From Mathematics to Abstract Machine
A formal derivation of an executable Krivine machine

Wouter Swierstra

Radboud University Nijmegen

`w.swierstra@cs.ru.nl`

This paper presents the derivation of an executable Krivine abstract machine from a small step interpreter for the simply typed lambda calculus in the dependently typed programming language Agda.

1 Introduction

There is a close relationship between lambda calculi with explicit substitutions and abstract machines. Biernacka and Danvy [7] have shown how to derive several well-known abstract machines including the Krivine machine [14, 15, 21, 22], the CEK machine [19], and the Zinc machine [23]. Starting with a functional program that evaluates by repeated head reduction, each of these abstract machines may be derived by a series of program transformations. Every transformation is carefully motivated in the accompanying text. This paper aims to nail down the correctness of these derivations further and, in the process, uncover even more structure.

In this paper we show how the derivation presented by Biernacka and Danvy can be formalized in the dependently typed programming language Agda [26]. What do we hope to gain by doing so? In their study relating evaluators and abstract machines, Ager et al. [1] state in the introduction:

> 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.

This paper demonstrates that these non-exhaustive matches are *not* inherent to a dependently typed programming language such as Agda. All the functions we present here are structurally recursive and provide alternatives for every case branch. This shift to a dependently typed language gives us many properties of evaluation 'for free.' For example, from the types alone we learn that evaluation is type preserving and that every term can be decomposed uniquely into a redex and evaluation context. Finally, using Agda enables us to provide a *machine-checked proof* of the correctness of every transformation. More specifically, this paper makes the following concrete contributions:

- We describe the implementation of a small step evaluator in Agda that normalizes by repeated head reduction (Section 3). To convince Agda's termination checker that our definition is sound, we provide a normalization proof in the style of Tait [31], originally sketched by Coquand [13] (Section 4).

- Applying the *refocusing* transformation [18], yields a small-step abstract machine that is not yet tail-recursive (Section 5). We prove that this transformation preserves the semantics and termination properties of the small-step evaluator from Section 4.

- This small-step abstract machine can be transformed further to derive the Krivine machine (Section 6). Once again, we show that the transformation preserves the semantics and termination properties of the small-step abstract machine from Section 5.

This paper is a literate Agda program. Rather than spelling out the details of every proof, we will only sketch the necessary lemmas and definitions. The complete source code, including proofs, is available online.[1] Every section in this paper defines a separate module, allowing us to reuse the same names for the functions and data types presented in individual sections. Finally, the code in this paper uses a short Agda Prelude that is included in an appendix. Readers unfamiliar with Agda may want to consult one of the many tutorials and introductions that are available [10, 27, 28].

2   Types and terms

Before we can develop the series of evaluators, we need to define the terms and types of the simply typed lambda calculus.

```
data Ty : Set where
  O : Ty
  _⇒_ : Ty → Ty → Ty
Context : Set
Context = List Ty
```

The data type Ty represents the types of the simply typed lambda calculus with one base type O. A context is defined to be a list of types. Typically the variables σ and τ range over types; the variables Γ and Δ range over contexts.

Next we define the data types of well-typed, well-scoped variables and lambda terms:

```
data Ref : Context → Ty → Set where
  Top : Ref (Cons σ Γ) σ
  Pop : Ref Γ σ → Ref (Cons τ Γ) σ
data Term : Context → Ty → Set where
  Lam : Term (Cons σ Γ) τ → Term Γ (σ ⇒ τ)
  App : Term Γ (σ ⇒ τ) → Term Γ σ → Term Γ τ
  Var : Ref Γ σ → Term Γ σ
```

These definitions are entirely standard. There are three constructors for the simply typed lambda calculus: Lam introduces a lambda, extending the context; the App constructor applies a term of type $\sigma \Rightarrow \tau$ to an argument of type σ; the Var constructor references a variable bound in the context.

Note that in the typeset code presented in this paper, any unbound variables in type signatures are implicitly universally quantified, as is the convention in Haskell [24] and Epigram [29]. When we wish to be more explicit about implicit arguments, we will adhere to Agda's notation of enclosing such arguments in curly braces.

Next, we can define the data types representing *closed* terms. A *closure* is a term t paired with an environment containing closed terms for all the free variables in t. Furthermore, closed terms are closed under application. This yields the two mutually recursive data types defined below.

---

[1]The source code, compatible with Agda version 2.2.6, is available from `http://www.cs.ru.nl/~wouters`.

```
data Closed : Ty → Set where
   Closure : Term Γ σ → Env Γ → Closed σ
   Clapp : Closed (σ ⇒ τ) → Closed σ → Closed τ
data Env : Context → Set where
   Nil : Env Nil
   _·_ : Closed σ → Env Γ → Env (Cons σ Γ)
```

This is a variation of Curien's $\lambda\rho$-calculus, proposed by Biernacka and Danvy [7]. A similar choice of closed terms was independently proposed by Coquand [13].

The aim of evaluation is to compute a *value* for every closed term. Closed lambda expressions are the only values in our language. The final definitions in this section capture this:

```
isVal : Closed σ → Set
isVal (Closure (Lam body) env) = Unit
isVal _ = Empty
data Value (σ : Ty) : Set where
   Val : (c : Closed σ) → isVal c → Value σ
```

With these types in place, we can specify the type of the evaluation function we will define in the coming sections:

```
evaluate : Closed σ → Value σ
```

3   Reduction

Writing $t\,[env]$ to denote the closure consisting of a term t and an environment *env*, the four rules in below specify a normal-order small step reduction relation for the closed terms. In this section, we will start to implement these rules in Agda.

$$\text{LOOKUP} \quad i\,[c_1, c_2, \ldots c_n] \rightarrow c_i$$

$$\text{APP} \quad (t_0\,t_1)\,[env] \rightarrow (t_0\,[env])\,(t_1\,[env])$$

$$\text{BETA} \quad ((\lambda t)\,[env])\,x \rightarrow t\,[x \cdot env]$$

$$\text{LEFT} \quad \text{if } c_0 \rightarrow c_0' \text{ then } c_0\,c_1 \rightarrow c_0'\,c_1$$

In the style of Danvy and Nielsen [18], we define a single reduction step in three parts. First, we decompose a closed term into a redex and an evaluation context. Second, we contract the redex to form a new closed term. Finally, we plug the resulting closed term back into the evaluation context.

To define such a three-step reduction step, we start by defining the Redex type, corresponding to the left-hand sides of the first three rules above.

```
data Redex : Ty → Set where
   Lookup : Ref Γ σ → Env Γ → Redex σ
   App : Term Γ (σ ⇒ τ) → Term Γ σ → Env Γ → Redex τ
   Beta : Term (Cons σ Γ) τ → Env Γ → Closed σ → Redex τ
```

Of course, every redex can be mapped back to the closed term that it represents.

```
fromRedex : Redex σ  →  Closed σ
fromRedex (Lookup i env)  =  Closure (Var i) env
fromRedex (App f x env)  =  Closure (App f x) env
fromRedex (Beta body env arg)  =  Clapp (Closure (Lam body) env) arg
```

Next, we define the contract function that computes the result of contracting a single redex:

```
_!_  : Env Γ  →  Ref Γ σ  →  Closed σ
Nil ! ()
(x · _) ! Top  =  x
(x · xs) ! Pop r  =  xs ! r

contract : Redex σ  →  Closed σ
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)
```

In the Lookup case, we look up the variable from the environment using the _!_ operator. The App case distributes the environment over the two terms. Finally, Beta reduction extends the environment with the argument arg, and uses the extended environment to create a new closure from the body of a lambda. Once again, the definition of the contract function closely follows the first three reduction rules that we formulated above.

While this describes how to contract a single redex, we still need to define the *decomposition* of a term into a redex and a reduction context. We begin by defining an evaluation context as the list of arguments encountered along the spine of a term:

```
data EvalContext : Ty  →  Ty  →  Set where
   MT : EvalContext σ σ
   ARG : Closed σ  →  EvalContext τ ρ  →  EvalContext (σ ⇒ τ) ρ
```

Ignoring the Ty indices for the moment, an evaluation context is simply a list of closed terms. Given any evaluation context ctx and term t, we would like to plug t in the context by iteratively applying t to all the arguments in ctx. For this to type check, the term t should abstract over all the variables in the evaluation context. We enforce this by indexing the EvalContext type by the 'source' and 'destination' types in the style of Atkey [3]. The plug operation itself then applies any arguments from the evaluation context to its argument term:

```
plug : EvalContext σ τ  →  Closed σ  →  Closed τ
plug MT f = f
plug (ARG x ctx) f = plug ctx (Clapp f x)
```

Finally, we define the decomposition of a closed term into a redex and evaluation context as a *view* [25, 32] on closed terms. Defining such a view consists of two parts: a data type Decomposition indexed by a closed term, and a function decompose that maps every closed term to its Decomposition.

We will start by defining a data type Decomposition. There are two constructors, corresponding to the two possible outcomes of decomposing a closed term c: either c is a value, in which case we have

the closure of a Lam-term and an environment; alternatively, c can be decomposed into a redex r and an evaluation context ctx, such that plugging the term corresponding to r in the evaluation context ctx is equal to the original term c:

```
data Decomposition : Closed σ → Set where
   Val : (body : Term (Cons σ Γ) τ) → (env : Env Γ) →
      Decomposition (Closure (Lam body) env)
   Decompose : (r : Redex σ) → (ctx : EvalContext σ τ) →
      Decomposition (plug ctx (fromRedex r))
```

Next we show how every closed term c can be decomposed into a Decomposition c. We do so by defining a pair of functions, load and unload. The load function traverses the spine of c, accumulating any arguments we encounter in an evaluation context until we find a redex or a closure containing a Lam. The unload function inspects the evaluation context that load has accumulated in order to decide if a lambda is indeed a value, or whether it still has further arguments, and hence corresponds to a Beta redex:

```
load : (ctx : EvalContext σ τ) (c : Closed σ) → 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 (σ ⇒ τ) ρ) (body : Term (Cons σ Γ) τ) (env : Env Γ)
      → 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
```

The decompose function itself simply kicks off load with an initially empty evaluation context.

```
decompose : (c : Closed σ) → Decomposition c
decompose c = load MT c
```

To perform a single reduction step, we decompose a closed term. If this yields a value, there is no further reduction to be done. If decomposition yields a redex and evaluation context, we contract the redex and plug the result back into the evaluation context:

```
headReduce : Closed σ → Closed σ
headReduce c with decompose c
headReduce ⌊Closure (Lam body) env⌋ | Val body env = Closure (Lam body) env
headReduce ⌊plug ctx (fromRedex redex)⌋ | Decompose redex ctx = plug ctx (contract redex)
```

Note that pattern matching on the Decomposition produces more information about the term that has been decomposed. This is apparent in the *forced patterns* [26], ⌊Closure (Lam body) env⌋ in the Val branch and ⌊plug ctx (fromRedex redex)⌋ in the Decompose branch, that appear on the left-hand side of the function definition.

This completes our definition of a single head reduction step.

4   Iterated head reduction

In the previous section we established how to perform a single reduction step. Now it should be straight-forward to define an evaluation function by iteratively reducing by a single step until we reach a value:

```
evaluate : Closed σ → Value σ
evaluate c = iterate (decompose c)
  where
  iterate : Decomposition c → Value σ
  iterate (Val val p) = Val val p
  iterate (Decompose r ctx) = iterate (decompose (plug ctx (contract r)))
```

There is one problem with this definition: it is not structurally recursive. It is rejected by Agda. Yet we know that the simply typed lambda calculus is strongly normalizing—so iteratively performing a single head reduction will always produce a value eventually. How can we convince Agda of this fact?

The Bove-Capretta method is one technique to transform a definition that is not structurally recursive into an equivalent definition that is structurally recursive over a new argument [9]. Essentially, it does structural recursion over the call graph of a function. In our case, we would like to have an inhabitant of the following data type:

```
data Trace : {c : Closed σ} → Decomposition c → Set where
  Done : (body : Term (Cons σ Γ) τ) → (env : Env Γ) → Trace (Val body env)
  Step :  Trace (decompose (plug ctx (contract r))) → Trace (Decompose r ctx)
```

We could then define the iterate function by structural induction over the trace:

```
iterate : {c : Closed σ} → (d : Decomposition c) → Trace d → Value σ
iterate (Val body env) (Done ⌊body⌋ ⌊env⌋) = Val (Closure (Lam body) env) unit
iterate (Decompose r ctx) (Step step) = iterate (decompose (plug ctx (contract r))) step
```

Although this definition does pass Agda's termination checker, the question remains how to provide the required Trace argument to our iterate function. That is we would like to define a function of type:

```
(t : Closed σ) → Trace t
```

A straightforward attempt to define such a function fails immediately. Instead, we need to define the following *logical relation* that strengthens our induction hypothesis:

```
Reducible : {σ : Ty} → (t : Closed σ) → Set
Reducible {O} t = Trace (decompose t)
Reducible {σ ⇒ τ} t = Pair (Trace (decompose t))
                           ((x : Closed σ) → Reducible x → Reducible (Clapp t x))
ReducibleEnv : Env Γ → Set
ReducibleEnv Nil = Unit
ReducibleEnv (x · env) = Pair (Reducible x) (ReducibleEnv env)
```

To prove that all closed terms are reducible, we follow the proof sketched by Coquand [13] and prove the following two lemmas.

```
lemma1 : (c : Closed σ) → Reducible (headReduce c) → Reducible c
lemma2 : (t : Term Γ σ) (env : Env Γ) → ReducibleEnv env → Reducible (Closure t env)
```

The proof of lemma2 performs induction on the term t. In each of the branches, we appeal to lemma1 in order to prove that Closure t env is also reducible. The proof of lemma1 is done by induction on σ and c. The only difficult case is that for closed applications, Clapp f x. In that branch, we need to show that Clapp (headReduce (Clapp f x)) y is equal to headReduce (Clapp (Clapp f x) y).

To prove the desired equality we observe that if decomposing Clapp f x yields a redex r and evaluation context ctx, then the decomposition of Clapp (Clapp f x) y must yield the same redex with the evaluation context obtained by adding y to the end of ctx. To complete the proof we define an auxiliary 'backwards view' on evaluation contexts that states that every evaluation context is either empty or arises by adding a closed term to the end of an evaluation context. Using this view, the required equality is easy to prove.

Using lemma1 and lemma2, we can prove our main theorem: every closed term is reducible. To do so, we define the following two mutually recursive theorems:

```
mutual
    theorem : (c : Closed σ) → Reducible c
    theorem (Closure t env) = lemma2 t env (envTheorem env)
    theorem (Clapp f x) = snd (theorem f) x (theorem x)

    envTheorem : (env : Env Γ) → ReducibleEnv env
    envTheorem Nil = unit
    envTheorem (t · ts) = (theorem t, envTheorem ts)
```

To prove that every closure is reducible, we appeal to lemma2 and prove that every closed term in the environment is also reducible. The proof that every closed application is reducible recurses over both arguments f and x. The recursive call to f yields a pair of a trace and a function of type:

```
((x : Closed σ) → Reducible x → Reducible (Clapp f x))
```

Applying this function to x and theorem x, yields the desired proof.

One important corollary of our theorem is that for every closed term c, we can compute an evaluation trace of c:

```
termination : { σ : Ty } → (c : Closed σ) → Trace (decompose c)
termination { O } c = theorem c
termination { σ ⇒ τ } c = fst (theorem c)
```

Now we can finally complete the definition of our small step evaluation function:

```
evaluate : Closed σ → Value σ
evaluate t = iterate (decompose t) (termination t)
```

The evaluate function iteratively performs a single step of head reduction, performing structural induction over the trace that we compute using the reducibility proof sketched above.

5   Refocusing

The small step evaluator presented in the previous section repeatedly decomposes a closed term into an evaluation context and a redex, contracts the redex, and plugs the contractum back into the evaluation

context. Before transforming this evaluator into the Krivine machine, we will show how to apply the re-focusing transformation to produce a *small-step abstract machine* [16]. This small-step abstract machine forms a convenient halfway point between the small step evaluator and the Krivine machine.

The key idea of refocusing is to compose the plugging and decomposition steps into a single refocus operation. Instead of repeatedly plugging and decomposing, the refocus function navigates directly to the next redex, if it exists:

> refocus : (ctx : EvalContext σ τ) (c : Closed σ) $\rightarrow$ 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

We can formalize this intuition about the behaviour of refocusing by proving the following lemma:

> refocusCorrect : (ctx : EvalContext σ τ) (c : Closed σ) $\rightarrow$
>   refocus ctx c $\equiv$ decompose (plug ctx c)

The proof by induction on ctx and c relies on an easy lemma:

> decomposePlug : (ctx : EvalContext σ τ) (c : Closed σ) $\rightarrow$
>   decompose (plug ctx c) $\equiv$ load ctx c

The proof of the decomposePlug lemma proceeds by simple induction on the evaluation context.

To rewrite our evaluator to use the refocus operation, we will need to adapt the Trace data type from the previous section. Iterated recursive calls will no longer call decompose and plug, but instead navigate to the next redex using the refocus function. The new Trace data type reflects just that:

> **data** Trace : Decomposition c $\rightarrow$ Set **where**
>   Done : (body : Term (Cons σ Γ) τ) $\rightarrow$ (env : Env Γ) $\rightarrow$ Trace (Val body env)
>   Step : Trace (refocus ctx (contract r)) $\rightarrow$ Trace (Decompose r ctx)

To prove that this new Trace data type is inhabited, we call the termination lemma from the previous section. Using the refocusCorrect lemma, we perform induction on the Trace data type from the previous section to construct a witness of termination. All this is done by the following termination function:

> termination : (c : Closed σ) $\rightarrow$ Trace (refocus MT c)

The definition of our evaluator is now straightforward. The iterate function repeatedly refocuses and contracts until a value has been reached:

> iterate : (d : Decomposition c) $\rightarrow$ Trace d $\rightarrow$ Value σ
> iterate (Val body env) (Done $\lfloor$body$\rfloor$ $\lfloor$env$\rfloor$) $=$ Val (Closure (Lam body) env) unit
> iterate (Decompose r ctx) (Step step) $=$ iterate (refocus ctx (contract r)) step
>
> evaluate : Closed σ $\rightarrow$ Value σ
> evaluate c $=$ iterate (refocus MT c) (termination c)

The evaluate function kicks off the iterate function with an empty evaluation context and a proof of termination.

Finally, we can also show that our new evaluator behaves the same as the evaluation function presented in the previous section. To do so, we prove the following lemma by induction on the decomposition of t:

```
correctness : {t : Closed σ} →
   (trace : Trace (refocus MT t)) → (trace' : Section4.Trace (decompose t)) →
   iterate (refocus MT t) trace ≡ Section4.iterate (decompose t) trace'
```

An important corollary of this correctness property is that our new evaluation function behaves identically to the evaluate function from the previous section:

```
corollary : (t : Closed σ) → evaluate t ≡ Section4.evaluate t
corollary t = correctness (termination t) (Section4.termination t)
```

This completes the definition and verification of the evaluator that arises by applying the refocusing transformation on the small step evaluator from Section 4.

6   The Krivine machine

In this section we will derive the Krivine machine from the evaluation function we saw previously. To complete our derivation, we perform a few further program transformations on the previous evaluation function.

We start by inlining the iterate function, making our refocus function recursive. Furthermore, the evaluate function in the previous section mapped App terms into closed Clapp terms, and subsequently evaluated the first argument of the resulting Clapp constructor, adding the second argument to the evaluation context. In this section, we will combine these two steps into a single transition—a transformation sometimes referred to as *compressing corridor transitions* [17]. As a result, we will no longer add closed applications to the environment or evaluation context. We introduce the following predicates enforcing the absence of Clapp constructors on closed terms, environments, and evaluation contexts respectively:

```
mutual
   isValidClosure : Closed σ → Set
   isValidClosure (Closure t env) = isValidEnv env
   isValidClosure (Clapp f x) = Empty

   isValidEnv : Env Δ → Set
   isValidEnv Nil = Unit
   isValidEnv (c · env) = Pair (isValidClosure c) (isValidEnv env)
isValidContext : EvalContext σ τ → Set
isValidContext MT = Unit
isValidContext (ARG (Closure t env) ctx) = Pair (isValidEnv env) (isValidContext ctx)
isValidContext (ARG (Clapp f x) env) = Empty
```

Given that the only valid closed terms are closures, we can define functions that project the underlying environment and term from any valid closed term:

```
getContext : Exists (Closed σ) isValidClosure → Context
getContext (Witness (Closure {Γ} t env) _) = Γ
getContext (Witness (Clapp f x) ())

getEnv : (c : Exists (Closed σ) isValidClosure) → Env (getContext c)
getEnv (Witness (Closure t env) p) = env
getEnv (Witness (Clapp f x) ())

getTerm : (c : Exists (Closed σ) isValidClosure) → Term (getContext c) σ
getTerm (Witness (Closure t env) p) = t
getTerm (Witness (Clapp f x) ())
```

Finally, we can define a new lookup operation that guarantees that looking up a variable in a valid environment will always return a closure:

```
lookup : Ref Γ σ → (env : Env Γ) → isValidEnv env →
   Exists (Closed σ) isValidClosure
lookup Top (Closure t env · _) (p1, p2) = Witness (Closure t env) p1
lookup Top (Clapp _ _ · _) ((), _)
lookup (Pop i) (_ · env) (_, p) = lookup i env p
```

If the argument reference is Top, we pattern match on the environment, which must contain a closure. We use the proof that the environment contains exclusively closures to discharge the Clapp branch. If the argument reference is Pop i, we recurse over i and the tail of the environment.

Once again, we define a Trace data type, describing the call-graph of the Krivine machine. The Trace data type is indexed by the three arguments to the Krivine machine: a term, an environment, and an evaluation context. The data type has a constructor for every transition; recursive calls to the abstract machine correspond to recursive arguments to a constructor:

```
data Trace : Term Γ σ → Env Γ → EvalContext σ τ → Set where
   Lookup : (i : Ref Γ σ) (p : isValidEnv env) →
     let c = lookup i env p in
     Trace (getTerm c) (getEnv c) ctx → Trace (Var i) env ctx
   App : (f : Term Γ (σ ⇒ τ)) (x : Term Γ σ) →
     Trace f env (ARG (Closure x env) ctx) →
     Trace (App f x) env ctx
   Beta : (ctx : EvalContext σ ρ) →
     (arg : Term H τ) → (argEnv : Env H) →
     (body : Term (Cons τ Γ) σ) →
     Trace body (Closure arg argEnv · env) ctx →
     Trace (Lam body) env (ARG (Closure arg argEnv) ctx)
   Done : (body : Term (Cons τ Γ) σ) → Trace (Lam body) env MT
```

Using this Trace, we can now define the final version of the refocus function, corresponding to the Krivine abstract machine, by structural recursion on this Trace.

```
refocus : (ctx : EvalContext σ τ) (t : Term Γ σ) (env : Env Γ) →
    Trace t env ctx → Value τ
refocus ctx ⌊Var i⌋ env (Lookup i q step) =
    let c = lookup i env q in
    refocus ctx (getTerm c) (getEnv c) step
refocus ctx ⌊App f x⌋ env (App f x step)
    = refocus (ARG (Closure x env) ctx) f env step
refocus ⌊ARG (Closure arg env') ctx⌋ ⌊Lam body⌋ env (Beta ctx arg env' body step)
    = refocus ctx body ((Closure arg env') · env) step
refocus ⌊MT⌋ ⌊Lam body⌋ env (Done body) = Val (Closure (Lam body) env) unit
```

In the case for variables, we look up the closure that the variable refers to in the environment, and continue evaluation with that closure's term and environment. In the case for App f x, we add the argument and current environment to the application context, and continue evaluating the term f. We distinguish two further cases for lambda terms: if the evaluation context is not empty, we can perform a beta reduction step; otherwise evaluation is finished.

We still need to prove that the Trace data type is inhabited. During execution, the Krivine machine only adds closures to the environment and evaluation context. During the termination proof, we will need to keep track of the following invariant on evaluation contexts and environments:

```
invariant : EvalContext σ τ → Env Γ → Set
invariant ctx env = Pair (isValidEnv env) (isValidContext ctx)
```

The proof of termination once again calls the termination proof from the previous section. An auxiliary lemma shows that any witness of termination for the small-step abstract machine in Section 5 will also suffice as a proof of termination of the Krivine machine.

```
termination : (t : Term Nil σ) → Trace t Nil MT
termination t = lemma MT t Nil (unit, unit) (Section5.termination (Closure t Nil))
    where
    lemma : (ctx : EvalContext σ τ) (t : Term Γ σ) (env : Env Γ) →
        invariant ctx env → Section5.Trace (Section5.refocus ctx (Closure t env)) →
        Trace t env ctx
```

The lemma is proven by straightforward induction on the evaluation context, the term, and the Trace data type from the previous section. Once we pattern match on the term and the evaluation context, we know which transition we wish to make, and hence which constructor of the Trace data type is required. Any recursive occurrences of the Trace data type can be produced by recursive calls to the lemma. The only other result necessary states that the lookup function and the _!_ operation we saw previously return the same closed term from an environment.

Finally, we can define the evaluation function that calls refocus with a suitable choice for its initial arguments:

```
evaluate : Term Nil σ → Value σ
evaluate t = refocus MT t Nil (termination t)
```

To conclude, we show that this final version of the refocus function behaves equivalently to the refocus function from the previous section. To prove this, we formulate the correctness property below.

```
correctness : (ctx : EvalContext σ τ) (t : Term Γ σ) (env : Env Γ) →
    (t₁ : Trace t env ctx) →
    (t₂ : Section5.Trace (Section5.refocus ctx (Closure t env))) →
    refocus ctx t env t₁ ≡ Section5.iterate (Section5.refocus ctx (Closure t env)) t₂
```

Once again, the proof proceeds by straightforward induction on the traces.

As a result of this correctness property, we can prove that our evaluation function behaves the same as the function presented in the previous section:

```
corollary : (t : Term Nil σ) → evaluate t ≡ Section5.evaluate (Closure t Nil)
corollary t = let trace  = termination t in
              let trace' = Section5.termination (Closure t Nil) in
              correctness MT t Nil trace trace'
```

By chaining together our correctness results, we can show that our Krivine machine produces the same value as our original evaluator based on repeated head reduction, thereby completing the formal derivation of the Krivine machine from a small step evaluator.

7   Discussion

There has been previous work on formalizing the derivations of abstract machines in Coq [6, 30]. In contrast to the development here, these formalizations are not executable but instead define the reduction behaviour as inductive relations between terms and values. The executability of our abstract machines comes at a price: we need to prove that the evaluators terminate, which requires a clever logical relation. On the other hand, it is easier to reason about executable functions. In type theory, definitional equalities are always trivially true—a fact you can only exploit if your functions compute.

This paper uses the Bove-Capretta method to prove termination of every evaluator. Chapman and Altenkirch use a similar logical relation to produce inhabitants of Bove-Capretta predicates when writing a big-step normalization algorithm [2]. There are, of course, alternative methods to show that a non-structurally recursive function does terminate. For example, it may be interesting to investigate how to adapt the normalization proof to use an order on lambda terms proposed by Gandy [20] to define a suitable accessibility relation.

Finally, you may wonder if the usage of logical relations to prove termination is 'cheating.' After all, the computational content of normalization proofs using logical relations is itself a normalization algorithm [4, 5, 8]—so is our small-step evaluator not just reading off the value from the trace that our proof computes? Not at all! In fact, the behaviour of the iterate function from Section 4 is *independent* of the trace we provide—once the iterate function matches on the argument decomposition, the trace passed as an argument to the iterate function is uniquely determined. The following statement is easy to prove:

```
collapsible : (d : Decomposition c) (t₁ t₂ : Trace d) → t₁ ≡ t₂
```

In other words, the traces themselves carry no computational content. Such *collapsible* data types may be erased by a suitable clever compiler [11, 12].

This paper focuses on the derivation of the Krivine abstract machine. There is no reason to believe that the other derivations of abstract machines [1, 7] may not be formalized in a similar fashion.

Acknowledgements

I would like to thank James McKinna for our entertaining and educational discussions. Małgorzata Biernacka, Pierre-Evariste Dagand, Olivier Danvy, Ilya Sergey, and Thomas van Noort all provided invaluable feedback on a draft version of this paper—for which I am grateful.

References

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy & Jan Midtgaard (2003): *A functional correspondence between evaluators and abstract machines*. In: *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming*, ACM, pp. 8–19.

[2] Thorsten Altenkirch & James Chapman (2009): *Big-step normalisation*. Journal of Functional Programming 19(3-4), pp. 311–333.

[3] Robert Atkey (2009): *Parameterised notions of computation*. Journal of Functional Programming 19(3 & 4), pp. 335–376.

[4] Ulrich Berger (1993): *Program extraction from normalization proofs*. Typed Lambda Calculi and Applications , pp. 91–106.

[5] Ulrich Berger, Stefan Berghofer, Pierre Letouzey & Helmut Schwichtenberg (2006): *Program extraction from normalization proofs*. Studia Logica 82(1), pp. 25–49.

[6] Małgorzata Biernacka & Dariusz Biernacki (2007): *Formalizing Constructions of Abstract Machines for Functional Languages in Coq*. In: *7th International Workshop on Reduction Strategies in Rewriting and Programming*, pp. 84–99.

[7] Małgorzata Biernacka & Olivier Danvy (2007): *A concrete framework for environment machines*. ACM Transactions on Computational Logic (TOCL) 9(1), pp. 6:1–6:30.

[8] Małgorzata Biernacka, Olivier Danvy & Kristian Støvring (2006): *Program extraction from proofs of weak head normalization*. Electronic Notes in Theoretical Computer Science 155, pp. 169–189.

[9] Ana Bove & Venanzio Capretta (2005): *Modelling general recursion in type theory*. Mathematical Structures in Computer Science 15(4), pp. 671–708.

[10] Ana Bove & Peter Dybjer (2009): *Dependent Types at Work*. In Ana Bove, Luís Barbosa, Alberto Pardo & Jorge Pinto, editors: *Language Engineering and Rigorous Software Development, Lecture Notes in Computer Science* 5520, Springer, pp. 57–99.

[11] Edwin Brady (2005): *Practical Implementation of a Dependently Typed Functional Programming Language*. Ph.D. thesis, University of Durham.

[12] Edwin Brady, Conor McBride & James McKinna (2003): *Inductive Families Need Not Store Their Indices*. In: *TYPES*, pp. 115–129.

[13] Thierry Coquand (1999): *Inductive Definitions and Type Theory: an introduction*. In: *Proceedings of the TYPES Summer School*.

[14] Pierre Crégut (1990): *An abstract machine for lambda-terms normalization*. In Mitchell Wand, editor: *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pp. 333–340.

[15] P.L. Curien (1991): *An abstract framework for environment machines*. Theoretical Computer Science 82(2), pp. 389–402.

[16] O. Danvy & K. Millikin (2008): *On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion*. Information Processing Letters 106(3), pp. 100–109.

[17] Olivier Danvy (2009): *From reduction-based to reduction-free normalization*. In Pieter Koopman, Rinus Plasmeijer & Doaitse Swierstra, editors: *Proceedings of the 6th International School on Advanced Functional Programming, LNCS* 5382, Springer-Verlag, pp. 66–164.

[18] Olivier Danvy & Lasse R. Nielsen (2004): *Refocusing in Reduction Semantics*. Technical Report RS-04-26, BRICS.

[19] Matthias Felleisen & Daniel P. Friedman (2005): *Control operators, the SECD-machine and the lambda-calculus*. Formal Description of Programming Concepts III .

[20] R.O. Gandy (1980): *Proofs of Strong Normalization*. In J.P. Seldin & J.R. Hindley, editors: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, pp. 457–490.

[21] Chris Hankin (1994): *Lambda Calculi, a guide for computer scientists*. Graduate Texts in Computer Science 1, Oxford University Press.

[22] Jean-Louis Krivine (2007): *A call-by-name lambda-calculus machine*. Higher Order and Symbolic Computation 20(3), pp. 199–207.

[23] Xavier Leroy (1990): *The ZINC experiment: an economical implementation of the ML language*. Technical Report, INRIA Rocquencourt.

[24] Conor McBride & James McKinna (2004): *The view from the left*. Journal of Functional Programming 14(1).

[25] Conor McBride & James McKinna (2004): *The view from the left*. Journal of Functional Programming 14(1), pp. 69–111.

[26] Ulf Norell (2007): *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Chalmers University of Technology.

[27] Ulf Norell (2008): *Dependently Typed Programming in Agda*. In Pieter Koopman, Rinus Plasmeijer & Doaitse Swierstra, editors: *Advanced Functional Programming, LNCS-Tutorial* 5832, Springer-Verlag, pp. 230–266.

[28] Nicolas Oury & Wouter Swierstra (2008): *The Power of Pi*. In: *ICFP '08: Proceedings of the Thirteenth ACM SIGPLAN International Conference on Functional Programming*, pp. 39–50.

[29] Simon Peyton Jones, editor (2003): *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press.

[30] Filip Sieczkowski, Malgorzata Biernacka & Dariusz Biernacki (2010): *Automating Derivations of Abstract Machines from Reduction Semantics: A Generic Formalization of Refocusing in Coq*. In: *22nd Symposium on Implementation and Application of Functional Languages*, pp. 72–88.

[31] W.W. Tait (1967): *Intensional interpretations of functionals of finite type I*. Journal of Symbolic Logic , pp. 198–212.

[32] Philip Wadler (1987): *Views: A way for pattern matching to cohabit with data abstraction*. In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 307–313.

A An Agda Prelude

```
module Prelude where
id : forall {a : Set} → a → a
id x = x
data Empty : Set where
magic : forall {a : Set} → Empty → a
magic ()
record Unit : Set where

unit : Unit
unit = record { }
data Pair (a b : Set) : Set where
    _,_ : a → b → Pair a b
fst : forall {a b} → Pair a b → a
fst (x, _) = x
snd : forall {a b} → Pair a b → b
snd (_, y) = y
data List (a : Set) : Set where
    Nil : List a
    Cons : a → List a → List a
data _≡_ {A : Set} (x : A) : A → Set where
    Refl : x ≡ x
infix 6 _≡_
sym : {a : Set} {x y : a} → x ≡ y → y ≡ x
sym Refl = Refl
cong : {a b : Set} {x y : a} → (f : a → b) → x ≡ y → f x ≡ f y
cong f Refl = Refl
data Exists (a : Set) (b : a → Set) : Set where
    Witness : (x : a) → b x → Exists a b
fsts : forall {a b} → Exists a b → a
fsts (Witness x _) = x
snds : forall {a b} → (x : Exists a b) → (b (fsts x))
snds (Witness _ y) = y
```