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

Maciej Buszka and Dariusz Biernacki(✉)

Faculty of Mathematics and Computer Science, University of Wrocław, Wrocław, Poland
{maciej.buszka,dabi}@cs.uni.wroc.pl

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) [19,31]. These definitional interpreters can be placed on a spectrum from most abstract to most explicit.

E. De Angelis and W. Vanhoof (Eds.): LOPSTR 2021, LNCS 13290, pp. 38–59, 2022.
https://doi.org/10.1007/978-3-030-98869-2_3

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 λ-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 Crégut's machine [13] for normalization of λ-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 [2,10,15,20,32]. 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 [31] transformation to continuation-passing style[1] 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 λ-calculus evaluation [4] and normalization [8], 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

[1] The transformation used by Reynolds was later formalized by Plotkin as call-by-value CPS translation [30].

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 treating such transformations [7,27], 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) [22] 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) [18] 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 idependent 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://bitbucket.org/pl-uwr/semantic-transformer

The rest of this article is structured as follows: In Sect. 2, we introduce the *Interpreter Definition Language* which is the meta-language accepted by the transformer and will be used in example evaluators throughout the paper. In Sect. 3, we present the algorithmic characterization of the functional correspondence. In Sect. 4, we briefly discuss the performance of the tool on a selection of case studies. In Sect. 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

```
(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

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

Fig. 2. Abstract syntax of the *IDL* terms

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.

As shown in Fig. 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: `String`s 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 `Term`s 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 Fig. 1 takes advantage of the functional representation of environments (`init` and `extend`) and it structurally recursively interprets λ-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 Fig. 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 destructure bound 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) [18] 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 Sect. 3.2. The abstract syntax of terms in ANF and an algorithm for transforming *IDL* programs into such form is presented in Fig. 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.

$$
\begin{array}{rl}
Com \ni c & ::= x \mid b \mid (\texttt{fun}\ (x \ldots)\ e) \mid (x\ x \ldots) \\
 & \mid \{r\ x \ldots\} \mid (\texttt{match}\ x\ (p\ e) \ldots) \\
Anf \ni e & ::= c \mid (\texttt{let}\ p\ c\ e) \mid (\texttt{error}\ s)
\end{array}
$$

$$
\begin{array}{rcl}
[\![\cdot]\!]\ \cdot & : & Term \times (Com \to Anf) \to Anf \\
[\![x]\!]\ k & = & k\,x \\
[\![b]\!]\ k & = & k\,b \\
[\![(\texttt{fun}\ (x \ldots)\ e)]\!]\ k & = & k\,(\texttt{fun}\ (x \ldots)\ [\![e]\!]\,id) \\
[\![(e_f\ e_{arg} \ldots)]\!]\ k & = & [\![e_f]\!]\,[\lambda x_f.\,[\![e_{arg} \ldots]\!]_s\,\lambda(x_{arg} \ldots).k\,(x_f\ x_{arg} \ldots)]_a \\
[\![(\texttt{let}\ p\ e_1\ e_2)]\!]\ k & = & [\![e_1]\!]\,\lambda c_1.(\texttt{let}\ p\ c_1\ [\![e_2]\!]\,k) \\
[\![\{r\ e \ldots\}]\!]\ k & = & [\![e \ldots]\!]_s\,\lambda(x \ldots).k\,\{r\ x \ldots\} \\
[\![(\texttt{match}\ e\ (p\ e_b))]\!]\ k & = & [\![e]\!]\,[\lambda x.k\,(\texttt{match}\ x\ (p\ [\![e_b]\!]\,id)]_a \\
[\![(\texttt{error}\ s)]\!]\ \_ & = & (\texttt{error}\ s)
\end{array}
$$

$$
\begin{array}{rcl}
[\cdot]_a\ \cdot & : & (Var \to Anf) \to Com \to Anf \\
[k]_a\ x & = & k\,x \\
[k]_a\ c & = & (\texttt{let}\ x\ c\ (k\,x))
\end{array}
$$

$$
\begin{array}{rcl}
[\![\cdot]\!]_s\ \cdot & : & Term^* \times (Var^* \to Anf) \to Anf \\
[\![e \ldots]\!]_s\ k & = & go(e \ldots, \epsilon, k) \\
go(\epsilon, x \ldots, k) & = & k\,(x \ldots) \\
go(e\,e_r \ldots, x_{acc} \ldots, k) & = & [\![e]\!]\,[\lambda x.go(e_r \ldots, x_{acc} \ldots x, k)]_a
\end{array}
$$

Fig. 3. ANF transformation for *IDL*

The $[\![\cdot]\!]\cdot$ function, written in CPS[2], is the main transformation function. Its arguments are a term to be transformed and a meta-language continuation 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 $[\cdot]_a$ 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 $[\![\cdot]\!]_s\cdot$ sequences computation of multiple expressions by creating a chain of let-bindings (using $[\cdot]_a$) 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

[2] See Appendix A of [18].

$$
\begin{aligned}
\nu \in VAddr\ \kappa \in KAddr \quad & l \in Label \quad \sigma \in Store \\
\delta \in PrimOp \subseteq Val^* \to Val \quad & \\
\rho \in Env = Var \to VAddr \quad & \\
Val \ni v ::= b \mid \delta \mid \{r\ \nu \ldots\} \mid \langle \rho, x \ldots, e \rangle \mid & \texttt{(def}\ x\ \texttt{(}x \ldots\texttt{)}\ e\texttt{)} \\
Cont \ni k ::= \langle \rho, p, e, \kappa \rangle \mid \langle\rangle \quad & \\
PartialConf \ni \gamma ::= \langle \rho, e, \kappa \rangle_e \mid \langle \nu, \kappa \rangle_c \quad & \\
Conf \ni \varsigma ::= \langle \sigma, \gamma \rangle &
\end{aligned}
$$

$$
\begin{aligned}
\langle \sigma, \langle \rho, x, \kappa \rangle_e \rangle &\Rightarrow \langle copy_v(\rho(x), l, \sigma), \langle \rho(x), \kappa \rangle_c \rangle \\
\langle \sigma, \langle \rho, b^l, \kappa \rangle_e \rangle &\Rightarrow \langle \sigma', \langle \nu, \kappa \rangle_c \rangle \\
&\quad \text{where } \langle \sigma', \nu \rangle = alloc_v(b, l, \sigma) \\
\langle \sigma, \langle \rho, \{r\ x \ldots\}^l, \kappa \rangle_e \rangle &\Rightarrow \langle \sigma', \langle \nu, \kappa \rangle_c \rangle \\
&\quad \text{where } \langle \sigma', \nu \rangle = alloc_v(\{r\ \rho(x) \ldots\}, l, \sigma) \\
\langle \sigma, \langle \rho, \texttt{(fun (}x \ldots\texttt{)}e\texttt{)}^l, \kappa \rangle_e \rangle &\Rightarrow \langle \sigma', \langle \nu, \kappa \rangle_c \rangle \\
&\quad \text{where } \langle \sigma', \nu \rangle = alloc_v(\langle \rho, x \ldots, e \rangle, l, \sigma) \\
\langle \sigma, \langle \rho, \texttt{(let}\ p\ c^l\ e\texttt{)}, \kappa \rangle_e \rangle &\Rightarrow \langle \sigma', \langle \rho, c, \kappa' \rangle_e \rangle \\
&\quad \text{where } \langle \sigma', \kappa' \rangle = alloc_k(\langle \rho, p, e, \kappa \rangle, l, \sigma) \\
\langle \sigma, \langle \rho, \texttt{(}x\ y \ldots\texttt{)}, \kappa \rangle_e \rangle &\Rightarrow apply(\sigma, \rho(x), \rho(y) \ldots, l) \\
\langle \sigma, \langle \rho, \texttt{(match}\ x\ \texttt{(}p\ e\texttt{)} \ldots\texttt{)}, \kappa \rangle_e \rangle &\Rightarrow match(\sigma, \rho, \rho(x), \langle p, e \rangle \ldots) \\
\langle \sigma, \langle \nu, \kappa \rangle_c \rangle &\Rightarrow match(\sigma, \rho, \nu, \kappa', \langle p, e \rangle) \\
&\quad \text{where } \langle \rho, p, e, \kappa' \rangle = deref_k(\sigma, \kappa)
\end{aligned}
$$

$$
apply(\sigma, \nu, \nu' \ldots, \kappa, l) = \begin{cases}
\langle \sigma, \langle \rho[(x \mapsto \nu') \ldots], e, \kappa \rangle_e \rangle \\
\quad \text{when } deref_v(\sigma, \nu) = \langle \rho, x \ldots, e \rangle \\
\langle \sigma, \langle \rho_0[(x \mapsto \nu') \ldots], e, \kappa \rangle_e \rangle \\
\quad \text{when } deref_v(\sigma, \nu) = \texttt{(def}\ y\ \texttt{(}x \ldots\texttt{)}\ e\texttt{)} \\
\langle \sigma', \langle \nu'', \kappa \rangle_c \rangle \\
\quad \text{when } deref_v(\sigma, \nu) = \delta \\
\quad \text{and } \langle \sigma', \nu'' \rangle = alloc_v(\delta(\sigma(\nu') \ldots), l, \sigma)
\end{cases}
$$

$$
match(\sigma, \rho, \nu, \kappa, \langle p, e \rangle \ldots) = \langle \sigma, \langle \rho', e', \kappa \rangle_e \rangle \text{ where } \rho' \text{ is the environment for the first matching branch with body } e'
$$

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

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 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 Fig. 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 σ along with five operations: $alloc_v$, $alloc_k$, $deref_v$, $deref_k$ and $copy_v$; interpretation of

primitive operations δ and implementation of *match* function which interprets pattern matching. The store maps value addresses ν to values v and continuation addresses κ 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 δ, 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 ρ, a pattern p to use for destructuring a value, the body e of a let expression and a pointer to the next continuation κ. 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 σ and a partial configuration γ. 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 ρ, an expression e and a continuation pointer κ. A continuation configuration holds an address ν of a value that has been computed so far and a pointer κ 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 ρ 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 *alloc*ating 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 κ'. 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 κ is a frame. In such a case the machine matches the stored pattern against the value pointed-to by ν. Otherwise κ 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 ν 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].

$$
\begin{aligned}
VAddr &= KAddr = Label \\
\widetilde{Val} \ni v &::= tp \mid \widetilde{\delta} \mid \{r\ \nu \ldots\} \mid \langle \rho, x \ldots, e \rangle \mid (\texttt{def}\ x\ (x \ldots)\ e) \\
\sigma \in Store &= (VAddr \to \mathbb{P}(\widetilde{Val})) \times (KAddr \to \mathbb{P}(Cont)) \\
alloc_v(v, l, \langle \sigma_v, \sigma_k \rangle) &= \langle \langle \sigma_v[l \mapsto \sigma_v(l) \cup \{v\}], \sigma_k \rangle, l \rangle \\
alloc_k(v, l, \langle \sigma_v, \sigma_k \rangle) &= \langle \langle \sigma_v, \sigma_k[l \mapsto \sigma_k(l) \cup \{k\}] \rangle, l \rangle \\
copy_v(\nu, l, \langle \sigma_v, \sigma_k \rangle) &= \langle \sigma_v[l \mapsto \sigma_v(l) \cup \sigma_v(\nu)], \sigma_k \rangle \\
deref_v(l, \langle \sigma_v, \sigma_k \rangle) &= \sigma_v \\
\tilde{\varsigma} \in \widetilde{Conf} &= Store \times \mathbb{P}(PartialConf) \\
\hline
\langle \sigma, C \rangle &\Rightarrow_a \langle \sigma' \sqcup \sigma, C \cup C' \rangle \\
&\quad \text{where } \sigma' = \bigsqcup \{\sigma' \mid \exists \gamma \in C.\, \langle \sigma, \gamma \rangle \Rightarrow \langle \sigma', \gamma' \rangle\} \\
&\quad \text{and } C' = \{\gamma' \mid \exists \gamma \in C.\, \langle \sigma, \gamma \rangle \Rightarrow \langle \sigma', \gamma' \rangle\}
\end{aligned}
$$

Fig. 5. An abstract abstract machine for *IDL*

An Abstract Abstract Machine. Let us now turn to a different instantiation of the template. Figure 5 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 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 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 values to account for address reuse. This style of abstraction is classical [28] and fairly straightforward [22]. When instantiated with this store, the transition relation $\Rightarrow$ becomes nondeterministic as pointer *deref*erencing nondeterministically returns 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 against.

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 $\tilde{\varsigma}$ 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 σ paired with every partial configuration γ. The resulting stores σ' 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.

$$
\begin{aligned}
[\![x]\!]_c\, k &= (k\ x) \\
[\![b]\!]_c\, k &= (\texttt{let}\ x\ b\ (k\ x)) \\
[\![\{r\ x\ldots\}]\!]_c\, k &= (\texttt{let}\ y\ \{r\ x\ldots\}\ (k\ y)) \\
[\![(\texttt{fun}\ \texttt{\#:atomic}\ (x\ldots) e)]\!]_c\, k &= (\texttt{let}\ y\ (\texttt{fun}\ (x\ldots)\ [\![e]\!]_d)\ (k\ y)) \\
[\![(\texttt{fun}\ (x\ldots) e)]\!]_c\, k &= (\texttt{let}\ y\ (\texttt{fun}\ (x\ldots k')\ [\![e]\!]_c\, k')\ (k\ y)) \\
[\![(f^l\ x\ldots)]\!]_c\, k &= \begin{cases} (f\ x\ldots\ k) & \text{when } noneA(l) \\ (\texttt{let}\ y\ (f\ x\ldots)\ (k\ y)) & \text{when } allA(l) \end{cases} \\
[\![(\texttt{match}\ x\ (p\ e)\ldots)]\!]_c\, k &= (\texttt{match}\ x\ (p\ [\![e]\!]_c\, k)\ldots) \\
[\![(\texttt{let}\ p\ c\ e)]\!]_c\, k &= \begin{cases} (\texttt{let}\ p\ [\![c]\!]_d\ [\![e]\!]_c\, k) & \text{when } trivial(c) \\ (\texttt{let}\ k'\ (\texttt{fun}\ (y)\ (\texttt{let}\ p\ y\ [\![e]\!]_c\, k))\ [\![c]\!]_c\, k') \end{cases} \\
[\![(\texttt{error}\ s)]\!]_c\, k &= (\texttt{error}\ s)
\end{aligned}
$$

Fig. 6. A translation for CPS terms

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 environment, a term and a continuation pointer. The store will be assembled from datatype and structure definitions of the program as well as base types. The initial term is the body of the `main` function of the interpreter and the environment is the global environment extended with `main`'s parameters bound to pointers to datatypes in the above-built store. The initial continuation is of course $\langle\rangle$ and the pointer is the label of the body of the `main` function. The analysis is computed by performing fixed-point iteration of $\Rightarrow_a$. The resulting store will contain a set of functions to which every variable (the only allowed term) in function position may evaluate (ensured by the use of $copy_v$ function). This result will be used in Sects. 3.3 and 3.4.

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 `#: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 l in function position may evaluate. If all functions are atomic then $allA(l)$ holds; if none of them are atomic then $noneA(l)$ 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, $[\![e]\!]_c k$ in Fig. 6 transforms a term e into CPS. Its second parameter is a program variable k which will bind the continuation at runtime. The second, $[\![e]\!]_d$ in Fig. 7 transforms a term e which should be kept in direct style.

$$
\begin{aligned}
[\![x]\!]_d &= x \\
[\![b]\!]_d &= b \\
[\![\{r\ x\ldots\}]\!]_d &= \{r\ x\ldots\} \\
[\![(\texttt{fun}\ \texttt{\#:atomic}\ (x\ldots)e)]\!]_d &= (\texttt{fun}\ (x\ldots)\ [\![e]\!]_d) \\
[\![(\texttt{fun}\ (x\ldots)e)]\!]_d &= (\texttt{fun}\ (x\ldots k')\ [\![e]\!]_c\, k') \\
[\![(f^l\ x\ldots)]\!]_d &= \begin{cases} (f\ x\ldots) & \text{when } allA(l) \\ (\texttt{let}\ k\ (\texttt{fun}\ (y)\ y)\ (f\ x\ldots\ k)) & \text{when } noneA(l) \end{cases} \\
[\![(\texttt{match}\ x\ (p\ e)\ldots)]\!]_d &= (\texttt{match}\ x\ (p\ [\![e]\!]_d)\ldots) \\
[\![(\texttt{let}\ p\ (f^l\ y\ldots)\ e)]\!]_d &= \begin{array}{l}(\texttt{let}\ k\ (\texttt{fun}\ (z)\ z) \\ \quad (\texttt{let}\ p\ (f\ y\ldots\ k)\ [\![e]\!]_d))\end{array} \quad \text{when } noneA(l) \\
[\![(\texttt{let}\ p\ c\ e)]\!]_d &= (\texttt{let}\ p\ [\![c]\!]_d\ [\![e]\!]_d) \\
[\![(\texttt{error}\ s)]\!]_d &= (\texttt{error}\ s)
\end{aligned}
$$

Fig. 7. A translation for terms which should be left 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.

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

(def main ([Term term]) (eval init term (fun (x) x)))
```

Fig. 8. An interpreter for λ-calculus in CPS

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 Fig. 1 yields an interpreter in CPS shown in Fig. 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 Fig. 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 Fig. 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

$$
\begin{aligned}
[\![x]\!] &= \begin{cases} \{\texttt{Prim}_x\} & \text{when } primOp(x) \\ \{\texttt{Top}_x\} & \text{when } topLevel(x) \wedge defun(x) \\ x & \text{otherwise} \end{cases} \\
[\![b]\!] &= b \\
[\![\{r\ x \ldots\}]\!] &= \{\texttt{r}\ [\![x]\!] \ldots\} \\
[\![(\texttt{fun}\ (x \ldots) e)^l]\!] &= \begin{cases} \{\texttt{Fun}_l\ fvs(l)\} & \text{when } defun(l) \\ (\texttt{fun}\ (x \ldots)\ [\![e]\!]) & \text{otherwise} \end{cases} \\
[\![(f^{l'}\ x \ldots)^l]\!] &= \begin{cases} (f\ [\![x]\!] \ldots) & \text{when } primOp(f) \vee topLevel(f) \\ (\texttt{apply}_l\ f\ [\![x]\!] \ldots) & \text{else when } allDefun(l') \\ (f\ [\![x]\!] \ldots) & \text{when } noneDefun(l') \end{cases} \\
[\![(\texttt{match}\ x\ (p\ e) \ldots)]\!] &= (\texttt{match}\ x\ (p\ [\![e]\!]) \ldots) \\
[\![(\texttt{let}\ p\ c\ e)]\!] &= (\texttt{let}\ p\ [\![c]\!]\ [\![e]\!]) \\
[\![(\texttt{error}\ s)]\!] &= (\texttt{error}\ s)
\end{aligned}
$$

Fig. 9. Selective defunctionalization algorithm for *IDL*

$$
\begin{aligned}
mkBranch(x \ldots, & \quad \delta) = (\{\texttt{Prim}_\delta\}\ (\delta\ x \ldots)) \\
mkBranch(x \ldots, & \ (\texttt{def}\ f\ (y \ldots)\ \texttt{e})) = (\{\texttt{Top}_f\}\ (f\ x \ldots)) \\
mkBranch(x \ldots, & \quad (\texttt{fun}\ (y \ldots)\ e)^l) = (\{\texttt{Fun}_l\ fvs(l)\}\ [\![e]\!]\,[y \mapsto x])
\end{aligned}
$$

$$
\begin{aligned}
mkApply(l, fn \ldots) = \ & (\texttt{def}\ \texttt{apply}_l\ (f\ x \ldots) \\
& \ (\texttt{match}\ f \\
& \quad mkBranch(x \ldots, fn) \ldots))
\end{aligned}
$$

Fig. 10. Top-level apply function generation

of Fig. 8 and performing let-inlining yields an encoding of the CEK abstract machine shown in Fig. 11 (again, the environment is left intact).

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 λ-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 Fig. 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 B. The next three interpreters correspond to big-step operational semantics for call-by-name λ-calculus, call-by-need (call-by-name with memoization) and a simple imperative language, respectively.

```
(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

Transformation of call-by-value and call-by-need λ-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 functional correspondence [4,9,11]. 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]).

Language	Interpreter style	Lang. Features	Result
call-by-value λ-calculus	denotational	·	CEK machine
	denotational	integers with add	CEK with add
	denotational, recursion via environment	integers, recursive let-bindings	similar to Reynolds's first-order interpreter
	denotational with conts.	shift and reset	two layers of conts.
	denotational, monadic	exceptions with handlers	explicit stack unwinding
	denotational, CPS		pointer to exception handler
	normalization by evaluation	·	strong CEK machine
call-by-name λ-calculus	big-step	·	Krivine machine
call-by-need λ-calculus	big-step (state passing)	memoization	lazy Krivine machine
simple imperative	big-step (state passing)	conditionals, while, assignment	·
micro-Prolog	CPS	backtracking, cut operator	logic engine

Fig. 12. Summary of tested interpreters

5 Conclusion

In this article we described an algorithm, based on the functional correspondence [4], 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. 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 [7,26,27,30], 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.

Acknowledgements. This article is based on the MSc thesis by the first author, written under the supervision of the second author at the Institute of Computer Science, University of Wrocław.

The project is co-financed by the Polish National Agency for Academic Exchange (PHC Polonium PPN/BFR/2020/1/00001), and by the National Science Centre of Poland, under grant no. 2019/33/B/ST6/00289.

We thank Tomasz Drab and Alan Schmitt for their encouraging interest in this project. We are also grateful to Maciej Piróg for his valuable comments on an earlier version and to the anonymous reviewers for their help in improving the presentation of this work.

A 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):

```
(def factorial (n)
  (match (< 0 n)
    (#t (* n (factorial (- n 1))))
    (#f 1)))

(def main ([Integer n]) (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:

```
(def factorial (n cont)
  (match (< 0 n)
    (#t (factorial (- n 1)
                   (fun (var) (cont (* n var)))))
    (#f (cont 1))))

(def main ([Integer n]) (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 conitnuation, 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 λ-calculus

In this section we present a more involved case study: deriving a previously unknown abstract machine from a normalization function for λ-calculus. The

```
(def-data Term
  {Var Integer}
  {App Term Term}
  {Abs Term})

(def-struct {Level Integer})
(def-struct {Fun Any})

(def cons #:atomic (val env)
  (fun #:atomic #:no-defun (n)
    (match n
      (0 val)
      (_ (env (- n 1))))))

(def reify (ceil val)
  (match val
    ({Fun f}
      {Abs (reify (+ ceil 1) (f {Level ceil}))})
    ({Level k} {Var (- ceil (+ k 1))})
    ({App f arg} {App (reify ceil f) (reify ceil arg)})))

(def apply (f arg)
  (match f
    ({Fun f} (f arg))
    (_ {App f arg})))

(def eval (expr env)
  (match expr
    ({Var n} (env n))
    ({App f arg} (apply (eval f env) (eval arg env)))
    ({Abs body} {Fun (fun #:name Closure (x)
                       (eval body (cons x env)))})))

(def run (term)
  (reify 0
    (eval term (fun #:atomic #:no-defun (x)
                 (error "empty env")))))
```

Fig. 13. A normalization function for call-by-value λ-calculus

normalization function is shown in Fig. 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 λ-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

```
(def reify (ceil val cont)
  (match val
    ({Fun f} (apply1 f {Level ceil} {Fun1 cont (+ ceil 1)}))
    ({Level k} (continue1 cont {Var (- ceil (+ k 1))}))
    ({App f arg} (reify ceil f {App1 arg ceil cont}))))

(def apply (f arg cont1)
  (match f
    ({Fun f} (apply1 f arg cont1))
    (_ (continue cont1 {App f arg}))))

(def eval (expr env cont2)
  (match expr
    ({Var n} (continue cont2 (env n)))
    ({App f arg} (eval f env {App3 arg cont2 env}))
    ({Abs body} (continue cont2 {Fun {Closure body env}}))))

;; evaluation
(def continue (fn2 var13)
  (match fn2
    ({Fun1 cont var3} (reify var3 var13 {Fun2 cont}))
    ({App4 cont2 var12} (apply var12 var13 cont2))
    ({App3 arg cont2 env} (eval arg env {App4 cont2 var13}))
    ({Cont cont4 var16} (reify var16 var13 cont4))))

(def run (term cont4)
  (eval term (fun (x) (error "empty env")) {Cont cont4 0}))

(def apply1 (fn x cont3)
  (match fn ({Closure body env}
             (eval body (cons x env) cont3))))

;; reification
(def continue1 (fn1 var11)
  (match fn1
    ({Fun2 cont} (continue1 cont {Abs var11}))
    ({App2 cont var10} (continue1 cont {App var10 var11}))
```

Fig. 14. A strong call-by-value machine for λ-calculus

which is represented as a function wrapped in a `Fun` record. The values also include `Level`s and `Term`s 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 Fig. 14. 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 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 `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. [3] 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 `semt` is as legible as the one derived manuallly.

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.) LOPSTR 2004. LNCS, vol. 3573, pp. 245–261. Springer, Heidelberg (2005). 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 Rep. Ser. **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.) TACS 2001. LNCS, vol. 2215, pp. 420–447. Springer, Heidelberg (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: Oliveira, B.C.S. (ed.) APLAS 2020. LNCS, vol. 12470, pp. 147–166. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64437-6_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, 3–9 September 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.) LOPSTR 2003. LNCS, vol. 3018, pp. 143–159. Springer, Heidelberg (2004). 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 Rep. Ser. **11**(26) (2004)
16. Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex. MIT Press, Cambridge (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, Ebberup, 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, 23–25 June 1993, pp. 237–247. ACM (1993). https://doi.org/10.1145/155090.155113
19. Friedman, D.P., Wand, M.: Essentials of Programming Languages, 3rd edn. MIT Press, Cambridge (2008)
20. Hannan, J., Miller, D.: From operational semantics to abstract machines. Math. Struct. Comput. Sci. **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, 18 September 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, 27–29 September 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 2013, Madrid, Spain, 16–18 September 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 Rep. Ser. **7** (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, 23–26 May 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, Heidelberg (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.) IFL 2010. LNCS, vol. 6647, pp. 72–88. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24276-2_5