BRICS RS-04-20 M. S. Ager: From Natural Semantics to Abstract Machines

BRICS

Basic Research in Computer Science

From Natural Semantics to Abstract Machines

Mads Sig Ager

BRICS Report Series RS-04-20

ISSN 0909-0878 October 2004

See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/04/20/`

From Natural Semantics to Abstract Machines*

Mads Sig Ager
BRICS†
Department of Computer Science
University of Aarhus‡

October 2004

Abstract

We describe how to construct correct abstract machines from the class of L-attributed natural semantics introduced by Ibraheem and Schmidt at HOOTS 1997. The construction produces stack-based abstract machines where the stack contains evaluation contexts. It is defined directly on the natural semantics rules. We formalize it as an extraction algorithm and we prove that the algorithm produces abstract machines that are equivalent to the original natural semantics. We illustrate the algorithm by extracting abstract machines from natural semantics for call-by-value, call-by-name, and call-by-need evaluation of lambda terms.

*Appears in Etalle, editor, *Pre-proceedings of the International Symposium on Logic-based Program Synthesis and Transformation*, LOPSTR 2004, pages 260-277.
†Basic Research in Computer Science (`www.brics.dk`),
funded by the Danish National Research Foundation.
‡IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark.
Email: `mads@brics.dk`

Contents

1 Introduction **3**

2 From natural semantics to abstract machines **3**
2.1 L-attributed natural semantics 4
2.2 Abstract-machine extraction 5
2.3 Correctness of the extraction 7

3 Applications **11**
3.1 Call-by-value evaluation of λ-terms 12
3.2 Call-by-name evaluation of λ-terms 13
3.3 Call-by-need evaluation of λ-terms 14
3.4 Other applications 16

4 Limitations **16**

5 Related work **17**

6 Conclusion **19**

1 Introduction

Abstract machines have been widely used in the implementation of programming languages [7]. Most of them have been invented from scratch and subsequently been proved to correctly implement the specification of a programming language [11]. Some of them have been derived from the specification of a programming language using some formal system [10, 17]. Most of these derivations use ad hoc derivation steps and are fairly complicated.

In this work we present a simple approach to the construction of correct abstract machines from natural semantics descriptions. At HOOTS 1997 Ibraheem and Schmidt introduced a restricted class of natural semantics called L-attributed natural semantics [12]. The class of L-attributed natural semantics is restricted to have a left-to-right ordering on the premises of each rule ensuring that a proof search using the rules can be performed as left-to-right tree traversals. We observe that for the class of L-attributed natural semantics it is possible to directly extract abstract machines from the natural semantics rules. The extracted machines are stack based and the stack contains evaluation contexts. We formalize this observation as an extraction algorithm and we prove that the algorithm produces abstract machines that are equivalent to the natural semantics.

The class of L-attributed natural semantics is large, containing for instance semantics for pure functional languages, impure functional languages, and imperative languages. The extraction algorithm makes it possible to mechanically extract abstract machines that are correct by construction from these semantics. We illustrate the extraction algorithm by extracting abstract machines from L-attributed natural semantics for call-by-value, call-by-name, and call-by-need evaluation of λ-terms.

The rest of this article is organized as follows. We first define the class of L-attributed natural semantics (Section 2.1). We next define an algorithm for extracting abstract machines from L-attributed natural semantics (Section 2.2) and prove its correctness (Section 2.3). We then consider applications of the extraction (Section 3). Finally, we consider limitations of the approach (Section 4), review related work (Section 5), and conclude (Section 6).

2 From natural semantics to abstract machines

We consider operational semantics for languages consisting of terms. Terms are inductively constructed from atomic terms using term constructors. Other than that, the terms are left unspecified. Values and environments are left unspecified.

- $t \in$ Terms,
- $op \in$ Term constructors,
- $v \in$ Values,

- $\rho \in$ Environments,
- $\sigma \in$ Stacks.

We use the notation $v : \sigma$ for the stack σ with the value v added as the top element. We use subscripting (t_i) and primes (t') to distinguish different occurrences of the meta variables.

2.1 L-attributed natural semantics

In this section we present a restricted class of natural semantics called L-attributed natural semantics. The definition below is essentially identical to the definition of Ibraheem and Schmidt [12]. Similar restrictions on the format of natural semantics rules can be found in Hannan and Miller's work on deriving abstract machines from operational semantics using proof-theoretic methods [10].

Definition 1 (L-attributed natural semantics) *A natural semantics is L-attributed if it consists of rules of the form:*

$$\frac{\rho_1 \vdash t'_1 \Downarrow v_1 \qquad \rho_2 \vdash t'_2 \Downarrow v_2 \qquad \dots \qquad \rho_m \vdash t'_m \Downarrow v_m}{\rho_0 \vdash op(t_1, \dots, t_n) \Downarrow v_{m+1}} \quad (\text{R})$$

where
$$\begin{array}{rcl} \rho_i & = & f_{\text{R}}^{\rho_i}(t_1, \dots, t_n, \rho_0, \dots, \rho_{i-1}, v_1, \dots, v_{i-1}) \\ t'_i & = & f_{\text{R}}^{t_i}(t_1, \dots, t_n, \rho_0, \dots, \rho_{i-1}, v_1, \dots, v_{i-1}) \\ v_{m+1} & = & f_{\text{R}}^{val}(t_1, \dots, t_n, \rho_0, \dots, \rho_m, v_1, \dots, v_m) \end{array}$$

for some partial functions $f_{\text{R}}^{\rho_i}$, $f_{\text{R}}^{t_i}$, and f_{R}^{val} with $1 \leq i \leq m$.

Rules with no premises ($m = 0$) are called axiom rules and rules with at least one premise ($m > 0$) are called non-axiom rules. The number of premises m of a rule is not related to the number of subterms n of the term in the conclusion of the rule as illustrated by the following examples:

- A semantics for a language of boolean-valued terms might contain a negation term constructor $\neg t$ with one subterm. The natural semantics rule for the evaluation of $\neg t$ would have one premise stating that the subterm t evaluates to a boolean b. The value in the conclusion of the rule would then be $\neg b$. In this case the number of premises equals the number of subterms.
- For a language with *if*-expressions the number of premises will be less than the number of subterms in rules for the *if* term constructor *if* t_0 *then* t_1 *else* t_2. There will be one premise for the evaluation of the first subterm t_0 and one premise for the evaluation of either t_1 or t_2.
- A natural semantics for call-by-value evaluation of λ-terms contains a rule for the application term constructor $t_0\, t_1$ with two subterms. This rule

has three premises: the evaluation of t_0 to a function value, the evaluation of t_1 to an argument value, and the evaluation of the application of the function value to the argument value. In this case the number of premises is greater than the number of subterms.

Compared to Kahn's original definition of natural semantics [13], an L-attributed natural semantics is restricted to working on ternary relations relating a term and an environment to a value. The restriction to ternary relations is not a serious restriction: environments, terms, and values are left unspecified, so all three components can have structure. In Kahn's format, each rule has an unordered collection of premises and the rule may have conditions. The L-attributed rules instead have a left-to-right ordering on the premises. This ordering is captured in the definition by ensuring that each of the environments ρ_i, terms t_i, and values v_i can be computed from the previous environments, terms, and values. Furthermore, the rules do not have explicit conditions. Conditions are encoded as part of the functional dependencies $f^{t_i}_{\mathrm{R}}$, $f^{\rho_i}_{\mathrm{R}}$, and f^{val}_{R} between environments, terms, and values. Therefore, the functions giving the dependencies are partial functions and a rule only applies if the dependency functions are defined for the given arguments.

Enforcing a left-to-right ordering on the premises of the rules ensures that if the semantics is deterministic, a proof search using the rules can be performed as a single left-to-right depth-first traversal. Therefore, if the semantics is deterministic, the proof search can be implemented as a recursively defined evaluator in a functional language. For the rest of the development in this article we do not assume that the semantics is deterministic.

2.2 Abstract-machine extraction

We now show how to extract an abstract machine directly from L-attributed natural semantics rules. The abstract machines we consider are state-transition systems operating on three types of states:

1. Triples $(t, \rho, \sigma)_{\mathcal{E}}$ consisting of a term, an environment, and a stack. States of this form correspond to evaluating the term t in the environment ρ and stack σ.

2. Pairs $(\sigma, v)_{\mathcal{A}}$ consisting of a stack and a value. States of this form correspond to 'applying' the stack σ to the value v.

3. Values v representing the final state of a computation.

Before defining the extraction, we introduce a bit of notation. Given an L-attributed natural semantics rule of the form

$$\frac{\rho_1 \vdash t'_1 \Downarrow v_1 \qquad \rho_2 \vdash t'_2 \Downarrow v_2 \qquad \dots \qquad \rho_m \vdash t'_m \Downarrow v_m}{\rho_0 \vdash op(t_1, \dots, t_n) \Downarrow v_{m+1}} \tag{R}$$

$$\begin{array}{rcl} \text{where} \quad \rho_i & = & f_{\mathrm{R}}^{\rho_i}(t_1, \ldots, t_n, \rho_0, \ldots, \rho_{i-1}, v_1, \ldots, v_{i-1}) \\ t'_i & = & f_{\mathrm{R}}^{t_i}(t_1, \ldots, t_n, \rho_0, \ldots, \rho_{i-1}, v_1, \ldots, v_{i-1}) \\ v_{m+1} & = & f_{\mathrm{R}}^{val}(t_1, \ldots, t_n, \rho_0, \ldots, \rho_m, v_1, \ldots, v_m) \end{array}$$

for $1 \le i \le m$, we define the tuples $\mathrm{R}_j[\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \ldots, \overline{\rho_j}, \overline{v_1}, \ldots, \overline{v_{j-1}}]$ for each $1 \le j \le m$. The overlining of terms, environments, and values indicates that the terms, environments, and values are only present in the tuple if they are used by dependency functions $f_{\mathrm{R}}^{\rho_k}$ or $f_{\mathrm{R}}^{t_k}$ for $k > j$ or by f_{R}^{val}. A term, environment, or value is used by a later dependency function if the corresponding variable occurs free in the body of one of these functions. For such a tuple, we define the application $f(\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \ldots, \overline{\rho_j}, \overline{v_1}, \ldots, \overline{v_{j-1}}, v_j)$ to be the application of the function f to the elements that are actually present in the tuple and a value supplying dummy arguments for the elements not present in the tuple. Supplying dummy arguments makes sense since they will not be used—if they were used, there would be a value corresponding to the overlined variable in the tuple.

With these notational conventions in place, we are ready to define the extraction of an abstract machine from an L-attributed natural semantics.

Definition 2 (Extracted abstract machine) *Given an L-attributed natural semantics where each rule has a distinct name, define the extracted abstract machine consisting of the following transition rules:*

1. *An unload rule to terminate the computation:*

$$(\sigma_0, v)_{\mathcal{A}} \rightarrow v$$

where σ_0 is the empty stack.

2. *For each axiom in the L-attributed natural semantics*

$$\rho \vdash op(t_1, \ldots, t_n) \Downarrow v \qquad (\mathrm{R})$$

the rule:

$$(op(t_1, \ldots, t_n), \rho, \sigma)_{\mathcal{E}} \rightarrow (\sigma, v)_{\mathcal{A}}$$

where $v = f_{\mathrm{R}}^{val}(t_1, \ldots, t_n, \rho)$.

3. *For each non-axiom rule in the L-attributed natural semantics*

$$\frac{\rho_1 \vdash t'_1 \Downarrow v_1 \qquad \rho_2 \vdash t'_2 \Downarrow v_2 \qquad \ldots \qquad \rho_m \vdash t'_m \Downarrow v_m}{\rho_0 \vdash op(t_1, \ldots, t_n) \Downarrow v_{m+1}} \qquad (\mathrm{R})$$

$$\begin{array}{rcl} \textit{where} \quad \rho_i & = & f_{\mathrm{R}}^{\rho_i}(t_1, \ldots, t_n, \rho_0, \ldots, \rho_{i-1}, v_1, \ldots, v_{i-1}) \\ t'_i & = & f_{\mathrm{R}}^{t_i}(t_1, \ldots, t_n, \rho_0, \ldots, \rho_{i-1}, v_1, \ldots, v_{i-1}) \\ v_{m+1} & = & f_{\mathrm{R}}^{val}(t_1, \ldots, t_n, \rho_0, \ldots, \rho_m, v_1, \ldots, v_m) \end{array}$$

for $1 \le i \le m$, the rules:

- *Initial evaluation rule:*

$$(op(t_1, \ldots, t_n), \rho_0, \sigma)_{\mathcal{E}} \rightarrow (t'_1, \rho_1, \mathrm{R}_1[\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \overline{\rho_1}] : \sigma)_{\mathcal{E}}$$

where $t'_1 = f_{\mathrm{R}}^{t_1}(t_1, \ldots, t_n, \rho_0)$ *and* $\rho_1 = f_{\mathrm{R}}^{\rho_1}(t_1, \ldots, t_n, \rho_0)$.

- *Stack application rules for* $1 \leq i \leq m-1$*:*

$$\begin{array}{l}(\mathrm{R}_i[\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \ldots, \overline{\rho_i}, \overline{v_1}, \ldots, \overline{v_{i-1}}] : \sigma, v_i)_{\mathcal{A}} \rightarrow \\ (t'_{i+1}, \rho_{i+1}, \mathrm{R}_{i+1}[\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \ldots, \overline{\rho_{i+1}}, \overline{v_1}, \ldots, \overline{v_i}] : \sigma)_{\mathcal{E}}\end{array}$$

$$\begin{array}{llll} \textit{where} \quad t'_{i+1} & = & f_{\mathrm{R}}^{t_{i+1}}(\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \ldots, \overline{\rho_i}, \overline{v_1}, \ldots, \overline{v_{i-1}}, v_i) & \textit{and} \\ \rho_{i+1} & = & f_{\mathrm{R}}^{\rho_{i+1}}(\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \ldots, \overline{\rho_i}, \overline{v_1}, \ldots, \overline{v_{i-1}}, v_i). & \end{array}$$

- *Final stack application rule:*

$$(\mathrm{R}_m[\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \ldots, \overline{\rho_m}, \overline{v_1}, \ldots, \overline{v_{m-1}}] : \sigma, v_m)_{\mathcal{A}} \rightarrow (\sigma, v_{m+1})_{\mathcal{A}}$$

where $v_{m+1} = f_{\mathrm{R}}^{val}(\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \ldots, \overline{\rho_m}, \overline{v_1}, \ldots, \overline{v_{m-1}}, v_m)$.

The stack introduced by the extraction algorithm is a stack of evaluation contexts. The extraction is therefore a new way of constructing evaluation contexts in the style of Felleisen [8]. (Our previous work on deriving abstract machines by continuation-passing style transforming and defunctionalizing evaluators provided another construction of evaluation contexts as defunctionalized continuations [1, 6].)

2.3 Correctness of the extraction

The extraction of Definition 2 is partially correct with respect to the original L-attributed natural semantics. The correctness is partial in the sense that we only consider finite derivations, i.e., convergent computations.

Theorem 1 (Equivalence) *An L-attributed natural semantics and the extracted abstract machine are equivalent. For all term constructors op, terms* $t_1, \ldots, t_n$*, environments* ρ*, and values* v*:*

$$\rho \vdash op(t_1, \ldots, t_n) \Downarrow v \Rightarrow (op(t_1, \ldots, t_n), \rho, \sigma_0)_{\mathcal{E}} \rightarrow^* v$$

and

$$(op(t_1, \ldots, t_n), \rho, \sigma_0)_{\mathcal{E}} \rightarrow^k v \Rightarrow \rho \vdash op(t_1, \ldots, t_n) \Downarrow v$$

for some finite $k > 0$*, where* σ_0 *is the empty stack.*

In order to prove Theorem 1 we prove two lemmas that each imply one part of the equivalence.

Lemma 1 *For all term constructors op, terms* $t_1, \ldots, t_n$*, environments* ρ*, stacks* σ*, and values* v*:*

$$\rho \vdash op(t_1, \ldots, t_n) \Downarrow v \Rightarrow (op(t_1, \ldots, t_n), \rho, \sigma)_{\mathcal{E}} \rightarrow^* (\sigma, v)_{\mathcal{A}}.$$

Proof: By induction on the height of the derivation of

$$\rho \vdash op(t_1, \ldots, t_n) \Downarrow v$$

Assume that the last rule used in the derivation was an axiom of the form

$$\rho \vdash op(t_1, \ldots, t_n) \Downarrow v \qquad (\text{R})$$

then by definition the extracted abstract machine contains the rule

$$(op(t_1, \ldots, t_n), \rho, \sigma)_{\mathcal{E}} \rightarrow (\sigma, v)_{\mathcal{A}}$$

where $v = f_{\text{R}}^{val}(t_1, \ldots, t_n, \rho_0)$ which is what we needed to show.

Assume that the last rule used in the derivation was a non-axiom rule of the form

$$\frac{\rho_1 \vdash t_1' \Downarrow v_1 \qquad \rho_2 \vdash t_2' \Downarrow v_2 \qquad \ldots \qquad \rho_m \vdash t_m' \Downarrow v_m}{\rho_0 \vdash op(t_1, \ldots, t_n) \Downarrow v_{m+1}} \qquad (\text{R})$$

where
$$\begin{array}{rcl} \rho_i & = & f_{\text{R}}^{\rho_i}(t_1, \ldots, t_n, \rho_0, \ldots, \rho_{i-1}, v_1, \ldots, v_{i-1}) \\ t_i' & = & f_{\text{R}}^{t_i}(t_1, \ldots, t_n, \rho_0, \ldots, \rho_{i-1}, v_1, \ldots, v_{i-1}) \\ v_{m+1} & = & f_{\text{R}}^{val}(t_1, \ldots, t_n, \rho_0, \ldots, \rho_m, v_1, \ldots, v_m). \end{array}$$

By inversion we know that each of the premises holds and therefore by the induction hypothesis for all $1 \leq i \leq m$

$$(t_i', \rho_i, \sigma)_{\mathcal{E}} \rightarrow^* (\sigma, v_i)_{\mathcal{A}}$$

for all stacks σ. For each $1 \leq j \leq m$, we prove that

$$(op(t_1, \ldots, t_n), \rho_0, \sigma)_{\mathcal{E}} \rightarrow^* (\text{R}_j[\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \ldots, \overline{\rho_j}, \overline{v_1}, \ldots, \overline{v_{j-1}}] : \sigma, v_j)_{\mathcal{A}}$$

by induction on j.

Base case: $j = 1$ and by definition of the extracted abstract machine there is an initial evaluation rule such that

$$(op(t_1, \ldots, t_n), \rho_0, \sigma)_{\mathcal{E}} \rightarrow (t_1', \rho_1, \text{R}_1[\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \overline{\rho_1}] : \sigma)_{\mathcal{E}}.$$

By the outer induction hypothesis on the premises of the L-attributed natural semantics rule, the following derivation exists:

$$\begin{array}{rl} (op(t_1, \ldots, t_n), \rho_0, \sigma)_{\mathcal{E}} \rightarrow & (t_1', \rho_1, \text{R}_1[\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \overline{\rho_1}] : \sigma)_{\mathcal{E}} \\ \rightarrow^* & (\text{R}_1[\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \overline{\rho_1}] : \sigma, v_1)_{\mathcal{A}}. \end{array}$$

Induction case: $j > 1$. By the induction hypothesis on $j-1$ we can derive

$$(op(t_1, \ldots, t_n), \rho_0, \sigma)_{\mathcal{E}} \rightarrow^* (\mathrm{R}_{j-1}[\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \ldots, \overline{\rho_{j-1}}, \overline{v_1}, \ldots, \overline{v_{j-2}}] : \sigma, v_{j-1})_{\mathcal{A}}.$$

By definition the extracted abstract machine contains the rule

$$\begin{array}{l}(\mathrm{R}_{j-1}[\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \ldots, \overline{\rho_{j-1}}, \overline{v_1}, \ldots, \overline{v_{j-2}}] : \sigma, v_{j-1})_{\mathcal{A}} \rightarrow \\ (t'_j, \rho_j, \mathrm{R}_j[\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \ldots, \overline{\rho_j}, \overline{v_1}, \ldots, \overline{v_{j-1}}] : \sigma)_{\mathcal{E}}.\end{array}$$

By the outer induction hypothesis on the premises of the L-attributed natural semantics rule, the following holds:

$$\begin{array}{l}(t'_j, \rho_j, \mathrm{R}_j[\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \ldots, \overline{\rho_j}, \overline{v_1}, \ldots, \overline{v_{j-1}}] : \sigma)_{\mathcal{E}} \rightarrow^* \\ (\mathrm{R}_j[\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \ldots, \overline{\rho_j}, \overline{v_1}, \ldots, \overline{v_{j-1}}] : \sigma, v_j)_{\mathcal{A}}.\end{array}$$

Putting these parts together finishes the subproof.

By what we have just proved with $j = m$ combined with the final stack application rule of the extracted abstract machine we have the following derivation

$$\begin{array}{lll}(op(t_1, \ldots, t_n), \rho, \sigma)_{\mathcal{E}} & \rightarrow^* & (\mathrm{R}_m[\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \ldots, \overline{\rho_m}, \overline{v_1}, \ldots, \overline{v_{m-1}}] : \sigma, v_m)_{\mathcal{A}} \\ & \rightarrow & (\sigma, v_{m+1})_{\mathcal{A}}\end{array}$$

which concludes the proof. $\square$

Setting $\sigma = \sigma_0$, the empty stack, in Lemma 1 we obtain one direction of Theorem 1.

Lemma 2 *For all term constructors op, terms $t_1, \ldots, t_n$, environments ρ, stacks σ, and values v, if*

$$(op(t_1, \ldots, t_n), \rho, \sigma)_{\mathcal{E}} \rightarrow^k v$$

for a finite $k > 0$ then

$$\rho \vdash op(t_1, \ldots, t_n) \Downarrow v'$$

for some value v' and there exists a prefix of the abstract machine derivation of length $a < k$ such that

$$(op(t_1, \ldots, t_n), \rho, \sigma)_{\mathcal{E}} \rightarrow^a (\sigma, v')_{\mathcal{A}}.$$

Proof: By induction on the length k of the derivation.

Base case: $k = 2$. The minimum length of a derivation of the extracted abstract machine is two steps, and the derivation has the form:

$$(op(t_1, \ldots, t_n), \rho, \sigma_0)_{\mathcal{E}} \rightarrow (\sigma_0, v)_{\mathcal{A}} \rightarrow v$$

where σ_0 is the empty stack. The first step of this derivation is only possible for a rule in the extracted abstract machine that corresponds to an axiom in the L-attributed natural semantics. Since such a rule exists in the extracted abstract machine, the following axiom must be a part of the natural semantics:

$$\rho \vdash op(t_1, \ldots, t_n) \Downarrow v$$

Setting $a = 1$ finishes this case.

Induction case: $k > 2$. Since the number of steps in the abstract-machine derivation is larger than two, the first rule used in the derivation was extracted from a natural semantics rule with $m \geq 1$ premises:

$$\frac{\rho_1 \vdash t'_1 \Downarrow v_1 \qquad \rho_2 \vdash t'_2 \Downarrow v_2 \qquad \ldots \qquad \rho_m \vdash t'_m \Downarrow v_m}{\rho_0 \vdash op(t_1, \ldots, t_n) \Downarrow v_{m+1}} \qquad (\mathrm{R})$$

where
$$\begin{array}{rcl} \rho_i & = & f_{\mathrm{R}}^{\rho_i}(t_1, \ldots, t_n, \rho_0, \ldots, \rho_{i-1}, v_1, \ldots, v_{i-1}) \\ t'_i & = & f_{\mathrm{R}}^{t_i}(t_1, \ldots, t_n, \rho_0, \ldots, \rho_{i-1}, v_1, \ldots, v_{i-1}) \\ v_{m+1} & = & f_{\mathrm{R}}^{val}(t_1, \ldots, t_n, \rho_0, \ldots, \rho_m, v_1, \ldots, v_m). \end{array}$$

We start by proving that for all $1 \leq i \leq m$ there exists a prefix of the abstract machine derivation of length $a_i < k - 1$ such that

$$(op(t_1, \ldots, t_n), \rho, \sigma)_{\mathcal{E}} \to^{a_i} (\mathrm{R}_i[\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \ldots, \overline{\rho_i}, \overline{v_1}, \ldots, \overline{v_{i-1}}] : \sigma, v_i)_{\mathcal{A}}$$

and

$$\rho_i \vdash t'_i \Downarrow v_i$$

for some value v_i. The proof is by induction on i.

Base case: $i = 1$. The derivation of length k has the following form

$$(op(t_1, \ldots, t_n), \rho, \sigma)_{\mathcal{E}} \to (t'_1, \rho_1, \mathrm{R}_1[\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \overline{\rho_1}] : \sigma)_{\mathcal{E}} \to^{k-1} v$$

By the outer induction hypothesis on $k - 1$

$$\rho_1 \vdash t'_1 \Downarrow v_1$$

and there exists a prefix of the abstract machine derivation of length $p < k - 1$ such that

$$(t'_1, \rho_1, \mathrm{R}_1[\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \overline{\rho_1}] : \sigma)_{\mathcal{E}} \to^p (\mathrm{R}_1[\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \overline{\rho_1}] : \sigma, v_1)_{\mathcal{A}}.$$

Since the stack is non-empty, a final state cannot be reached in less than two steps, so we know that $p < k-2$. Letting $a_1 = p+1$ finishes this case.

Inductive case: $i = t + 1$ for some $t \geq 1$. By the induction hypothesis on t

$$(op(t_1, \ldots, t_n), \rho, \sigma)_{\mathcal{E}} \to^{a_t} (\mathrm{R}_t[\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \ldots, \overline{\rho_t}, \overline{v_1}, \ldots, \overline{v_{t-1}}] : \sigma, v_t)_{\mathcal{A}}$$

for some $a_t < k - 1$. By definition, the extracted abstract machine contains the stack application rule

$$\begin{array}{l} (\mathrm{R}_t[\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \ldots, \overline{\rho_t}, \overline{v_1}, \ldots, \overline{v_{t-1}}] : \sigma, v_t)_{\mathcal{A}} \to \\ (t'_{t+1}, \rho_{t+1}, \mathrm{R}_{t+1}[\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \ldots, \overline{\rho_{t+1}}, \overline{v_1}, \ldots, \overline{v_t}] : \sigma)_{\mathcal{E}}. \end{array}$$

We have that

$$(t'_{t+1}, \rho_{t+1}, \mathrm{R}_{t+1}[\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \ldots, \overline{\rho_{t+1}}, \overline{v_1}, \ldots, \overline{v_t}] : \sigma)_{\mathcal{E}} \to^{k-(a_t+1)} v$$

and by the outer induction hypothesis

$$\rho_{t+1} \vdash t'_{t+1} \Downarrow v_{t+1}$$

for some v_{t+1} and there exists a prefix of the abstract machine derivation of length $p < k - (a_t + 1)$ such that

$$\begin{array}{l}(t'_{t+1}, \rho_{t+1}, \mathrm{R}_{t+1}[\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \ldots, \overline{\rho_{t+1}}, \overline{v_1}, \ldots, \overline{v_t}] : \sigma)_{\mathcal{E}} \to^{p} \\ (\mathrm{R}_{t+1}[\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \ldots, \overline{\rho_{t+1}}, \overline{v_1}, \ldots, \overline{v_t}] : \sigma, v_{t+1})_{\mathcal{A}}\end{array}$$

Set $a_{t+1} = a_t + p + 1 < k$. Since the stack is non-empty in the configuration after a_{t+1} steps, a final state cannot be reached in less than two steps. Therefore $a_{t+1} < k - 1$ which finishes the case.

We have just proved that for each $1 \leq i \leq m$, $\rho_i \vdash t'_i \Downarrow v_i$. Therefore, we can build a derivation of $\rho_0 \vdash op(t_1, \ldots, t_n) \Downarrow v$ using the natural semantics rule from which the first step in the abstract-machine derivation was extracted. We have also proved that there exists a prefix of the abstract machine derivation of the form

$$\begin{array}{l}(op(t_1, \ldots, t_n), \rho, \sigma)_{\mathcal{E}} \to^{a_m} \\ (\mathrm{R}_m[\overline{t_1}, \ldots, \overline{t_n}, \overline{\rho_0}, \ldots, \overline{\rho_m}, \overline{v_1}, \ldots, \overline{v_{m-1}}] : \sigma, v_m)_{\mathcal{A}}\end{array}$$

where $a_m < k - 1$. Combining this with the final stack application rule extracted from the natural semantics rule yields the prefix

$$(op(t_1, \ldots, t_n), \rho, \sigma)_{\mathcal{E}} \to^{a_m+1} (\sigma, v)_{\mathcal{A}}$$

of length $a_m + 1 < k$ which finishes the proof.

□

Setting $\sigma = \sigma_0$, the empty stack, in Lemma 2 we obtain the second direction of Theorem 1. Therefore, the proof of Theorem 1 is a straightforward corollary of Lemmas 1 and 2.

3 Applications

In Section 2 we have shown that abstract machines can be extracted directly from L-attributed natural semantics. In this section we illustrate this extraction.

3.1 Call-by-value evaluation of λ-terms

We first consider the following standard natural semantics for call-by-value evaluation of λ-terms. Terms are λ-calculus terms: variables x, abstractions $\lambda x.t$, and applications $t_0\, t_1$. Values are closures $\langle x, t, \rho\rangle$, which are triples containing a variable, a term, and an environment. An environment ρ is a partial function from variables to values.

$$\rho \vdash x \Downarrow \rho(x) \qquad (\textsc{Var})$$

$$\rho \vdash \lambda x.t \Downarrow \langle x, t, \rho\rangle \qquad (\textsc{Lam})$$

$$\frac{\rho \vdash t_0 \Downarrow \langle x, t', \rho'\rangle \qquad \rho \vdash t_1 \Downarrow v' \qquad \rho'[x \mapsto v'] \vdash t' \Downarrow v}{\rho \vdash t_0\, t_1 \Downarrow v} \qquad (\textsc{App})$$

This natural semantics is obviously L-attributed: there is a left-to-right ordering of the premises of each rule, and the dependency of later terms, environments, and values on previous terms, environments and values can be easily specified as functions. Therefore, we can apply the extraction of Section 2.2 to obtain an abstract machine. The resulting abstract machine is as follows:

1. Unload rule:
$$(\sigma_0, v)_{\mathcal{A}} \to v$$

2. Axiom rules:
$$(x, \rho, \sigma)_{\mathcal{E}} \to (\sigma, \rho(x))_{\mathcal{A}}$$
$$(\lambda x.t, \rho, \sigma)_{\mathcal{E}} \to (\sigma, \langle x, t, \rho\rangle)_{\mathcal{A}}$$

3. Non-axiom rules:
$$(t_0\, t_1, \rho, \sigma)_{\mathcal{E}} \to (t_0, \rho, \textsc{App}_1[t_1, \rho] : \sigma)_{\mathcal{E}}$$
$$(\textsc{App}_1[t_1, \rho] : \sigma, v_1)_{\mathcal{A}} \to (t_1, \rho, \textsc{App}_2[v_1] : \sigma)_{\mathcal{E}}$$
$$(\textsc{App}_2[\langle x, t, \rho'\rangle] : \sigma, v_2)_{\mathcal{A}} \to (t, \rho'[x \mapsto v_2], \textsc{App}_3[\,] : \sigma)_{\mathcal{E}}$$
$$(\textsc{App}_3[\,] : \sigma, v)_{\mathcal{A}} \to (\sigma, v)_{\mathcal{A}}$$

We identify this machine as a variant of the CEK machine [9]. The only difference is that the extracted machine pushes an empty evaluation context on the stack in the function application rule. This evaluation context is removed from the stack by the last rule and the value is passed unchanged to the next evaluation context. Our extracted machine is therefore not properly tail-recursive. We are currently extending our extraction to identify when the last evaluation context is empty and the f_{R}^{val} is the 'identity function' that just returns the value of the last premise of a rule. In this case we could avoid adding an evaluation context to the stack and not define the final stack application rule, which would correspond to a tail-call optimization.

3.2 Call-by-name evaluation of λ-terms

The following natural semantics is the standard semantics for call-by-name evaluation of λ-terms. As in Section 3.1, terms are λ-calculus terms: variables x, abstractions $\lambda x.t$, and applications $t_0\, t_1$. Values are closures $\langle x, t, \rho\rangle$ which are triples containing a variable, a term, and an environment. An environment ρ is a partial function from variables to pairs (t, ρ) consisting of a term and an environment.

$$\frac{\rho(x) = (t, \rho') \qquad \rho' \vdash t \Downarrow v}{\rho \vdash x \Downarrow v} \tag{Var}$$

$$\rho \vdash \lambda x.t \Downarrow \langle x, t, \rho\rangle \tag{Lam}$$

$$\frac{\rho \vdash t_0 \Downarrow \langle x, t, \rho'\rangle \qquad \rho'[x \mapsto (t_1, \rho)] \vdash t \Downarrow v}{\rho \vdash t_0\, t_1 \Downarrow v} \tag{App}$$

This natural semantics is L-attributed. It is easy to see that the Lam and App rules fit the format of L-attributed natural semantics, but the Var rule deserves a bit of explanation. The rule has one premise and a condition. Putting it into L-attributed form, we have a rule of the form:

$$\frac{\rho_1 \vdash t_1 \Downarrow v}{\rho_0 \vdash x \Downarrow v} \tag{Var'}$$

The condition $\rho(x) = (t, \rho')$ of the Var rule needs to be captured in the functional dependencies $f^{t_1}_{\text{Var'}}$ and $f^{\rho_1}_{\text{Var'}}$. The following functions capture the condition:

$$f^{t_1}_{\text{Var'}}(x, \rho_0) = \begin{cases} t & \text{if } \rho_0(x) = (t, \rho') \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$f^{\rho_1}_{\text{Var'}}(x, \rho_0) = \begin{cases} \rho' & \text{if } \rho_0(x) = (t, \rho') \\ \text{undefined} & \text{otherwise} \end{cases}$$

If the dependency functions are undefined for some arguments, the condition is not true, and the rule does not apply.

With this explanation, we see that the natural semantics is L-attributed, and we can apply the extraction of Section 2.2 to obtain an abstract machine. The resulting abstract machine is as follows:

1. Unload rule:
$$(\sigma_0, v)_{\mathcal{A}} \rightarrow v$$

2. Axiom rules:
$$(\lambda x.t, \rho, \sigma)_{\mathcal{E}} \rightarrow (\sigma, \langle x, t, \rho\rangle)_{\mathcal{A}}$$

3. Non-axiom rules:

$$(x, \rho, \sigma)_{\mathcal{E}} \to (t, \rho', \textsc{Var}_1[\,] : \sigma)_{\mathcal{E}} \quad \text{if } \rho(x) = (t, \rho')$$
$$(\textsc{Var}_1[\,] : \sigma, v)_{\mathcal{A}} \to (\sigma, v)_{\mathcal{A}}$$
$$(t_0\, t_1, \rho, \sigma)_{\mathcal{E}} \to (t_0, \rho, \textsc{App}_1[t_1, \rho] : \sigma)_{\mathcal{E}}$$
$$(\textsc{App}_1[t, \rho] : \sigma, \langle x, t', \rho' \rangle)_{\mathcal{A}} \to (t', \rho'[x \mapsto (t, \rho)], \textsc{App}_2[\,] : \sigma)_{\mathcal{E}}$$
$$(\textsc{App}_2[\,] : \sigma, v)_{\mathcal{A}} \to (\sigma, v)_{\mathcal{A}}$$

As in the call-by-value case, the machine is not properly tail recursive. Both the $\textsc{Var}_1$ and $\textsc{App}_2$ evaluation contexts are empty, and when given a value they both pass it directly to the next evaluation context on the stack. We are currently extending the extraction algorithm to avoid generating these empty evaluation contexts. Such an extension would correspond to a tail-call optimization.

One might hope to obtain the Krivine machine [5] from the call-by-name semantics. However, the extraction always gives two transition relations: an eval transition relation where the left-hand side of the transitions are triples and an apply transition relation where the left-hand side of the transitions are pairs. The Krivine machine only has one transition relation, so we cannot directly obtain it by the extraction of Section 2.2. It is easy to transform the machine obtained into the Krivine machine, but in its current form the extraction does not give it directly.

3.3 Call-by-need evaluation of λ-terms

Launchbury gave a natural semantics for call-by-need evaluation of λ-terms [14] which Sestoft used as the starting point of his derivation of a lazy abstract machine [17]. Before deriving an abstract machine, Sestoft changed the renaming behaviour of Launchbury's natural semantics. In this section we start from Sestoft's revised version of Launchbury's natural semantics and we apply the extraction algorithm to obtain a lazy abstract machine.

The terms of the natural semantics are so-called normalized λ-terms: variables x, abstractions $\lambda x.t$, applications $t\,x$, and *let*-expressions of the form $let\ x_1 = t_1, \ldots, x_n = t_n\ in\ t$ (we use the abbreviation $let\ \{x_i = t_i\}\ in\ t$ for such *let*-expressions). In normalized λ-terms the argument in an application is restricted to being a variable and non-trivial arguments are bound in *let*-expressions. The bindings in a *let*-expression are mutually recursive.

The natural semantics is substitution based and we write $t[t'/x]$ for the naive simultaneous substitution of t' for all free occurrences of x in t. We let Γ, Δ, and Θ denote stores, which are partial functions from variables to terms, and A denote a set of variables. Following Sestoft, we distinguish between two types of variables: pointers p denoting an element in the store and *let*- or λ-bound variables x. Sestoft's revised version of Launchbury's natural semantics is as follows[1]:

[1]We have slightly reformatted Sestoft's rules by writing $(\Gamma, A) \vdash t \Downarrow (\Delta, w)$ instead of $\Gamma : t \Downarrow_A \Delta : w$. This reformatting is inessential, but it makes it easier to realize that the semantics is L-attributed.

$$\frac{(\Gamma, A \cup \{p\}) \vdash t \Downarrow (\Delta, w)}{(\Gamma[p \mapsto t], A) \vdash p \Downarrow (\Delta[p \mapsto w], w)} \tag{\textsc{Var}}$$

$$(\Gamma, A) \vdash \lambda x.t \Downarrow (\Gamma, \lambda x.t) \tag{\textsc{Lam}}$$

$$\frac{(\Gamma, A) \vdash t \Downarrow (\Delta, \lambda y.t') \qquad (\Delta, A) \vdash t'[p/y] \Downarrow (\Theta, w)}{(\Gamma, A) \vdash t\,p \Downarrow (\Theta, w)} \tag{\textsc{App}}$$

$$\frac{(\Gamma[p_i \mapsto \hat{t_i}], A) \vdash \hat{t} \Downarrow (\Delta, w)}{(\Gamma, A) \vdash \mathit{let}\ \{x_i = t_i\}\ \mathit{in}\ t \Downarrow (\Delta, w)} \tag{\textsc{Let}}$$

In the LET rule, the p_i are fresh variables in the sense that they do not occur in Γ, A, or $\mathit{let}\ \{x_i = t_i\}\ \mathit{in}\ t$. The notation $\hat{t}$ is shorthand for the substitution $t[p_1/x_1, \ldots, p_n/x_n]$.

The restriction to normalized λ-terms together with the above rules for *let*-expressions and variables ensures sharing of argument expressions, i.e., call-by-need evaluation. In normalized terms, non-trivial argument expressions are *let*-bound. The LET rule allocates such arguments in the store and the VAR rule updates the store with the value of the expression ensuring that the argument expression is only evaluated once.

When evaluating variables using the VAR rule, the variable that is currently being evaluated is removed from the store to rule out recursions where the evaluation of a variable requires the value of the variable itself. The variable removed from the store is added again once its value is known. The set of variables A records the variables that are left out of the store at any point in the derivation. In the LET rule the freshness requirement on the variables can be checked locally by inspecting the store and the set of variables currently left out of the store.

The natural semantics is L-attributed. Environments are pairs consisting of a store and a set of variables, terms are normalized λ-terms, and values are pairs consisting of a store and a term. There is a clear left-to-right ordering on the premises as also identified by Sestoft [17, Page 5]:

> "[the rules are] essentially sequential: to build a derivation tree, one must determine the final heap of any left-hand premise before proceeding to any right-hand premise."

Applying the extraction algorithm to the natural semantics yields the following abstract machine:

1. Unload rule:
$$(\sigma_0, v)_{\mathcal{A}} \rightarrow v$$

2. Axiom rules:
$$(\lambda x.t, (\Gamma, A), \sigma)_{\mathcal{E}} \rightarrow (\sigma, (\Gamma, \lambda x.t))_{\mathcal{A}}$$

3. Non-axiom rules:

$$(p, (\Gamma[p \mapsto t], A), \sigma)_{\mathcal{E}} \to (t, (\Gamma, A \cup \{p\}), \text{VAR}_1[p] : \sigma)_{\mathcal{E}}$$

$$(\text{VAR}_1[p] : \sigma, (\Delta, w))_{\mathcal{A}} \to (\sigma, (\Delta[p \mapsto w], w))_{\mathcal{A}}$$

$$(t\, p, (\Gamma, A), \sigma)_{\mathcal{E}} \to (t, (\Gamma, A), \text{APP}_1[p, (\Gamma, A)] : \sigma)_{\mathcal{E}}$$

$$(\text{APP}_1[p, (\Gamma, A)] : \sigma, (\Delta, \lambda y.t))_{\mathcal{A}} \to (t[p/y], (\Delta, A), \text{APP}_2[\,] : \sigma)_{\mathcal{E}}$$

$$(\text{APP}_2[\,] : \sigma, v)_{\mathcal{A}} \to (\sigma, v)_{\mathcal{A}}$$

$$(let\ \{x_i = t_i\}\ in\ t, (\Gamma, A), \sigma)_{\mathcal{E}} \to (\hat{t}, (\Gamma[p_i \mapsto \hat{t_i}], A), \text{LET}_1[\,] : \sigma)_{\mathcal{E}}$$

$$(\text{LET}_1[\,] : \sigma, v)_{\mathcal{A}} \to (\sigma, v)_{\mathcal{A}}$$

The transition rule for *let*-expressions has the same restrictions as the LET rule of the natural semantics: each p_i is a fresh variable in the sense that they do not occur in Γ, A, or $let\ \{x_i = t_i\}\ in\ t$ and the notation $\hat{t}$ is shorthand for the substitution $t[p_1/x_1, \ldots, p_n/x_n]$.

This abstract machine is essentially Sestoft's mark 1 machine. Our machine contains two transition relations and is not properly tail-recursive whereas Sestoft's is a variant of the Krivine machine with only one transition relation. Sestoft notices that when the machine adds a variable to the variable set A, the same variable is also added to the stack. Therefore, the variable set is not needed in the machine since the freshness restriction in the transition rule for *let*-expressions can be checked using the stack instead of the set of variables.

3.4 Other applications

In Sections 3.1, 3.2, and 3.3 we have constructed abstract machines from natural semantics for call-by-value, call-by-name, and call-by-need evaluation of λ-terms. Many natural semantics fit the format of L-attributed natural semantics. For instance, one can give an L-attributed natural semantics for λ-calculus extended with exceptions, state, and combinations of exceptions and state and therefore stack inspection can be specified with an L-attributed natural semantics [2]. Simple imperative languages can also be given L-attributed natural semantics. From each of these natural semantics, the extraction algorithm yields a correct abstract machine.

4 Limitations

The extraction presented in Section 2.2 has three main limitations:

1. The extraction algorithm is restricted to L-attributed natural semantics, which rules out some natural semantics. For instance, the mini-ML natural semantics of Kahn is not L-attributed because of cyclic dependencies used to model recursive bindings [13].

2. If an L-attributed natural semantics contains multiple rules for the same term, the abstract machine resulting from the extraction is non-deterministic.

3. As explained in Sections 3.1 and 3.2, the extraction algorithm does not give properly tail-recursive abstract machines.

We are currently working on extending both the class of L-attributed natural semantics and the extraction algorithm to address these limitations.

Another limitation of the approach is that we only consider partial correctness in the sense that we only consider convergent computations. In order to address the issue of divergent computations we would have to provide a means of reasoning about divergent computations in the framework of natural semantics. Ibraheem and Schmidt considered divergent computations by applying a coinductive interpretation of some of the natural semantics rules [12]. We leave such a generalization for future work.

5 Related work

Defining natural semantics and abstract machines separately and then proving that they coincide is standard. Most semantics textbooks describe both kinds of semantics and show how to relate them [16, 18]. The goal of our work is to mechanize the extraction of abstract machines from natural semantics so that the extracted abstract machines are correct by construction.

In previous work, we have observed that defunctionalized, continuation-passing style evaluators are transition systems, i.e., abstract machines [1, 2, 3, 4, 6]. Starting from an evaluator written in a functional programming language such as ML [15], we (1) transform the evaluator into continuation-passing style to make its flow of control explicit, and (2) defunctionalize the continuations to make them first order, obtaining a stack of evaluation contexts. The result is the ML-encoding of an abstract machine, and the correctness of the abstract machine is a corollary of the correctness of the original evaluator and of the program transformations. The evaluators are direct encodings of natural semantics:

$$\begin{tikzcd}
\begin{array}{c}\text{Natural}\\\text{semantics}\end{array} \arrow[d, "\text{ML-encoding}"'] & & \begin{array}{c}\text{Abstract}\\\text{machine}\end{array} \arrow[d, "\text{ML-encoding}"] \\
\mathit{Eval} \arrow[r] & \mathit{Eval}_{cps} \arrow[r] & \mathit{Eval}_{defun}
\end{tikzcd}$$

The work presented in this article is a different approach to constructing correct abstract machines from natural-semantics descriptions. We extract a correct abstract machine directly from the natural semantics rules:

$$\begin{tikzcd}
\begin{array}{c}\text{Natural}\\\text{semantics}\end{array} \arrow[r, dashed] & \begin{array}{c}\text{Abstract}\\\text{machine}\end{array}
\end{tikzcd}$$

The idea of characterizing the left-to-right processing natural semantics in the form of L-attributed natural semantics is due to Ibraheem and Schmidt [12]. The motivation for their definition came from L-attributed grammars. Ibraheem and Schmidt are concerned with reasoning about divergent computations in the framework of natural semantics. To this end, they start from L-attributed natural semantics and generate sets of positive (or convergent) rules and negative (or divergent) rules. Using an inductive interpretation of the positive rules and a coinductive interpretation of the negative rules allows them to reason about divergent computations. In contrast, we only consider convergent computations, and we extract an abstract machine from an L-attributed natural semantics that is equivalent to the natural semantics.

Hannan and Miller derive abstract machines from natural semantics using proof theory [10]. Their derivation consists in encoding a natural semantics in a proof-theoretic meta-language and then carrying out transformations at the meta-language level. In that sense, the work of Hannan and Miller is closely related to our previous work on deriving abstract machine by using standard program transformations on an encoding of a natural semantics in a functional language. One of Hannan and Miller's derivation steps relies on a left-to-right ordering of the premises of the natural semantics rules. This restriction seems to correspond to our present restriction to L-attributed natural semantics. Hannan and Miller derive abstract machines for call-by-name and call-by-value evaluation of λ-terms. Their starting points, called the $\mathcal{N}_0$ and $\mathcal{V}_0$ proof systems, are L-attributed natural semantics. Both natural semantics are very close to the standard ones presented in Sections 3.1 and 3.2. The difference is that λ-terms are de Bruijn encoded, environments are lists of values, and there are explicit rules for looking up an index in an environment. Applying our extraction to these L-attributes natural semantics yields abstract machines that are very similar to the machines extracted in Sections 3.1 and 3.2.

Kahn introduced natural semantics and presented natural semantics for various aspects of programming languages [13]. For instance, he presented a natural semantics for mini-ML which is almost L-attributed: removing the *letrec* construct from the language, the semantics is L-attributed and we can extract an abstract machine directly. The problem with the *letrec* construct is that there is a cyclic dependency between the value of a term and the environment in which the term is evaluated. The environment in which to evaluate the term can therefore not be defined solely as a function of previous terms, environments, and values and therefore the semantics is not L-attributed.

Sestoft derived a lazy abstract machine from Launchbury's natural semantics for call-by-need evaluation of λ-terms [17]. His derivation consists of a number of intuitive steps and a proof of the correctness of each of the steps. As illustrated in Section 3.3 the extraction algorithm presented in this article applies to the natural semantics and the resulting machine is essentially Sestoft's mark 1 machine. Sestoft obtained the mark 1 machine by introducing a stack of evaluation contexts and subsequently proving the correctness of the resulting machine. Our present work shows that such a stack introduction can be applied to a wide range of natural semantics and proves the correctness of the stack introduction algorithm once and for all instead of relying on a correctness proof for each machine. Sestoft only uses the mark 1 machine as a stepping stone, and goes on to introduce environments (besides the stores already present), closures, and variable indices. He proves the correctness of the resulting machines.

6 Conclusion

We have presented a simple and mechanical extraction of correct abstract machines from the class of L-attributed natural semantics introduced by Ibraheem and Schmidt. We have formalized this extraction as an extraction algorithm and proved its correctness. The class of L-attributed natural semantics is large, containing semantics for call-by-value, call-by-name, and call-by-need functional languages, as well as imperative languages. For each L-attributed natural semantics the extraction algorithm produces a correct abstract machine.

Acknowledgements. To Olivier Danvy for his encouragement, fruitful discussions, and useful comments. Thanks are also due to Neil Jones, Julia Lawall, Jan Midtgaard, and the anonymous reviewers for their useful comments.

References

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Dale Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 8–19. ACM Press, 2003.

[2] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. Technical Report BRICS RS-03-35, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, November 2003. Presented at the 2nd APPSEM II workshop, Talinn, Estonia, April 2004.

[3] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters*, 90(5):223–232, 2004.

[4] Dariusz Biernacki and Olivier Danvy. From interpreter to logic engine by defunctionalization. Technical Report BRICS RS-04-5, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, March 2004. To appear in the proceedings of the 2003 International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2003).

[5] Pierre Crégut. An abstract machine for the normalization of λ-terms. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 333–340. ACM Press, 1990.

[6] Olivier Danvy. A rational deconstruction of Landin's SECD machine. Technical Report BRICS RS-03-33, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, October 2003.

[7] Stephan Diehl, Pieter Hartel, and Peter Sestoft. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16:739–751, 2000.

[8] Matthias Felleisen. *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages.* PhD thesis, Department of Computer Science, Indiana University, Bloomington, Indiana, August 1987.

[9] Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. Unpublished lecture notes. `http://www.ccs.neu.edu/home/matthias/3810-w02/readings.html`, 1989-2001.

[10] John Hannan and Dale Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992.

[11] Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional runtime systems within the lambda-sigma calculus. *Journal of Functional Programming*, 1(1):1–100, January 1993.

[12] Husain Ibraheem and David A. Schmidt. Adapting big-step semantics to small-step style: Coinductive interpretations and "higher-order" derivations. In Andrew Gordon, Andrew Pitts, and Carolyn Talcott, editors, *Electronic Notes in Theoretical Computer Science*, volume 10. Elsevier, 2000.

[13] Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39, Passau, Germany, February 1987. Springer-Verlag.

[14] John Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 144–154. ACM Press, 1993.

[15] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.

[16] Flemming Nielson and Hanne Riis Nielson. *Semantics with Applications.* John Wiley & Sons, 1992.

[17] Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.

[18] Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundation of Computing Series. The MIT Press, 1993.

Recent BRICS Report Series Publications

RS-04-20 Mads Sig Ager. *From Natural Semantics to Abstract Machines*. October 2004. 21 pp. Presented at the *International Symposium on Logic-based Program Synthesis and Transformation*, LOPSTR 2004, Verona, Italy, August 26–28, 2004.

RS-04-19 Bolette Ammitzbøll Madsen and Peter Rossmanith. *Maximum Exact Satisfiability: NP-completeness Proofs and Exact Algorithms*. October 2004. 20 pp.

RS-04-18 Bolette Ammitzbøll Madsen. *An Algorithm for Exact Satisfiability Analysed with the Number of Clauses as Parameter*. September 2004. 4 pp.

RS-04-17 Mayer Goldberg. *Computing Logarithms Digit-by-Digit*. September 2004. 6 pp.

RS-04-16 Karl Krukow and Andrew Twigg. *Distributed Approximation of Fixed-Points in Trust Structures*. September 2004. 25 pp.

RS-04-15 Jesús Fernando Almansa. *Full Abstraction of the UC Framework in the Probabilistic Polynomial-time Calculus ppc*. August 2004.

RS-04-14 Jesper Makholm Byskov. *Maker-Maker and Maker-Breaker Games are PSPACE-Complete*. August 2004. 5 pp.

RS-04-13 Jens Groth and Gorm Salomonsen. *Strong Privacy Protection in Electronic Voting*. July 2004. 12 pp. Preliminary abstract presented at Tjoa and Wagner, editors, *13th International Workshop on Database and Expert Systems Applications*, DEXA '02 Proceedings, 2002, page 436.

RS-04-12 Olivier Danvy and Ulrik P. Schultz. *Lambda-Lifting in Quadratic Time*. June 2004. 34 pp. To appear in *Journal of Functional and Logic Programming*. This report supersedes the earlier BRICS report RS-03-36 which was an extended version of a paper appearing in Hu and Rodríguez-Artalejo, editors, *Sixth International Symposium on Functional and Logic Programming*, FLOPS '02 Proceedings, LNCS 2441, 2002, pages 134–151.

RS-04-11 Vladimiro Sassone and Paweł Sobociński. *Congruences for Contextual Graph-Rewriting*. June 2004. 29 pp.