} : (\texttt{u} :: \Delta) \vdash \texttt{v})(\texttt{x} : \Gamma \vdash \texttt{u})\\ \texttt{Trace x env (FN (Val (Closure (}\lambda\texttt{ body) env2) tt) ctx)}\end{array}}{\texttt{Trace (}\lambda\texttt{ body) env2 (ARG (Closure x env) ctx)}}$$

$$\text{Beta}\ \frac{\begin{array}{c}\{\texttt{env} : \texttt{Env}\ \Gamma\}(\texttt{ctx} : \texttt{EvalContext u w})(\texttt{argBody} : (\texttt{a} :: \Delta) \vdash \texttt{b})\\ (\texttt{argEnv} : \texttt{Env}\ \Delta)(\texttt{body} : ((\texttt{a} \Rightarrow \texttt{b}) :: \Gamma) \vdash \texttt{u})\\ \texttt{Trace body (Closure (}\lambda\texttt{ argBody) argEnv} \cdot \texttt{env) ctx}\end{array}}{\texttt{Trace (}\lambda\texttt{ argBody) argEnv (FN (Val (Closure (}\lambda\texttt{ body) env) tt) ctx)}}$$

CEK machine

- We can obtain CEK trace from refocusing and small-step evaluators trace
- We use that to show termination and correctness
- No closure making step - simpler than in Felleisen's presentation of the rules
- Executable and terminating

CEK machine

- We can obtain CEK trace from the single-step evaluator trace
- We use it to prove correctness and termination

```
refocus kont .(` i) env (Lookup i p trace) =
  let c = (lookup i env p) in refocus kont (getTerm c) (getEnv c) trace
  refocus .MT .(ƛ body) env (Done body) = Val (Closure (ƛ body) env) tt
refocus kont .(f ∘ x) env (Left f x trace) =
  refocus (ARG (Closure x env) kont) f env trace
refocus (ARG (Closure x argEnv) kont) .(ƛ body) env (Right .env body x trace) =
  refocus (FN (Val (Closure (ƛ body) env) tt) kont) x argEnv trace
refocus (FN (Val (Closure (ƛ body) env₂) tt) ctx) .(ƛ argBody) env (Beta ctx argBody .env body trace) =
  refocus ctx body (Closure (ƛ argBody) env · env₂) trace
```

Future work

- Parigiot $\lambda\mu$ calculus
- Biernacka & Biernacki context based Tait-style relation

Acknowledgements

Julian Rathke

Wouter Swierstra

Thorsten Altenkirch

References

Wouter Swierstra (2012)
From Mathematics to Abstract Machine

Biernacka & Danvy (2007)
A concrete framework for environment machines

Altenkrich & Chapman (2007)
Big-step normalisation

Girard et al (1989)
Proofs and types

Sieczkowski, Biernacka & Biernacki (2011)
Automating derivations of abstract machines from reduction semantics

Tait (1967)
Intensional interpretations of functionals of finite type i.

Questions?