Automating the Functional Correspondence between Higher-Order Evaluators and Abstract Machines

Maciej Buszka and Dariusz Biernacki

Institute of Computer Science, University of Wroclaw, Poland

Abstract. The functional correspondence is a manual derivation technique transforming higher-order evaluators into the semantically equivalent abstract machines. The transformation consists of two well-known program transformations: translation to continuation-passing style that uncovers the control flow of the evaluator and Reynolds’s defunctionalization that generates a first-order transition function. Ever since the transformation was first described by Danvy et al. it has found numerous applications in connecting known evaluators and abstract machines, but also in discovering new abstract machines for a variety of λ-calculi as well as for logic-programming, imperative and object-oriented languages. We present an algorithm that automates the functional correspondence. The algorithm accepts an evaluator written in a dedicated minimal functional meta-language and it first transforms it to administrative normal form, which facilitates program analysis, before performing selective translation to continuation-passing style, and selective defunctionalization. The two selective transformations are driven by a control-flow analysis that is computed by an abstract interpreter obtained using the abstracting abstract machines methodology, which makes it possible to transform only the desired parts of the evaluator. The article is accompanied by an implementation of the algorithm in the form of a command-line tool that allows for automatic transformation of an evaluator embedded in a Racket source file and gives fine-grained control over the resulting machine.

Keywords: Evaluator · Abstract machine · Continuation-passing style · Defunctionalization.

1 Introduction

When it comes to defining or prototyping a programming language one traditionally provides an interpreter for the language in question (the object-language) written in another language (the meta-language). These definitional interpreters can be placed on a spectrum from most abstract to most explicit. At the abstract end lie the concise meta-circular interpreters which use meta-language constructs to interpret the same constructs in the object-language (e.g., using anonymous functions to model functional values, using conditionals for if
expressions, etc.). In the middle one might place various evaluators with some constructs interpreted by simpler language features (e.g., with environments represented as lists or dictionaries instead of functions), but still relying on the evaluation order of the meta-language. The explicit end is occupied by first-order machine-like interpreters which use an encoding of a stack for handling control-flow of the object-language.

When it comes to modelling an implementation of a programming language, and a functional one in particular, one traditionally constructs an abstract machine, i.e., a first-order tail-recursive transition system for program execution. Starting with Landin’s SECD machine [25] for $\lambda$-calculus, many abstract machines have been proposed for various evaluation strategies and with differing assumptions on capabilities of the runtime (e.g., substitution vs environments). Notable work includes: Krivine’s machine [24] for call-by-name reduction, Felleisen and Friedman’s CEK machine [17] and Crégu’s machine [13] for normalization of $\lambda$-terms in normal order. Manual construction of an abstract machine for a given evaluation discipline can be challenging and it requires a proof of equivalence w.r.t. the higher-level semantics, therefore methods for deriving the machines from natural or reduction semantics have been developed [20,15,32,10]. However, one of the most fruitful and accessible abstract machine derivation methods was developed in the realm of interpreters and program transformations by Danvy et al. who introduced a functional correspondence between higher-order evaluators and abstract machines [4] – the topic of the present work.

The functional correspondence is a realization that Reynolds’s transformation to continuation-passing style [31] and defunctionalization, which allow one to transform higher-order, meta-circular, compositional definitional interpreters into first-order, tail-recursive ones, can be seen as a general method of actually transforming an encoding of a denotational or natural semantics into an encoding of an equivalent abstract machine. The technique has proven to be indispensable for deriving a correct-by-construction abstract machine given an evaluator in a diverse set of languages and calculi including normal and applicative order $\lambda$-calculus evaluation [4] and normalization [3], call-by-need strategy [5] and Haskell’s STG language [29], logic engine [11], delimited control [9], computational effects [6], object-oriented calculi [14] and Coq’s tactic language [23]. Despite these successes and its mechanical nature, the functional correspondence has not yet been transformed into a working tool which would perform the derivation automatically.

The goal of this work is to give an algorithmic presentation of the functional correspondence that has been implemented by the first author as a semantics transformer. In particular, we describe the steps required to successfully convert the human-aided derivation method into a computer algorithm for transforming evaluators into a representation of an abstract machine. Our approach hinges on control-flow analysis as the basis for both selective continuation-passing style transformation and partial defunctionalization, and, unlike most of the works

---

1 The transformation used by Reynolds was later formalized by Plotkin as call-by-value CPS translation [30].
treating such transformations, we do not rely on a type system. In order to obtain correct, useful and computable analysis we employ the abstracting abstract machines methodology (AAM) which allows for deriving the analysis from an abstract machine for the meta-language. This derivation proved very capable in handling the non-trivial meta-language containing records, anonymous functions and pattern matching. The resulting analysis enables automatic transformation of user specified parts of the interpreter as opposed to whole-program-only transformations. The transformation, therefore, consists of: (1) transformation to administrative normal form (ANF) that facilitates the subsequent steps, (2) control-flow analysis using the AAM technique and selective (based on the analysis) CPS transformation that makes the control flow in the evaluator explicit and independent from the meta-language, (3) control-flow analysis once more and selective (again, based on the analysis) defunctionalization that replaces selected function spaces with their first-order representations (e.g., closures and stacks), and (4) let inlining that cleans up after the transformation.

The algorithm has been implemented in the Haskell programming language giving raise to a tool — semt — performing the transformation. The tool accepts evaluators embedded in Racket source files. Full Racket language is available for testing the evaluators. We tested the tool on multiple interpreters for a diverse set of programming language calculi. It is available at:

[https://github.com/mbuszka/semantic-transformer](https://github.com/mbuszka/semantic-transformer)

The rest of this article is structured as follows: In Section 2 we introduce the Interpreter Definition Language or IDL which is the meta-language accepted by the transformer and will be used in example evaluators throughout the paper. In Section 3 we present the algorithmic characterization of the functional correspondence. In Section 4 we briefly discuss the performance of the tool on a selection of case studies. In Section 5 we point at future avenues for improvement and conclude. In Appendix A we illustrate the functional correspondence with a minimal example, for the readers unfamiliar with the CPS transformation and/or defunctionalization. Appendix B contains an extended example—a transformation of a normalization-by-evaluation function for λ-calculus into the corresponding abstract machine.

2 Interpreters and the meta-language

The Interpreter Definition Language or IDL is the meta-language used by semt—a semantic transformer that we have developed. It is a purely functional, higher-order, dynamically typed language with strict evaluation order. It features named records and pattern matching which allow for convenient modelling of abstract syntax of the object-language as well as base types of integers, booleans and strings. The concrete syntax is in fully parenthesized form and the programs can be embedded in a Racket source file using a provided library with syntax definitions.
def-data Term
  String
  {Abs String Term}
  {App Term Term})

(def init (x) (error "empty environment"))

(def extend (env y v)
  (fun (x) (if (eq? x y) v (env x))))

(def eval (env term)
  (match term
    ([String x] (env x))
    ({Abs x body} (fun (v) (eval (extend env x v) body)))
    ({App fn arg} ((eval env fn) (eval env arg)))))

(def main ([Term term]) (eval init term))

Fig. 1. A meta-circular interpreter for λ-calculus

\[
x, y, z, f \in \text{Var} \quad r \in \text{StructName} \quad s \in \text{String} \quad b \in \text{Int} \cup \text{Boolean} \cup \text{String}
\]
\[
Tp \ni tp ::= \text{String} \mid \text{Integer} \mid \text{Boolean}
\]
\[
\text{Pattern} \ni p ::= x \mid b \mid _ \mid \{ r \ p \ldots \} \mid (tp x)
\]
\[
\text{Term} \ni t ::= x \mid b \mid (\text{fun} (x\ldots) t) \mid (t \ t\ldots) \mid \{ r \ t\ldots \}
\]
\[
\phantom{t} \mid (\text{let} p t t) \mid (\text{match} t (p t\ldots) \mid (\text{error} s)
\]

Fig. 2. Abstract syntax of the IDL terms

As shown in Figure 1 a typical interpreter definition consists of several top-level function definitions which may be mutually recursive. The def-data form introduces a datatype definition. In our case it defines a type Term for terms of λ-calculus. It is a union of three types: Strings representing variables of λ-calculus; records with label Abs and two fields of types String and Term representing abstractions; and records labeled App which contain two Terms and represent applications. A datatype definition may refer to itself, other previously defined datatypes and records, the base types of String, Integer and Boolean or a placeholder type Any. The main function is treated as an entry point for the evaluator and must have its arguments annotated with their type.

The match expression matches an expression against a list of patterns. Patterns may be variables (which will be bound to the value being matched), wildcards _, base type patterns, e.g., [String x] or record patterns, such as {Abs x body}. The fun form introduces anonymous function, error "..." stops execution and signals the error. Finally, application of a function is written as in Racket, i.e., as a list of expressions (e.g., (eval init term)). The evaluator in Figure 1 takes advantage of the functional representation of environments
(init and extend) and it structurally recursively interprets \(\lambda\)-terms (eval).

The evaluation strategy for the object-language is in this case inherited from the meta-language, and, therefore, call by value (we assumed IDL strict) \[31\].

The abstract syntax of the IDL terms is presented in Figure 2. The meta-variables \(x, y, z\) denote variables; \(r\) denotes structure (aka record) names; \(s\) is used to denote string literals and \(b\) is used for all literal values – strings, integers and booleans. The meta-variable \(tp\) is used in pattern matches which check whether a value is one of the primitive types. The patterns are referred to with variable \(p\) and may be a variable, a literal value, a wildcard, a record pattern or a type test. Terms are denoted with variable \(t\) and are either a variable, a literal value, an anonymous function, an application, a record constructor, a let binding (which may destruct bind term with a pattern), a pattern match or an error expression.

3 Transformation

The transformation described in this section consists of three main stages: translation to administrative normal form, selective translation to continuation-passing style, and selective defunctionalization. After defunctionalization the program is in the desired form of an abstract machine. The last step taken by the transformer is inlining of administrative let-bindings introduced by previous steps in order to obtain more readable results. In the remainder of this section we will describe the three main stages of the transformation and the algorithm used to compute the control-flow analysis.

3.1 Administrative Normal Form

The administrative normal form (ANF) \[13\] is an intermediate representation for functional languages in which all intermediate results are let-bound to names. This shape greatly simplifies later transformations as programs do not have complicated sub-expressions. From the operational point of view, the only place where a continuation is grown when evaluating program in ANF is a let-binding. This property ensures that a program in ANF is also much easier to evaluate using an abstract machine which will be taken advantage of in Section 3.2.

The abstract syntax of terms in ANF and an algorithm for transforming IDL programs into such form is presented in Figure 3. The terms are partitioned into three levels: variables, commands and expressions. Commands \(c\) extend variables with values – base literals, record constructors (with variables as sub-terms) and abstractions (whose bodies are in ANF); and with redexes like applications of variables and match expressions (which match on a variable and have branches in ANF). Expressions \(e\) in ANF have the shape of a possibly empty sequence of let-bindings ending with either an error term or a command.

The \(\llbracket\cdot\rrbracket\) function, written in CPS\[4\], is the main transformation function. Its arguments are a term to be transformed and a meta-language continuation.
\[\begin{align*}
Com \ni c &::= x \mid b \mid \text{(fun} (x\ldots) e) \mid (x \ x\ldots) \\
&\quad \mid (r \ x\ldots) \mid \text{(match} x (p \ e)\ldots)
\end{align*}\]

\[\begin{align*}
Anf \ni e &::= c \mid \text{(let} p c e) \mid \text{(error} s)
\end{align*}\]

\[
\begin{align*}
\llbracket \cdot \rrbracket_{a} &::= \text{Term} \times (\text{Com} \rightarrow \text{Anf}) \rightarrow \text{Anf} \\
\llbracket x \rrbracket_{a} &= k \ x \\
\llbracket b \rrbracket_{a} &= k \ b \\
\llbracket \text{(fun} (x\ldots) e) \rrbracket_{a} &= k \ \llbracket \text{fun} (x\ldots) [e] id \rrbracket \\
\llbracket (ef e_{arg\ldots}) \rrbracket_{a} &= [ef] \ \llbracket \lambda x_{f}. \ [e_{arg\ldots}]_{a} \lambda (x_{arg\ldots}) \cdot k (x_{f} \ x_{arg\ldots}) \rrbracket_{a} \\
\llbracket (\text{let} \ p \ c \ e_{2}) \rrbracket_{a} &= [c_{1}] \ \llbracket \lambda c_{1}. (\text{let} \ p \ c_{1} [e_{2}] \rrbracket \\
\llbracket (r \ e\ldots) \rrbracket_{a} &= [e_{\ldots}]_{a} \ \llbracket \lambda (x\ldots) \cdot k (r \ x\ldots) \rrbracket_{a} \\
\llbracket \text{(match} e (p \ e_{0}) \rrbracket_{a} &= [e] \ [\lambda x. k \ (\text{match} x (p \ [e_{0}] id)]_{a} \\
\llbracket (\text{error} s) \rrbracket_{a} &= (\text{error} s)
\end{align*}\]

\[
\begin{align*}
\llbracket [\cdot]_{a} &::= (\text{Var} \rightarrow \text{Anf}) \rightarrow \text{Com} \rightarrow \text{Anf} \\
\llbracket [\cdot]_{a} x &= k \ x \\
\llbracket [\cdot]_{a} c &= \text{(let} x c (k \ x) \\
\llbracket [\cdot]_{a}^{*} &::= \text{Term}^{*} \times (\text{Var}^{*} \rightarrow \text{Anf}) \rightarrow \text{Anf} \\
\llbracket [\cdot]_{a}^{*} (e \ldots) k &= \text{go}(e \ldots, e, c, k) \\
\text{go}(e, x, \ldots, k) &= k (x\ldots) \\
\text{go}(e, e_{r}, x_{acc}, \ldots, k) &= [e] \ [\lambda x. \text{go}(e_{r}, \ldots, x_{acc}, \ldots, x, k)]_{a}
\end{align*}\]

Fig. 3. ANF transformation for IDL

which will be called to obtain the term for the rest of the transformed input. This function decomposes the term according to the (informal) evaluation rules and uses two helper functions. Function \[\llbracket [\cdot]_{a} \rrbracket_{a}\] transforms a continuation expecting a variable (which are created when transforming commands) into one accepting any command by let-binding the passed argument \(c\) when necessary. Function \[\llbracket [\cdot]_{a}^{*} \rrbracket_{a}\] sequences computation of multiple expressions by creating a chain of let-bindings (using \[\llbracket [\cdot]_{a} \rrbracket_{a}\]) and then calling the continuation with created variables.

### 3.2 Control-Flow Analysis

The analysis most relevant to the task of deriving abstract machines from interpreters is the control-flow analysis. Its objective is to find for each expression in a program an over-approximation of a set of functions it may evaluate to \[28\]. This information can be used in two places: when determining whether a function and applications should be CPS transformed and for checking which functions an expression in operator position may evaluate to. There are a couple of different approaches to performing this analysis available in the literature: abstract interpretation \[28\], (annotated) type systems \[28\] and abstract abstract machines \[22\]. We chose to employ the last approach as it allows for derivation of the control-flow analysis from an abstract machine for IDL. The derivation technique guarantees correctness of the resulting interpreter and hence provides
\( \nu \in \text{VAddr} \ \kappa \in \text{KAddr} \ \ l \in \text{Label} \ \sigma \in \text{Store} \)
\( \delta \in \text{PrimOp} \subseteq \text{Val}^* \rightarrow \text{Val} \)
\( \rho \in \text{Env} = \text{Var} \rightarrow \text{VAddr} \)
\( \text{Val} \ni v := b \mid \delta \mid \{r \nu \ldots\} \mid \langle \text{def} \ x \ (x \ldots) \ e \rangle \)
\( \text{Cont} \ni k := \langle \rho, p, e, \kappa \rangle \mid \langle \rangle \)
\( \text{PartialConf} \ni \gamma := \langle \rho, e, \kappa \rangle_e \mid \langle \nu, \kappa \rangle_e \)
\( \text{Conf} \ni \zeta := \langle \sigma, \gamma \rangle \)

\[
\begin{align*}
\langle \sigma, \langle \rho, x, \kappa \rangle_e \rangle & \Rightarrow \langle \text{copy}_x(\rho(x), l, \sigma), \langle \rho(x), \kappa \rangle_e \rangle \\
\langle \sigma, \langle \rho, b', \kappa \rangle_e \rangle & \Rightarrow \langle \sigma', \langle \nu, \kappa \rangle_e \rangle \\
& \quad \text{where } \langle \sigma', \nu \rangle = \text{alloc}_v(b, l, \sigma) \\
\langle \sigma, \langle \rho, \{r \ x \ldots\}^l, \kappa \rangle_e \rangle & \Rightarrow \langle \sigma', \langle \nu, \kappa \rangle_e \rangle \\
& \quad \text{where } \langle \sigma', \nu \rangle = \text{alloc}_v(\{r \rho(x)\ldots\}, l, \sigma) \\
\langle \sigma, \langle \rho, \langle \text{fun} \ (x\ldots) e \rangle^l, \kappa \rangle_e \rangle & \Rightarrow \langle \sigma', \langle \nu, \kappa \rangle_e \rangle \\
& \quad \text{where } \langle \sigma', \nu \rangle = \text{alloc}_v(\langle \rho, x\ldots, e \rangle, l, \sigma) \\
\langle \sigma, \langle \rho, \langle \text{let} \ p \ c \ e \rangle^l, \kappa \rangle_e \rangle & \Rightarrow \langle \sigma', \langle \rho, c, \kappa' \rangle_e \rangle \\
& \quad \text{where } \langle \sigma', \kappa' \rangle = \text{alloc}_v(\langle \rho, p, e, \kappa \rangle, l, \sigma) \\
\langle \sigma, \langle \rho, \langle x \ y \ldots\rangle, \kappa \rangle_e \rangle & \Rightarrow \text{apply}(\sigma, \rho(x), \rho(y)\ldots, l) \\
\langle \sigma, \langle \rho, \langle \text{match} \ x \langle p \ e \ldots\rangle, \kappa \rangle_e \rangle & \Rightarrow \text{match}(\sigma, \rho, \rho(x), \rho(e)\ldots) \\
\langle \sigma, \langle \nu, \kappa \rangle_e \rangle & \Rightarrow \text{match}(\sigma, \rho, \nu, \kappa, \langle \rho, e \rangle) \\
& \quad \text{where } \langle \rho, p, e, \kappa' \rangle = \text{deref}_k(\sigma, \kappa) \\
\end{align*}
\]

\[
\begin{align*}
\text{apply}(\sigma, \nu, \nu', \ldots, \kappa, l) = & \left\{ \begin{array}{ll}
\langle \sigma, \langle \rho(x \mapsto \nu')\ldots, e, \kappa \rangle_e \rangle & \text{when } \text{deref}_e(\sigma, \nu) = \langle \rho, x\ldots, e \rangle \\
\langle \sigma, \langle \rho_0(\langle x \mapsto \nu' \rangle\ldots, e, \kappa) \rangle_e \rangle & \text{when } \text{deref}_e(\sigma, \nu) = \langle \text{def} \ y \ (x \ldots) \ e \rangle \\
\langle \sigma', \langle \nu', \kappa \rangle_e \rangle & \text{when } \text{deref}_e(\sigma, \nu) = \delta \\
\langle \sigma', \langle \nu'' \rangle_e \rangle & \text{when } \text{deref}_e(\sigma, \nu) = \delta \\
\text{and } \langle \sigma', \nu'' \rangle & = \text{alloc}_v(\delta(\sigma(\nu'\ldots), l, \sigma) \\
\end{array} \right.
\end{align*}
\]

\[
\text{match}(\sigma, \rho, \nu, \kappa, \langle p, e \rangle\ldots) = \langle \sigma, \langle \rho', e', \kappa \rangle_e \rangle \text{ where } \rho' \text{ is the environment}
\]

for the first matching branch with body \( e' \).

---

Fig. 4. A template abstract machine for IDL terms in ANF

We high confidence in the actual implementation of the machine. We next present the template for acquiring both concrete and abstract versions of the abstract machine for IDL. The former machine defines the semantics of IDL; the latter computes the CFA.

**A Machine Template** We will begin with a template of a machine for IDL terms in A-normal form, presented in Figure 4. It is a CEK-style machine modified to explicitly allocate memory for values and continuations in an abstract store. The template is parameterized by: implementation of the store \( \sigma \) along with five operations: \text{alloc}_e, \text{alloc}_k, \text{deref}_e, \text{deref}_k \) and \text{copy}_v; interpretation of primitive operations \( \delta \) and implementation of \text{match} function which interprets pattern matching. The store maps value addresses \( \nu \) to values \( v \) and continuation addresses \( \kappa \) to continuations \( k \). The environment maps program variables
to value locations. The values on which the machine operates are the following: base values $b$, primitive operations $\delta$, records with addresses as fields, closures and top-level functions. Thanks to terms being in A-normal form, there are only two kinds of continuations which form a stack. The stack frames $\langle \rho, p, e, \kappa \rangle$ are introduced by let-bindings. They hold an environment $\rho$, a pattern $p$ to use for destructuring a value, the body $e$ of a let expression and a pointer to the next continuation $\kappa$. The bottom of the stack is marked by the empty continuation $\langle \rangle$. We assume that every term has a unique label $l$ which will be used in the abstract version of the machine to implement store addresses.

The machine configurations are pairs of a store $\sigma$ and a partial configuration $\gamma$. This split of configuration into two parts will prove beneficial when we instantiate the template to obtain an abstract interpreter. There are two classes of partial configurations. An evaluation configuration contains an environment $\rho$, an expression $e$ and a continuation pointer $\kappa$. A continuation configuration holds an address $\nu$ of a value that has been computed so far and a pointer $\kappa$ to a resumption which should be applied next.

The first case of the transition relation $\Rightarrow$ looks up a pointer for the variable $x$ in the environment $\rho$ and switches to continuation mode. It modifies the store via copy function which ensures that every occurrence of a variable has a corresponding binding in the store. The next three cases deal with values by allocating them in the store and switching to continuation mode. When the machine encounters a let-binding it allocates a continuation for the body $e$ of the expression and proceeds to evaluate the bound command $c$ with the new pointer $\kappa'$. In case of applications and match expressions the resulting configuration is decided using auxiliary functions apply and match, respectively. Finally, in continuation mode, the machine may only transition if the continuation loaded from the address $\kappa$ is a frame. In such a case the machine matches the stored pattern against the value pointed-to by $\nu$. Otherwise $\kappa$ points to a $\langle \rangle$ instead and the machine has reached the final state. The auxiliary function apply checks what kind of function is referenced by $\nu$ and proceeds accordingly.

**A Concrete Abstract Machine** The machine template can now be instantiated with a store, a match implementation which finds the first matching branch and interpretation for primitive operations in order to obtain a concrete abstract machine. By choosing Store to be a mapping with infinite domain we can ensure that alloc can always return a fresh address. In this setting the store-allocated continuations are just an implementation of a stack. The extra layer of indirection introduced by storing values in a store can also be disregarded as the machine operates on persistent values. Therefore, the resulting machine, which we omit, corresponds to a CEK-style abstract machine which is a canonical formulation for call-by-value functional calculi [16].

**An Abstract Abstract Machine** Let us now turn to a different instantiation of the template. Figure 6 shows the missing pieces of an abstract abstract machine for IDL. The abstract values use base type names $tp$ to represent any
value of that type, abstract versions of primitive operations, records, closures and
top-level functions. The interpretation of primitive operations must approximate
their concrete counterparts.

The store is represented as a pair of finite mappings from labels to se-
sets of abstract values and continuations, respectively. This bounding of store domain and
range ensures that the state-space of the machine becomes finite and therefore
and range ensures that the state-space of the machine becomes finite and therefore
therefore
then can be used for computing an analysis. To retain soundness w.r.t. the concrete
abstract machine the store must map a single address to multiple valu-
es to account for address reuse. This style of abstraction is classical [28] and fairly
straightforward [22]. When instantiated with this store, the transition relation
becomes nondeterministic as pointer dereferencing nondeterministically re-
turns one of the values available in the store. Additionally the implementation of
the match function is also nondeterministic in the choice of the branch to match
again.

This machine is not yet suitable for computing the analysis as the state
space is still too large since every machine configuration has its own copy of
the store. To circumvent this problem a standard technique of widening [28] can
be employed. In particular, following [22], we use a global store. The abstract
configuration $\zeta$ is a pair of a store and a set of partial configurations. The abstract
transition $\Rightarrow_{a}$ performs one step of computation using $\Rightarrow$ on the global store $\sigma$
paired with every partial configuration $\gamma$. The resulting stores $\sigma'$ are merged
together and with the original store to create a new, extended global store. The
partial configurations $C'$ are added to the initial set of configurations $C$. The
transition relation $\Rightarrow_{a}$ is deterministic so it can be treated as a function. This
function is monotone on a finite lattice and therefore is amenable to fixed-point
iteration.

**Computing the Analysis** With the abstract transition function in hand we
can now specify the algorithm for obtaining the analysis. To start the abstract
interpreter we must provide it with an initial configuration: a store, an envi-
Fig. 6. A translation for CPS terms

\[ [x], k = (k x) \]
\[ [y], k = (\text{let } x \leftarrow (k x)) \]
\[ [\langle \text{r x . . . } \rangle], k = (\text{let } y \leftarrow \langle \text{r x . . . } \rangle \langle k y \rangle) \]
\[ [(\text{fun }\#:\text{atomic } \langle \text{x . . . } \rangle e)], k = (\text{let } y \leftarrow \langle \text{fun } \langle \text{x . . . } \rangle \langle e \rangle_k \rangle \langle k y \rangle) \]
\[ [(\text{fun } \langle \text{x . . . } \rangle e)], k = (\text{let } y \leftarrow \langle \text{fun } \langle \text{x . . . } \text{k'} \rangle \langle e \rangle_k \text{k'} \rangle \langle k y \rangle) \]
\[ [(\text{match } x \langle \text{p e . . . } \rangle)], k = (\text{match } x \langle \text{p } \langle e \rangle_k \rangle \langle \text{e} \rangle_k \rangle \langle \text{e} \rangle_k \rangle \langle \text{e} \rangle_k \rangle \langle \text{e} \rangle_k \rangle) \]
\[ [(\text{let } p \langle e \rangle)], k = (\text{let } y \leftarrow \langle \text{fun } \langle \text{y} \rangle \langle \text{let } p \langle e \rangle \langle \text{e} \rangle_k \rangle \langle \text{e} \rangle_k \rangle \langle \text{e} \rangle_k \rangle) \]
\[ [(\text{error } s)], k = (\text{error } s) \]

3.3 Selective CPS transformation

In this section we formulate an algorithm for selectively transforming the program into continuation-passing style. All functions (both anonymous and top-level) marked #:\text{atomic} by the user will be kept in direct style. The main function is implicitly marked as atomic since its interface should be preserved as it is an entry point of the interpreter. Primitive operations are treated as atomic at call-site. Atomic functions may call non-atomic ones by providing the called function an identity continuation. The algorithm uses the results of the control-flow analysis to determine atomicity of functions to which a variable labeled \text{l} in function position may evaluate. If all functions are atomic then \text{allA(l)} holds; if none of them are atomic then \text{noneA(l)} holds. When both atomic and non-atomic functions may be called the algorithm cannot proceed and signals an error in the source program.

The algorithm consists of two mutually recursive transformations. The first, \[ [\langle e \rangle_k], k \] in Figure 6 transforms a term \( e \) into CPS. Its second parameter is a program variable \( k \) which will bind the continuation at runtime. The second, \[ [\langle e \rangle_d], k \] in Figure 7 transforms a term \( e \) which should be kept in direct style.

The first five clauses of the CPS translation deal with values. When a variable is encountered it may be immediately returned by applying a continuation. In
other cases the value must be let-bound in order to preserve the A-normal form of the term and then the continuation is applied to the introduced variable. The body $e$ of an anonymous function is translated using $[e]_d$ when the function is marked atomic. When the function is not atomic a new variable $k'$ is appended to its parameter list and its body is translated using $[e]_c k'$. The form of an application depends on the atomicity of functions which may be applied. When none of them is atomic the continuation $k$ is passed to the function. When all of them are atomic the result of the call is let-bound and returned by applying the continuation $k$. Match expression is transformed by recursing on its branches. Since the continuation is always a program variable no code gets duplicated. When transforming a let expression the algorithm checks whether the bound command $c$ is trivial — meaning it will call only atomic functions when evaluated. If it is, then it can remain in direct style $[c]_d$, no new continuation has to be introduced and the body can be transformed by $[e]_c k$. If the command is non-trivial then a new continuation is created and bound to $k'$. This continuation uses a fresh variable $y$ as its parameter. Its body is the let-expression binding $y$ instead of command $c$ and with body $e$ transformed with the input continuation $k$. The bound command $c$ is transformed with the newly introduced continuation $k'$. Finally, the translation of error throws out the continuation.

The transformation for terms which should be kept in direct style begins similarly to the CPS one — with five clauses for values. In case of an application the algorithm considers two possibilities: when all functions are atomic the call remains in direct style, when none of them are atomic a new identity continuation $k$ is constructed and is passed to the called function. A match expression is again transformed recursively. A let binding of a call to a CPS function gets special treatment to preserve A-normal form by chaining allocation of identity continuation with the call. In other cases a let binding is transformed recursively. An error expression is left untouched.

Each top-level function definition in a program is transformed in the same fashion as anonymous functions. After the transformation the program is still
in ANF and can be again analyzed by the abstract abstract machine of the previous section. CPS-transforming the direct-style interpreter of Figure 1 yields an interpreter in CPS shown in Figure 8 (after let-inlining for readability), where we assume that the operations on environments were marked as atomic and therefore have not changed.

3.4 Selective Defunctionalization

The second step of the functional correspondence and the last stage of the transformation is selective defunctionalization. The goal is to defunctionalize function spaces deemed interesting by the author of the program. To this end top-level and anonymous functions may be annotated with #:no-defun to skip defunctionalization of function spaces they belong to. In the algorithm of Figure 9 the predicate defun specifies whether a function should be transformed. Predicates primOp and topLevel specify whether a variable refers to (taking into account the scoping rules) primitive operation or top-level function, respectively. There are three cases to consider when transforming an application. If the variable in operator position refers to top-level function or primitive operation it can be left as is. Otherwise we can utilize the results of control-flow analysis to obtain the set of functions which may be applied. When all of them should be defunctionalized (allDefun) then a call to the generated apply function is introduced, when none of them should (noneDefun) then the application is left as is. If the requirements are mixed then an error in the source program is signaled. To transform an abstraction, its free variables (fvs(l)) are collected into a record. The apply functions are generated using mkApply as specified in Figure 10 where the fn... is a list of functions which may be applied. After the transformation the program is no longer in A-normal form since variables referencing top-level functions may have been transformed into records. However it does not pose a problem since the majority of work has already been done and the last step – let-inlining does not require the program to be in ANF. Defunctionalizing the CPS interpreter of Figure 8 and performing let-inlining yields an encoding of the CEK abstract machine shown in Figure 11 (again, the environment is left intact).
\[ \lfloor x \rfloor = \begin{cases} \{ \text{Prim} \} & \text{when } \text{primOp}(x) \\ \{ \text{Top} \} & \text{when } \text{topLevel}(x) \land \text{defun}(x) \\ x & \text{otherwise} \end{cases} \]

\[ \lfloor b \rfloor = b \]

\[ \lfloor \{ r \, x \, \ldots \} \rfloor = \{ r \lfloor x \rfloor \ldots \} \]

\[ \lfloor \{ \text{Fun} \, fvs(l) \} \rfloor = \{ \text{Fun} \, fvs(l) \} \]

\[ \lfloor \{ \text{match} \, x \, (p \, e) \ldots \} \rfloor = \{ \text{match} \, x \, (p \lfloor e \rfloor \ldots) \} \]

\[ \lfloor \{ \text{let} \, p \, c \, e \} \rfloor = \{ \text{let} \, p \lfloor c \rfloor \lfloor e \rfloor \} \]

\[ \lfloor \{ \text{error} \, s \} \rfloor = \{ \text{error} \, s \} \]

Fig. 9. Selective defunctionalization algorithm for IDL

\[
mkBranch(x \ldots, \delta) = \{ \{ \text{Prim} \} \ (\delta \ x \ldots) \} \]
\[
mkBranch(x \ldots, \{ \text{def} \ l \ (y \ldots) \ e \}) = \{ \{ \text{Top} \} \ (l \ x \ldots) \} \]
\[
mkBranch(x \ldots, \{ \text{fun} \ (y \ldots) \ e \}) = \{ \{ \text{Fun} \, fvs(l) \} \ [e] \ [y \mapsto \ x] \} \]
\[
mkApply(l, fn\ldots) = \{ \{ \text{def apply!} \ (l \ x \ldots) \}
\]
\[ \quad \text{mkBranch}(x \ldots, fn\ldots) \} \}
\]

Fig. 10. Top-level apply function generation

4 Case Studies

We studied the efficacy of the algorithm and the implementation on a number of programming language calculi. Figure 12 shows a summary of interpreters on which we tested the transformer. The first group of interpreters is denotational (mostly meta-circular) in style and covers various extensions of the base \(\lambda\)-calculus with call-by-value evaluation order. The additions we tested include: integers with addition, recursive let-bindings, delimited control operators – shift and reset with CPS interpreter based on [9] and exceptions in two styles: monadic with exceptions as values (functions return either value or an exception) and in CPS with success and error continuations. The last interpreter for call-by-value in Figure 12 is a normalization function based on normalization by evaluation technique transcribed from [1]. We find this result particularly satisfactory, since it leads to a non-trivial and previously unpublished abstract machine – we give more details in Appendix [19]. The next three interpreters correspond to big-step operational semantics for call-by-name \(\lambda\)-calculus, call-by-need (call-by-name with memoization) and a simple imperative language, respectively.

Transformation of call-by-value and call-by-need \(\lambda\)-calculus yielded machines very similar to the CEK and Krivine machines, respectively. We were also able to replicate the machines previously obtained via manual application of the func-
(def-data Cont
  {Halt}
  {App1 arg env cont}
  {App2 fn cont})

(def-struct {Closure body env x})

(def eval (env term cont)
  (match term
    ([String x] (continue cont (env x)))
    ({Abs x body} (continue cont (Closure body env x)))
    ({App fn arg} (eval env fn {App1 arg env cont}))))

(def apply (fn v cont)
  (let {Fun body env x} fn)
    (eval (extend env x v) body cont))

(def continue (cont val)
  (match cont
    ({Halt} val))
    ({App1 arg env cont} (eval env arg {App2 val cont}))
    ({App2 fn cont} (apply fn val cont)))

(def main ([Term term]) (eval {Init} term {Halt}))

Fig. 11. An encoding of the CEK machine for λ-calculus

The biggest differences were due to introduction of administrative transitions in handling of applications. This property hints at a potential for improvement by introducing an inlining step to the transformation. An interesting feature of the transformation is the ability to select which parts of the interpreter should be transformed and which should be considered atomic. These choices are reflected in the resulting machine, e.g., by transforming an environment look up in call-by-need interpreter we obtain a Krivine machine which has the search for a value in the environment embedded in its transition rules, while marking it atomic gives us a more abstract formulation from [4]. Another consequence of this feature is that one can work with interpreters already in CPS and essentially skip directly to defunctionalization (as tested on micro-Prolog interpreter of [11]).

5 Conclusion

In this article we described an algorithm, based on the functional correspondence [4,9,11], that allows for automatic derivation of an abstract machine given an interpreter which typically corresponds to denotational or natural semantics, allowing the user for fine-grained control over the shape of the resulting machine.
| Language       | Interpreter style | Lang. Features                          | Result                  |
|---------------|------------------|----------------------------------------|-------------------------|
| call-by-value | denotational     | ·                                       | CEK machine             |
| λ-calculus    | denotational     | integers with add                      | CEK with add            |
| call-by-value | denotational,   | integers, recursive let-bindings       | similar to Reynolds's   |
|               | recursion via   |                                         | first-order interpreter |
|               | environment     |                                         |                         |
| call-by-value | denotational    | shift and reset                        | two layers of conts.    |
|               | with conts.     |                                         |                         |
| call-by-value | denotational,   | exceptions                              | explicit                |
|               | monadic         |                                         | stack unwinding         |
|               | CPS             |                                         | pointer to             |
|               | normalization   |                                         | exception handler       |
|               | by evaluation   |                                         |                         |
| call-by-name  | big-step        | ·                                       | strong CEK machine      |
| λ-calculus    |                 |                                         |                         |
| call-by-need  | big-step        | memoization                             | lazy Krivine machine    |
| λ-calculus    | (state passing) |                                         |                         |
| simple        | big-step        | conditionals,                          |                         |
| imperative    | (state passing) | while, assignment                       |                         |
| micro-Prolog  | CPS             | backtracking,                          | logic engine            |
|               |                 | cut operator                            |                         |

**Fig. 12.** Summary of tested interpreters

In order to enable the transformation we derived a control-flow analysis for IDL using the abstracting abstract machines methodology. We implemented the algorithm in the Haskell programming language and used this tool to transform a selection of interpreters. To the best of our knowledge this is the first, reasonably generic, implementation of the functional correspondence.

The correctness of the tool relies on the correctness of each of the program transformations involved in the derivation that are classic and in some form have been proven correct in the literature [30][27][26][7], as well as on the correctness of the control-flow analysis we take advantage of. An extensive number of experiments we have carried out indicates that the tool indeed is robust.

In order to improve the capabilities of semt as a practical tool for semantics engineering, the future work could include extending the set of primitive operations and adding the ability to import arbitrary Racket functions and provide their abstract specification. The tool could also be extended to accommodate other output formats such as \LaTeX\ figures or low level C code [19].

Another avenue for improvement lies in extensions of the meta-language capabilities. Investigation of additions such as control operators, nondeterministic choice and concurrency could yield many opportunities for diversifying the set of interpreters (and languages) that may be encoded in the IDL. In particular control operators could allow for expressing the interpreter for a language with delimited control (or algebraic effects [12][21]) in direct style.
References

1. Abel, A.: Normalization by Evaluation: Dependent Types and Impredicativity. Habilitation thesis, LMU (2013)
2. Ager, M.S.: From natural semantics to abstract machines. In: Etalle, S. (ed.) Logic Based Program Synthesis and Transformation, 14th International Symposium, LOPSTR 2004, Verona, Italy, August 26-28, 2004, Revised Selected Papers. Lecture Notes in Computer Science, vol. 3573, pp. 245–261. Springer (2004). https://doi.org/10.1007/11506676_16
3. Ager, M.S., Biernacki, D., Danvy, O., Midtgaard, J.: From interpreter to compiler and virtual machine: a functional derivation. BRICS Report Series 10(14) (2003)
4. Ager, M.S., Biernacki, D., Danvy, O., Midtgaard, J.: A functional correspondence between evaluators and abstract machines. In: Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden. pp. 8–19. ACM (2003). https://doi.org/10.1145/888251.888254
5. Ager, M.S., Danvy, O., Midtgaard, J.: A functional correspondence between call-by-need evaluators and lazy abstract machines. Inf. Process. Lett. 90(5), 223–232 (2004). https://doi.org/10.1016/j.ipl.2004.02.012
6. Ager, M.S., Danvy, O., Midtgaard, J.: A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. Theor. Comput. Sci. 342(1), 149–172 (2005). https://doi.org/10.1016/j.tcs.2005.06.008
7. Banerjee, A., Heintze, N., Riecke, J.G.: Design and correctness of program transformations based on control-flow analysis. In: Kobayashi, N., Pierce, B.C. (eds.) Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001, Sendai, Japan, October 29-31, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2215, pp. 420–447. Springer (2001). https://doi.org/10.1007/3-540-45500-0_21
8. Biernacka, M., Biernacki, D., Charatonik, W., Drab, T.: An abstract machine for strong call by value. In: d. S. Oliveira, B.C. (ed.) Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12470, pp. 147–166. Springer (2020). https://doi.org/10.1007/978-3-030-45500-0_8
9. Biernacka, M., Biernacki, D., Danvy, O.: An operational foundation for delimited continuations in the CPS hierarchy. Log. Methods Comput. Sci. 1(2) (2005). https://doi.org/10.2168/LMCS-1(2:5)2005
10. Biernacka, M., Charatonik, W., Zielinska, K.: Generalized refocusing: From hybrid strategies to abstract machines. In: Miller, D. (ed.) 2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK. LIPIcs, vol. 84, pp. 10:1–10:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). https://doi.org/10.4230/LIPIcs.FSCD.2017.10
11. Biernacki, D., Danvy, O.: From interpreter to logic engine by defunctionalization. In: Bruynooghe, M. (ed.) Logic Based Program Synthesis and Transformation, 13th International Symposium LOPSTR 2003, Uppsala, Sweden, August 25-27, 2003, Revised Selected Papers. Lecture Notes in Computer Science, vol. 3018, pp. 143–159. Springer (2003). https://doi.org/10.1007/978-3-540-25938-1_13
12. Biernacki, D., Piróg, M., Polesiuk, P., Sieczkowski, F.: Abstracting algebraic effects. Proc. ACM Program. Lang. 3(POPL), 6:1–6:28 (2019). https://doi.org/10.1145/3290319
13. Crégut, P.: An abstract machine for lambda-terms normalization. In: Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990. pp. 333–340. ACM (1990). https://doi.org/10.1145/91556.91681

14. Danvy, O., Johannsen, J.: Inter-deriving semantic artifacts for object-oriented programming. J. Comput. Syst. Sci. 76(5), 302–323 (2010). https://doi.org/10.1016/j.jcss.2009.10.004

15. Danvy, O., Nielsen, L.R.: Refocusing in reduction semantics. BRICS Report Series 11(26) (2004)

16. Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex. MIT Press (2009)

17. Felleisen, M., Friedman, D.P.: Control operators, the SECD-machine, and the λ-calculus. In: Wirsing, M. (ed.) Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III, Elsberup, Denmark, 25-28 August 1986. pp. 193–222. North-Holland (1987)

18. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: Cartwright, R. (ed.) Proceedings of the ACM SIGPLAN’93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993. pp. 237–247. ACM (1993). https://doi.org/10.1145/155090.155113

19. Friedman, D.P., Wand, M.: Essentials of programming languages (3. ed.). MIT Press (2008)

20. Hannan, J., Miller, D.: From operational semantics to abstract machines. Mathematical Structures in Computer Science 2(4), 415–459 (1992). https://doi.org/10.1017/S0960129500001559

21. Hillerström, D., Lindley, S.: Liberating effects with rows and handlers. In: Chapman, J., Swierstra, W. (eds.) Proceedings of the 1st International Workshop on Type-Driven Development, TyDe@ICFP 2016, Nara, Japan, September 18, 2016. pp. 15–27. ACM (2016). https://doi.org/10.1145/2976022.2976033

22. Horn, D.V., Might, M.: Abstracting abstract machines. In: Hudak, P., Weirich, S. (eds.) Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010. pp. 51–62. ACM (2010). https://doi.org/10.1145/1863543.1863553

23. Jedynak, W., Biernacka, M., Biernacki, D.: An operational foundation for the tactic language of Coq. In: Peña, R., Schrijvers, T. (eds.) 15th International Symposium on Principles and Practice of Declarative Programming, PPDP ’13, Madrid, Spain, September 16-18, 2013. pp. 25–36. ACM (2013). https://doi.org/10.1145/2505879.2505890

24. Krivine, J.: A call-by-name lambda-calculus machine. High. Order Symb. Comput. 20(3), 199–207 (2007). https://doi.org/10.1007/s10990-007-9018-9

25. Landin, P.J.: The mechanical evaluation of expressions. Comput. J. 6(4), 308–320 (1964). https://doi.org/10.1093/comjnl/6.4.308

26. Nielsen, L.: A denotational investigation of defunctionalization. BRICS Report Series 7 (09 2010). https://doi.org/10.7146/brics.v7i47.20214

27. Nielsen, L.R.: A selective CPS transformation. In: Brookes, S.D., Mislove, M.W. (eds.) Seventeenth Conference on the Mathematical Foundations of Programming Semantics, MFPS 2001, Aarhus, Denmark, May 23-26, 2001. Electronic Notes in Theoretical Computer Science, vol. 45, pp. 311–331. Elsevier (2001). https://doi.org/10.1016/S1571-0661(04)80969-1
28. Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis. Springer (1999). https://doi.org/10.1007/978-3-662-03811-6

29. Piróg, M., Biernacki, D.: A systematic derivation of the STG machine verified in coq. In: Gibbons, J. (ed.) Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell, Haskell 2010, Baltimore, MD, USA, 30 September 2010. pp. 25–36. ACM (2010). https://doi.org/10.1145/1863523.1863528

30. Plotkin, G.D.: Call-by-name, call-by-value and the lambda-calculus. Theor. Comput. Sci. 1(2), 125–159 (1975). https://doi.org/10.1016/0304-3975(75)90017-1

31. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. High. Order Symb. Comput. 11(4), 363–397 (1998). https://doi.org/10.1023/A:1010027404223

32. Sieczkowski, F., Biernacka, M., Biernacki, D.: Automating derivations of abstract machines from reduction semantics: - A generic formalization of refocusing in Coq. In: Hage, J., Morazán, M.T. (eds.) Implementation and Application of Functional Languages - 22nd International Symposium, IFL 2010, Alphen aan den Rijn, The Netherlands, September 1-3, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6647, pp. 72–88. Springer (2010). https://doi.org/10.1007/978-3-642-24276-2_5
A Primer on the Functional Correspondence

In this section we present a simple illustrative example for the reader unfamiliar with CPS transformations and/or defunctionalization applied in program development. We use IDL as our meta-language, and we perform the two transformations by hand, without using our tool automating the functional correspondence.

In this example, our object-language is the built-in data type of integers, and the interpreter we are going to transform interprets a given natural number as its factorial (the negative integers are arbitrarily mapped to 1):

\[
\text{(def factorial (} n \text{)} \\text{match (< 0 n) \text{(#t (* n (factorial (- n 1)))) (#f 1))})
\]

\[
\text{(def main ([} Integer n \text{]) (factorial n))}
\]

The familiar factorial function is written in direct style, with no explicit mention of the return stack and with a nested recursive call. The main function is the entry point for the computation.

Let us CPS transform factorial. To this end, we introduce a functional parameter cont – a continuation – that represents the rest of the computation before factorial returns a value to main. Returning a value from the function is expressed by passing it to the continuation ((cont 1)). The nested recursive function call becomes a tail call by passing the function a continuation (fun (var) (cont (* n var))). The initial continuation is the identity function – once factorial completes, it returns the final result. Here is the CPS version of the program:

\[
\text{(def factorial (} n \text{ cont) \text{match (< 0 n) \text{(#t (factorial (- n 1)) (fun (var) (cont (* n var)))) (#f (cont 1))})}
\]

\[
\text{(def main ([} Integer n \text{]) (factorial n (fun (x) x)))}
\]

Next, we defunctionalize the continuations. Defunctionalization consists in replacing a function space with a first-order data type and a function interpreting the constructors of this data type. Each constructor represents a function introduction in the defunctionalized function space; the arguments of the constructor are the values of the free variables of the corresponding function.

In our case there are two constructors, one for the continuation in the recursive call that should remember the values of the variables cont and n (call it Cont), and a 0-argument one for the initial continuation (call it Halt). We then introduce the continue function that interprets the constructors accordingly and we replace calls to continuations with calls to this function. The resulting first-order program reads as follows:
(def-struct {Cont cont n})
(def-struct {Halt })

(def continue (fn var)
  (match fn
    ({Cont cont n} (continue cont (* n var)))
    ({Halt } var)))

(def factorial (n cont)
  (match (< 0 n)
    (#t (factorial (- n 1) {Cont cont n}))
    (#f (continue cont 1))))

(def main ([Integer n]) (factorial n {Halt}))

What we have obtained is a functional encoding of an abstract machine. The machine operates in two modes: factorial and continue. The mutually tail-recursive calls model machine transitions. The data type of the defunctionalized continuation, isomorphic with a list of integers, represents the stack of the machine. The machine did not have to be invented, but instead it was mechanically derived. This is a very simple example of the general phenomenon known as the functional correspondence that applies to evaluators of virtually arbitrary complexity.

B Normalization by Evaluation for $\lambda$-calculus

In this section we present a more involved case study: deriving a previously unknown abstract machine from a normalization function for $\lambda$-calculus. The normalization function is shown in Figure[13] It is based on the technique called normalization by evaluation, and this particular definition has been adapted from [1]. The main idea is to use standard evaluator for call-by-value $\lambda$-calculus to evaluate terms to values and then reify them back into terms.

The terms use de Bruijn indices to represent bound variables. Since normalization requires reduction under binders the evaluator must work with open terms. We use de Bruijn levels (Level) to model variables in open terms. The eval function as usual transforms a term in a given environment into a value which is represented as a function wrapped in a Fun record. The values also include Levels and Terms which are introduced by the reify function. The apply function handles both the standard case of applying a functional value (case Fun) and the non-standard one which occurs during reification of the value and amounts to emitting the syntax node for application. The reification function (reify) turns a value back into a term. When its argument is a Fun it applies the function f to a Level representing unknown variable. When reified, a Level is turned back into de Bruijn index. Lastly, reification of an (syntactic) application proceeds recursively. The main function first evaluates a term in an empty environment and then reifies it back into its normal form. As usual, we keep
the environment implementation unchanged during the transformation and we annotate the functional values to be named Closure.

The transformed normalization function is presented in Figure 13. We notice that the machine has two classes of continuations. The first set (handled by continue1) is responsible for the control-flow of reification procedure. The second set (handled by continue) is responsible for the control-flow of evaluation and for switching to reification mode. We observe that the stack used by the

Fig. 13. A normalization function for call-by-value λ-calculus
Fig. 14. A strong call-by-value machine for $\lambda$-calculus

machine consists of a prefix of only evaluation frames and a suffix of only reification frames. The machine switches between evaluation and reification in three places. In line 3 reification of a closure requires evaluation of its body therefore machine uses apply1 to evaluate the closure with a Level as an argument. The switch in other direction is due to evaluation finishing: in line 32 a closure’s body has been evaluated and has to be reified and then enclosed in an Abs (enforced
by the \texttt{Fun2} frame); in line 35 the initial term has been reduced and the value can be reified.

The machine we obtained, to our knowledge, has not been described in the literature. It is somewhat similar to the one mechanically obtained by Ager et al. \cite{ager01b} who also used the functional correspondence to derive the machine. Their machine uses meta-language with mutable state in order to generate fresh identifiers for variables in open terms instead of de Bruijn levels and it operates on compiled rather than source terms. The machine we obtained using \texttt{semt} is as legible as the one derived manually.