Deriving Compilers and Virtual Machines for a Multi-Level Language*

Atsushi Igarashi[1] and Masashi Iwaki[2]

[1] Kyoto University, Japan, `igarashi@kuis.kyoto-u.ac.jp`
[2] Hitachi, Ltd., Japan, `masashi.iwaki.ew@hitachi.com`

Abstract. We develop virtual machines and compilers for a multi-level language, which supports multi-stage specialization by composing program fragments with quotation mechanisms. We consider two styles of virtual machines—ones equipped with special instructions for code generation and ones without—and show that the latter kind can deal with, more easily, low-level code generation, which avoids the overhead of (run-time) compilation by manipulating instruction sequences, rather than source-level terms, as data. The virtual machines and accompanying compilers are derived by program transformation, which extends Ager et al.'s derivation of virtual machines from evaluators.

1 Introduction

Multi-level (or multi-stage) languages are designed to support manipulation of program fragments as data and execution of generated code, often by the mechanism of quasi-quotation and eval as in Lisp. Most of those languages are considered extensions of the two-level λ-calculus [1] to an arbitrary number of levels, which has been proposed and studied by Glück and Jørgensen [2].

In the last decade, designs, semantics, and type systems of multi-level languages have been studied fairly extensively by many people [3–11]. On the other hand, implementation issues have been discussed mostly in the context of *two-level* systems [12–15], in which generated code itself does not generate code. As is pointed out by Wickline et al. [4], implementation of two-level languages does not extend straightforwardly to multi-level, especially when one wants a program to generate low-level machine code directly, since there is possible code-size blow-up in generating instructions that themselves generate instructions.

Wickline et al. [4] have addressed this problem by developing an extension of the Categorical Abstract Machine (CAM) [16] with a facility for run-time code generation and a compilation scheme for a multi-level extension of ML called ML$^\Box$. Unfortunately, however, the design of the extended CAM is rather ad-hoc and it is not clear how their technique can be applied to different settings.

* Supported in part by Grant-in-Aid for Scientific Research No. 19300007 from MEXT of Japan.

Our Approach and Contributions. We develop virtual machines (VMs) and compilers for multi-level languages as systematically as possible, by extending Ager et al.'s technique [17, 18] to derive from evaluators, by a sequence of well-known program transformations, abstract machines (which take a source term as an input) or VMs (which take an instruction sequence) with compilers. Although this technique has been shown to be applicable to various evaluation strategies including call-by-value, call-by-name, call-by-need, and even strong reduction [19], application to multi-level languages is new (at least, to our knowledge).

We also identify the following two aspects of compilation schemes and how they appear in the derivation of VMs.

– One aspect is whether a VM generates low-level code or source-level code. It would be desirable that a VM support low-level code generation since the overhead of compilation of the generated code can be reduced.
– The other is whether or not a VM is equipped with instructions *dedicated* for emitting instructions. At first, it may sound counter-intuitive that a VM supports code generation without such instructions. It is, however, possible by introducing two execution modes to a VM: in one mode, an instruction is executed as usual, and in the other, the same instruction emits some code. Correspondingly, a compiler will generate the same instruction for the same source language construct, however deep it appears under quotation. We call this scheme *uniform compilation*, while we call the other scheme, using a dedicated instruction set for code generation, *non-uniform compilation*.

Interestingly, the choice between uniform or non-uniform compilation naturally arises during the derivation process. We also find out that deriving VMs supporting low-level code generation fails when non-uniform compilation is chosen; we discuss why it is difficult from the viewpoint of our derivation scheme.

Our main technical contributions can be summarized as follows:

– Derivation of compilers and VMs for a foundational typed calculus $\lambda^{\bigcirc}$ by Davies [3] for multi-level languages; and
– Identification of the two compilation schemes of uniform and non-uniform compilation, which, in fact, arise naturally during derivation.

Although we omit it from this paper for brevity, we have also succeeded to apply the same derivation scheme to another calculus $\lambda^{\Box}$ [8] of multi-level languages.

The Rest of the Paper. We start with reviewing $\lambda^{\bigcirc}$ in Section 2. Then, we first describe the uniform compilation scheme and a VM that generates low-level code in Section 3 and then the non-uniform compilation, which fails at low-level code generation, in Section 4. After discussing related work in Section 5, we conclude in Section 6. The concrete OCaml code of the derivation is available at `http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/papers/VMcircle.html`.

2 $\lambda^{\bigcirc}$

$\lambda^{\bigcirc}$ [3] is a typed λ-calculus, which corresponds to linear-time temporal logic with the temporal operator $\bigcirc$ ("next") by the Curry-Howard isomorphism. A

$\lambda^{\bigcirc}$-term is considered a multi-level generating extension, which repeatedly takes part of the input of a program and yields a residual program, which is a specialized generating extension; and its type system can be considered that for a multi-level binding-time analysis [2, 3]. The term syntax includes **next** and **prev**, which roughly correspond to backquote and unquote in Lisp, respectively. So, in addition to the usual β-reduction, $\lambda^{\bigcirc}$ has reduction to cancel **next** by **prev**: $\textbf{prev}(\textbf{next}\, t) \longrightarrow t$. Unlike Lisp, however, all variables are statically bound and substitution is capture-avoiding, or "hygienic" [20]. For example, the term $(\lambda x.\, \textbf{next}(\lambda y.\, \textbf{prev}\, x))\, (\textbf{next}\, y)$ reduces to $\textbf{next}(\lambda z.\, y)$ in two steps—notice that the bound variable y has been renamed to a fresh one to avoid variable capture. It is a common practice to generate fresh names in implementations where variables have named representation. In this paper, we adopt de Bruijn indices to represent variable binding with a low-level, nameless implementation in mind. So, index shifting will be used to avoid variable capture, instead of renaming bound variables.

2.1 Syntax and Operational Semantics

We first give the syntax and a big-step semantics of a variant of $\lambda^{\bigcirc}$, in which variables are represented by de Bruijn indices. The definitions of terms t, values v, and environments E, are given by the following grammar:

$$t ::= n \mid \lambda t \mid t_0\, t_1 \mid \textbf{next}\, t \mid \textbf{prev}\, t \qquad v ::= \langle E, t\rangle \mid \ulcorner t \urcorner \qquad E ::= \cdot \mid v :: E$$

The *level* of a (sub)term is the number of **next**s minus the number of **prev**s to reach the (sub)term. A variable n refers to the n-th λ-binder *at the same level*. For example, $\lambda y.\, \textbf{next}(\lambda x.\, x(\textbf{prev}\, y))$ will be represented by $\lambda\, \textbf{next}(\lambda 0(\textbf{prev}\, 0))$, not $\lambda\, \textbf{next}(\lambda 0(\textbf{prev}\, 1))$, since x appears at level 1 but y at level 0. This indexing scheme is required because an environment is a list of bindings of level-0 variables and variables at higher levels are treated like constants—so, in order for indices to correctly work, binders at higher levels have to be ignored in computing indices. A value is either a function closure $\langle E, t\rangle$ or a quotation $\ulcorner t \urcorner$[3]. An environment E is a list of values. We focus on a minimal set of language features in this paper but our derivation works when recursion or integers are added.

These definitions can be easily represented by datatype definitions in OCaml, which we use as a meta language in this paper.

```
type term = Var of int | Abs of term | App of term * term
          | Next of term | Prev of term
type value = Clos of env * term | Quot of term  and env = value list
```

As we have mentioned evaluation in $\lambda^{\bigcirc}$ can go under λ-binders. To deal with it, we need "shift" operations to adjust indices. The expression $t \uparrow^{\ell}_{j}$ denotes a term obtained by incrementing the indices of free level-ℓ variables by 1. The

[3] In Davies [3], **next** is used for $\ulcorner\cdot\urcorner$. Our intention here is to distinguish an *operator* for quotation and the result of applying it. Also, we do not stratify values by levels as in [3] since it is not really necessary—the type system does the stratification.

auxiliary argument j counts the number of λ-binders encountered, in order to avoid incrementing the indices of bound variables.

$$n \uparrow_j^\ell = \begin{cases} n+1 & (\text{if } n \geq j \text{ and } \ell = 0) \\ n & (\text{otherwise}) \end{cases} \qquad (t_0\, t_1) \uparrow_j^\ell = (t_0 \uparrow_j^\ell)\,(t_1 \uparrow_j^\ell)$$

$$(\mathbf{next}\, t) \uparrow_j^\ell = \mathbf{next}(t \uparrow_j^{\ell-1})$$

$$(\lambda t) \uparrow_j^\ell = \lambda(t \uparrow_{j+1}^\ell) \qquad (\mathbf{prev}\, t) \uparrow_j^\ell = \mathbf{prev}(t \uparrow_j^{\ell+1})$$

Notice that ℓ is adjusted when **next** or **prev** is encountered. Shifting $E \uparrow^\ell$ of environments is defined as a pointwise extension of term shifting; we omit the definition. We implement these functions as `shift` and `shiftE`, respectively, whose straightforward definitions are also omitted.

Now, we define the call-by-value, big-step operational semantics of $\lambda^\bigcirc$ with the judgment $E \vdash t \Downarrow^\ell r$ where r is either a value v (when $\ell = 0$) or a term t' (otherwise), read "level-ℓ term t evaluates to r under environment E". The inference rules for this judgment are given in Fig. 1, in which $E(n)$ stands for the n-th element of E. As usual, bottom-up reading gives how to evaluate an expression, given an environment and a level. The rules for the case $\ell = 0$ are straightforward extensions of those for the λ-calculus. The rules Eq-— mean that, when $\ell \geq 1$ (i.e., the term is under **next**), the result of evaluation is *almost* the input term; only subterms inside **prev** at level 1 is evaluated, as is shown in E-Prev, in which the quotation of the value is canceled. To avoid variable capture, indices of quoted terms in the environment have to be shifted (by $E \uparrow^\ell$), when evaluation goes under λ-bindings (Eq-Abs).[4] Fig. 2 shows the derivation for the evaluation of $\mathbf{next}(\lambda\, \mathbf{prev}(\lambda\, \mathbf{next}(\lambda\, \mathbf{prev}\, 0))\, (\mathbf{next}\, 0))$, which could be written `` `(lambda (x) ,((lambda (y) `(lambda (z) ,y)) `x)) `` in Scheme.

The type system, which we omit mainly for brevity, guarantees the absence of type errors and that a term of a quotation type evaluates to a quoted term $\ulcorner t \urcorner$, where t is well typed at level 0 and does not contain subterms at a negative level. Our evaluator simply discards type information and types do not play important roles in our development. We assume every term is well typed.

2.2 Environment-Passing, Continuation-Passing Evaluator for $\lambda^\bigcirc$

Once an operational semantics is defined, it is a straightforward task to write an environment-passing, continuation-passing evaluator. It takes not only a term, an environment, and a continuation, but also a level of the input term; hence, the evaluator has type `term * int * env * (value -> value) -> value` (the return type of continuations is fixed to `value`).

```
type cont = value -> value
(* eval0 : term * int * env * cont -> value *)
let rec eval0 (t, l, e, k) = match t, l with
  Var n, 0 -> k (List.nth e n)
```

[4] In the implementation below, shifting is applied to values in an environment eagerly, but it can be delayed to reduce overhead, until the values are referred to by a corresponding variable.

$$\boxed{E \vdash t \Downarrow^0 v}$$

$$\frac{(E(n) = v)}{E \vdash n \Downarrow^0 v}\ (\textsc{E-Var}) \qquad \frac{}{E \vdash n \Downarrow^\ell n}\ (\textsc{Eq-Var})$$

$$\frac{}{E \vdash \lambda t \Downarrow^0 \langle E, t\rangle}\ (\textsc{E-Abs}) \qquad \frac{E \uparrow^\ell \vdash t \Downarrow^\ell t'}{E \vdash \lambda t \Downarrow^\ell \lambda t'}\ (\textsc{Eq-Abs})$$

$$\frac{E \vdash t_0 \Downarrow^0 \langle E', t\rangle \qquad E \vdash t_1 \Downarrow^0 v \qquad v :: E' \vdash t \Downarrow^0 v'}{E \vdash t_0\, t_1 \Downarrow^0 v'}\ (\textsc{E-App}) \qquad \frac{E \vdash t_0 \Downarrow^\ell t_0' \qquad E \vdash t_1 \Downarrow^\ell t_1'}{E \vdash t_0\, t_1 \Downarrow^\ell t_0'\, t_1'}\ (\textsc{Eq-App})$$

$$\frac{E \vdash t \Downarrow^1 t'}{E \vdash \mathbf{next}\, t \Downarrow^0 \ulcorner t' \urcorner}\ (\textsc{E-Next}) \qquad \frac{E \vdash t \Downarrow^{\ell+1} t'}{E \vdash \mathbf{next}\, t \Downarrow^\ell \mathbf{next}\, t'}\ (\textsc{Eq-Next})$$

$$\boxed{E \vdash t \Downarrow^\ell t' \quad (\ell \geq 1)}$$

$$\frac{E \vdash t \Downarrow^0 \ulcorner t' \urcorner}{E \vdash \mathbf{prev}\, t \Downarrow^1 t'}\ (\textsc{E-Prev}) \qquad \frac{E \vdash t \Downarrow^\ell t'}{E \vdash \mathbf{prev}\, t \Downarrow^{\ell+1} \mathbf{prev}\, t'}\ (\textsc{Eq-Prev})$$

Fig. 1. The operational semantics of $\lambda^{\bigcirc}$.

$$\mathcal{D} \equiv \dfrac{\dfrac{\dfrac{\dfrac{}{\ulcorner 1 \urcorner :: \cdot \vdash 0 \Downarrow^0 \ulcorner 1 \urcorner}\,\textsc{E-Var}}{\ulcorner 1 \urcorner :: \cdot \vdash \mathbf{prev}\, 0 \Downarrow^1 1}\,\textsc{E-Prev}}{\ulcorner 0 \urcorner :: \cdot \vdash \lambda\, \mathbf{prev}\, 0 \Downarrow^1 \lambda 1}\,\textsc{Eq-Abs}}{\ulcorner 0 \urcorner :: \cdot \vdash \mathbf{next}(\lambda\, \mathbf{prev}\, 0) \Downarrow^0 \ulcorner \lambda 1 \urcorner}\,\textsc{E-Next}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot \vdash \lambda\, \mathbf{next}(\lambda\, \mathbf{prev}\, 0) \Downarrow^0 \langle \cdot, \mathbf{next}(\lambda\, \mathbf{prev}\, 0)\rangle}\,\textsc{E-Abs} \quad \dfrac{\dfrac{}{\cdot \vdash 0 \Downarrow^1 0}\,\textsc{Eq-Var}}{\cdot \vdash \mathbf{next}\, 0 \Downarrow^0 \ulcorner 0 \urcorner}\,\textsc{E-Next} \quad \begin{matrix}\vdots \\ \mathcal{D}\end{matrix}}{\cdot \vdash (\lambda\, \mathbf{next}(\lambda\, \mathbf{prev}\, 0))\ \mathbf{next}\, 0 \Downarrow^0 \ulcorner \lambda 1 \urcorner}\,\textsc{E-App}}{\cdot \vdash \mathbf{prev}((\lambda\, \mathbf{next}(\lambda\, \mathbf{prev}\, 0))\ \mathbf{next}\, 0) \Downarrow^1 \lambda 1}\,\textsc{E-Prev}}{\cdot \vdash \lambda\, \mathbf{prev}((\lambda\, \mathbf{next}(\lambda\, \mathbf{prev}\, 0))\ \mathbf{next}\, 0) \Downarrow^1 \lambda\lambda 1}\,\textsc{Eq-Abs}}{\cdot \vdash \mathbf{next}(\lambda\, \mathbf{prev}((\lambda\, \mathbf{next}(\lambda\, \mathbf{prev}\, 0))\ \mathbf{next}\, 0)) \Downarrow^0 \ulcorner \lambda\lambda 1 \urcorner}\,\textsc{E-Next}$$

Fig. 2. The derivation of $\cdot \vdash \mathbf{next}(\lambda\, \mathbf{prev}((\lambda\, \mathbf{next}(\lambda\, \mathbf{prev}\, 0))\ \mathbf{next}\, 0)) \Downarrow^0 \ulcorner \lambda\lambda 1 \urcorner$.

```
| Abs t0, 0 -> k (Clos (e, t0))
| App(t0, t1), 0 -> eval0 (t0, 0, e, fun1 (Clos(e',t')) ->
                      eval0 (t1, 0, e, fun2 v -> eval0 (t', v::e', k)))
| Next t0, 0 -> eval0 (t0, 1, e, fun3 (Quot t) -> k (Quot t))
| Prev t0, 1 -> eval0 (t0, 0, e, fun4 (Quot t) -> k (Quot t))
| Var n, l -> k (Quot (Var n))
| Abs t0, l -> eval0 (t0, l, shiftE (e, l), fun5 (Quot t) ->
                 k (Quot (Abs t)))
| App(t0, t1), l -> eval0 (t0, l, e, fun6 (Quot t2) ->
                      eval0 (t1, l, e, fun7 (Quot t3) ->
                       k (Quot (App(t2, t3)))))
| Next t0, l -> eval0 (t0, l+1, e, fun8 (Quot t) -> k (Quot (Next t)))
| Prev t0, l -> eval0 (t0, l-1, e, fun9 (Quot t) -> k (Quot (Prev t)))
(* main0 : term -> value *)
let main0 t = eval0 (t, 0, [], fun0 v -> v)
```

Underlines with subscripts are not part of the program—they will be used to identify function abstractions in the next section. We use a constructor `Quot` of `value` to represent both quoted values $\ulcorner t \urcorner$ and terms returned when `l > 0`. So, the continuations in the fourth and fifth branches (corresponding to E-Next and E-Prev) are (essentially) the identity function (except for checking the constructor). Note that, in the last five branches, which correspond to the rules Eq-—, a term is constructed by using the same constructor as the input.

3 Deriving a Uniform Compiler and VM with Low-Level Code Generation

We first give a very brief review of Ager et al.'s functional derivation of a compiler and a VM [17, 18]. A derivation from a continuation-passing evaluator consists of the following steps:

1. defunctionalization [21] to represent continuations by first-order data;
2. currying transformation to split compile- and run-time computation; and
3. defunctionalization to represent run-time computation by first-order data.

The first step makes a tail-recursive, first-order evaluator, which can be viewed as an abstract machine.[5] The succeeding steps decompose the abstract machine into two functions: the first function that takes a λ-term and generates an intermediate datum is a compiler and the second function that interprets intermediate data is a VM—the intermediate data, obtained by the the third step of defunctionalization, are VM instructions.

We will follow these steps mostly but claim, however, that it is not just an exercise. We will see an interesting issue of the distinction between uniform and non-uniform compilation naturally arises from how the abstract machine can be curried. Also, a VM with low-level code generation cannot be obtained solely

[5] According to Ager et al.'s terminology, an abstract machine takes a λ-term as an input whereas a VM takes an instruction sequence obtained by compiling a term.

by following this scheme: since these derivation steps preserve the behavior of the original evaluator, the resulting VM would yield quoted *source terms* even when VM instructions are introduced. So, we have to devise an additional step to derive a new VM for low-level code generation.

The following commuting diagram illustrates our derivation scheme:

$$\begin{tikzcd}
\text{terms:} & t_0 \arrow[rr, "AM"] \arrow[dr, squiggly, "compile"'] & & \ulcorner t_1 \urcorner \arrow[rr] \arrow[dr, squiggly] & & \ulcorner t_2 \urcorner \arrow[rr] \arrow[dr, squiggly] & & {} \\
\text{instructions:} & & I_0 \arrow[ur, Rightarrow, "VM_H"] \arrow[rr, Rightarrow, "VM_L"] & & I_1 \arrow[ur, Rightarrow] \arrow[rr, Rightarrow] & & I_2 \arrow[ur, Rightarrow] \arrow[r, equal] & {}
\end{tikzcd}$$

The solid arrows on top represent executions of an abstract machine, which is extensionally equal to the initial evaluator; since $\lambda^{\bigcirc}$-terms are multi-level generating extensions, a residual program t_1 (possibly with further inputs) obtained by executing t_0 will be executed again. We decompose $\longrightarrow$ into a compiler $\rightsquigarrow$ and a VM $\overset{VM_H}{\Longrightarrow}$; and then derive a VM $\overset{VM_L}{\Longrightarrow}$ with low-level code generation, which commutes with $\overset{VM_H}{\Longrightarrow}$ followed by compilation. So, once t_0 is compiled, the run-time system (that is, $\overset{VM_L}{\Longrightarrow}$) can forget about source-level terms.

The following subsections describe each step of the derivation in detail.

3.1 Defunctionalizing Continuations

The first step is defunctionalization of continuations. The basic idea of defunctionalization [21] is to represent functional values by datatype constructors and to replace function applications by calls to an “apply” function. This function executes the function body corresponding to the given constructor, which also carries the value of free variables in the original function abstraction. In the definition of the evaluator in the last section, there are ten function abstractions of type `value -> value`: one in `main0` and nine in `eval0`. So, the datatype `cont` is given ten constructors.

The resulting code is as follows (throughout the paper, shaded part represents main changes from the previous version):

```
type cont = Cont0 | Cont1 of term * env * cont | ... | Cont9 of cont
(* eval1 : term * int * env * cont -> value *)
let rec eval1 (t, l, e, k) = match t, l with
  Var n, 0 -> appK1 (k, List.nth e n )
| App (t0, t1), 0 -> eval1 (t0, 0, e, Cont1 (t1, e, k) )
| Var n, l -> appK1 (k, Quot (Var n) )
| App (t0, t1), l -> eval1 (t0, l, e, Cont6 (t1, l, e, k) )  ...
(* appK1 : cont * value -> value *)
and appK1 (k, v) = match k, v with
  Cont0, v -> v
| Cont1 (t1, e, k), v -> eval1 (t1, e, Cont2 (v, k))
| Cont2 (Clos (e', t'), k), v -> eval1 (t', v::e', k)
| Cont5 k, Quot t -> appK1 (k, Quot (Abs t))
| Cont6 (t1, l, e, k), Quot t2 -> eval1 (t1, l, e, Cont7 (t2, k))
| Cont7 (t2, k), Quot t3 -> appK1 (k, Quot (App (t2, t3)))
...
(* main1 : term -> value *)
let main1 t = eval1 (t, 0, [], Cont0)
```

The occurrences of $\underline{\texttt{fun}}_i$ have been replaced with constructors Cont$i$, applied to free variables in the function body. The bodies of those functions are moved to branches of the apply function appK1. For example, the initial continuation is represented by Cont0 (without arguments) and the corresponding branch in appK1 just returns the input v.

The derived evaluator can be viewed as a CEK-style abstract machine [22] for $\lambda^{\bigcirc}$. Indeed, for the pure λ-calculus fragment, this evaluator behaves exactly like the CEK-machine [18].

3.2 Currying and Primitive Recursive Evaluator

Now, we decompose eval1 above into two functions for compilation and execution. For this purpose, we first curry eval1 so that it takes compile-time entities such as terms as arguments and returns a "run-time computation," i.e., a function, which takes run-time entities such as environments and continuations as arguments and returns a value. Also, the evaluator is transformed into a primitive recursive form in such a way that closures carry run-time computation, instead of terms. This transformation removes the dependency of run-time entities on compile-time entities.

Actually, at this point, we have two choices about how it is curried: one choice is to curry to `term * int -> env * cont -> value` and the other is to `term -> int * env * cont -> value`. The former choice amounts to regarding a level as compile-time information, so the resulting compiler can generate different instructions from the same term, depending on its levels; it leads to non-uniform compilation, which will be discussed in Section 4. In this section, we proceed with the latter choice, in which the resulting compiler will depend only on the input term, so it necessarily generates the same instruction from the same term, regardless of its levels.

The currying transformation yields the following code:

```
type value = Clos of env * compt | Quot of term  and env = ...
and compt = int * env * cont -> value
and cont = Cont0 | Cont1 of compt * env * cont | ...
| Cont6 of compt * int * env * cont | Cont7 of term * cont | ...
(* appK2 : cont * value -> value *)
let rec appK2 (k, v) = match k, v with
  Cont0, v -> v
| Cont1 (c1, e, k), v -> c1 ( 0, e, Cont2 (v, k) )
| Cont6 (c1, l, e, k), Quot t2 -> c1 ( l, e, Cont7 (t2, k) )
| Cont7 (t2, k), Quot t3 -> appK2 (k, Quot (App (t2, t3)))
...
(* eval2 : term -> compt *)
let rec eval2 t = match t with
  Var n -> (fun0 (l, e, k) -> if l = 0 then appK2 (k, List.nth e n)
                                     else appK2 (k, Quot (Var n)))
| Abs t0 -> let c0 = eval t0 in
     (fun1 (l, e, k) -> if l = 0 then appK2 (k, Clos (e, c0))
                                 else c0 (l, shiftE (e, l), Cont5 k))
| App(t0,t1) -> let c0 = eval2 t0 and c1 = eval2 t1 in
     (fun2 (l, e, k) -> if l = 0 then c0 (0, e, Cont1 (c1, e, k))
                                 else c0 (l, e, Cont6 (c1, l, e, k))) ...
(* main2 : term -> value *)
let main2 t = eval2 t (0, [], Cont0)
```

Case branching in `eval2` is now in two steps and the second branching on levels is under function abstractions, which represent run-time computation. Some occurrences of `term` in `cont` have been replaced with `compt`, but arguments to `Cont7` (as well as `Quot`) remains the same because it records the *result* of evaluation of the function part of an application at a level greater than 0.

Note that the definitions of `value`, `env` and `cont` are now independent of that of `term`, indicating the separation of compile- and run-time. Also, unlike the previous version, functions `appK2` and `eval2` are not mutually recursive. The function `eval2` becomes primitive recursive and also higher-order (it returns a functional value); we get rid of `fun`s by another defunctionalization.

3.3 Defunctionalizing Run-Time Computation

The next step is to make `compt` first-order data by applying defunctionalization. Here, the datatype for `compt` will be represented by using lists:

```
type compt = inst list
and inst = Compt0 of int | Compt1 of compt | Compt2 of compt | ...
```

rather than

```
type compt = Compt0' of int | Compt1' of compt
           | Compt2' of compt * compt | ...
```

which would be obtained by straightforward defunctionalization. In fact, the latter can be embedded into the former—`Compt0' n` and `Compt2'(c0,c1)` are represented by `[Compt0 n]` and `[Compt2 c1; c0]`, respectively. This scheme allows defunctionalized run-time computation to be represented by a linear data structure, that is, a sequence of instructions. Indeed, as its name suggests, `inst` can be viewed as machine instructions. The resulting evaluator `eval3`, which generates a value of type `compt` from a term, is a compiler; a new apply function `appC3`, which interprets `compt`, together with `appK3` is a VM. In the following code, constructors of `inst` are given mnemonic names.

```
type value = ... and env = ... and cont = ... and compt = inst list
and inst = Access of int | Close of compt | Push of compt | Enter | Leave
(* eval3 : term -> compt *)
let rec eval3 t = match t with
  Var n -> [Access n]  | Abs t0 -> [Close (eval3 t0)]
| App (t0, t1) -> Push (eval3 t1) :: (eval3 t0)
| Next t0 -> Enter :: eval3 t0  | Prev t0 -> Leave :: eval3 t0
(* appK3 : cont * value -> value *)
let rec appK3 (k, v) = match k, v with ...
| Cont1 (c1, e, k), v -> appC3 (c1, 0, e, Cont2 (v,k) )
| Cont6 (c1, l, e, k), Quot t2 -> appC3 (c1, l, e, Cont7(t2, k) )
...
(* appC3 : compt * int * env * cont -> value *)
and appC3 (c, l, e, k) = match c, l with
  [Access n], 0 -> appK3 (k, List.nth e n)
| [Access n], l -> appK3 (k, Quot (Var n))
| [Close c0], 0 -> appK3 (k, Clos (e, c0))
| [Close c0], l -> appC3 (c0, l, shiftE (e, l), Cont5 k)
| Push c1::c0, 0 -> appC3 (c0, 0, e, Cont1 (c1, e, k))
| Push c1::c0, l -> appC3 (c0, l, e, Cont6 (c1, l, e, k))
...
(* main3 : term -> value *)
let main3 t = appC3 (eval3 t, 0, [], Cont0)
```

The compiler `eval3` is uniform since it generates the same instruction regardless of the levels of subterms and the VM interprets the same instruction differently, according to the level.

3.4 Virtual Machine for Low-Level Code Generation

Code generation in the VM derived above is still high-level: as shown in the branch for `Cont7` of `appK3` (or `appK2`), terms, not instructions, are generated during execution. The final step is to derive a VM that generates instructions. This is, in fact, rather easy—everywhere a term constructor appears, we apply the compiler by hand (but leave variables unchanged): for example, the branch

```
  | [Access n], l -> appK3 (k, Quot (Var n))
```

in `appC3` becomes

```
  | [Access n], l -> appK3 (k, Quot [Access n ]))
```

Other changes include replacement of type `term` with `compt` in `value` or `cont` and new definitions to shift indices in an instruction list.

Here is the final code:

```
type value = ... | Quot of compt  and compt = ...  and inst = ...
```

```
and cont = ... | Cont7 of compt * cont | ...  and env = ...

let rec shift_inst (i, l, j) = ... and shift_compt (c, l, j) = ...
let rec shiftE (e, l) = ...

(* eval4 : term -> compt *)
let rec eval4 t = ... (* the same as eval3 *)

(* appK4 : cont * value -> value *)
and appK4 (k, v) = match k, v with ...
| Cont7 (c2, k), Quot c3 -> appK4 (k, Quot (Push c3::c2))
...
(* appC4 : compt * int * env * cont -> value *)
and appC4 (c, l, e, k) = match (c, l) with
| [Access n], l -> appK4 (k, Quot [Access n])
...
(* main4 : term -> value *)
let main4 t = appC4 (eval4 t, 0, [], Cont0)
```

The definitions of code blocks c, instructions I, continuations k, the compiler $[\![t]\!]$, and the transition $\Longrightarrow$ of the VM states are summarized in Fig. 3 (in which the names of continuation constructors are also renamed). An (intermediate) state is of the form $\langle c, \ell, E, k\rangle$ (corresponding to an input to `appC4`), $\langle k, v\rangle$, or $\langle k, c\rangle$ (corresponding to an input to `appK4`). A VM instruction is executed differently according to ℓ. For example, **close**(c) creates a function closure and passes it to the current continuation when $\ell = 0$, whereas the same instruction generates code to build a closure when $\ell > 0$, by first pushing `QAbs` onto the continuation stack and executing the body—when this execution finished, the VM reaches the state $\langle$`QAbs`$(k), c\rangle$, in which c is the generated function body; finally the VM returns an instruction to build a closure (that is, **close**).

4 Non-Uniform Compilation and Failure of Low-level Code Generation

In this section, we briefly describe the derivation of a non-uniform compiler with a VM and see how and why low-level code generation fails. As we already mentioned, currying the evaluation function as `term * int -> env * cont -> value`, by regarding levels as compile-time information, leads us to a non-uniform compiler, which generates special instructions for code generation if the given level is greater than 0. We skip the intermediate steps and show only the resulting non-uniform compiler and VM for high-level code generation, obtained after defunctionalizing `compt`, which is first defined to be `env * cont -> value`.

```
type value = ... and env = ... and cont = ... and compt = inst list
and inst = Access of int | ... | Leave | QVar of int | PushQAbs of int
          | PushQApp of compt | PushQNext | PushQPrev
```

Instructions, values, and continuations:

$$\begin{aligned}
I &::= \textbf{access}\ n \mid \textbf{close}(c) \mid \textbf{push}(c) \mid \textbf{enter} \mid \textbf{leave}\\
c &::= I_0; \cdots ; I_n\\
v &::= \langle E, c\rangle \mid \ulcorner c \urcorner\\
k &::= \texttt{Halt} \mid \texttt{EvArg}(c, E, k) \mid \texttt{EvBody}(v, k) \mid \texttt{Quote}(k) \mid \texttt{Unquote}(k)\\
&\quad\mid\ \texttt{QAbs}(k) \mid \texttt{QApp'}(c, \ell, E, k) \mid \texttt{QApp}(c, k) \mid \texttt{QNext}(k) \mid \texttt{QPrev}(k)
\end{aligned}$$

Compilation:

$$\begin{aligned}
[\![n]\!] &= \textbf{access}\ n & [\![\textsf{\textbf{next}}\ t]\!] &= \textbf{enter}; [\![t]\!]\\
[\![\lambda t]\!] &= \textbf{close}([\![t]\!]) & [\![\textsf{\textbf{prev}}\ t]\!] &= \textbf{leave}; [\![t]\!]\\
[\![t_0\ t_1]\!] &= \textbf{push}([\![t_1]\!]); [\![t_0]\!]
\end{aligned}$$

VM transition:

$$\begin{aligned}
c &\Longrightarrow \langle c,\ 0,\ \cdot,\ \texttt{Halt}\rangle\\
\langle \textbf{access}\ n,\ 0,\ E,\ k\rangle &\Longrightarrow \langle k,\ E(n)\rangle\\
\langle \textbf{close}(c),\ 0,\ E,\ k\rangle &\Longrightarrow \langle k,\ \langle E, c\rangle\rangle\\
\langle \textbf{push}(c'); c,\ 0,\ E,\ k\rangle &\Longrightarrow \langle c,\ 0,\ E,\ \texttt{EvArg}(c', E, k)\rangle\\
\langle \textbf{enter}; c,\ 0,\ E,\ k\rangle &\Longrightarrow \langle c,\ 1,\ E,\ \texttt{Quote}(k)\rangle\\
\langle \textbf{leave}; c,\ 1,\ E,\ k\rangle &\Longrightarrow \langle c,\ 0,\ E,\ \texttt{Unquote}(k)\rangle\\
\langle \textbf{access}\ n,\ \ell,\ E,\ k\rangle &\Longrightarrow \langle k,\ \textbf{access}\ n\rangle & (\ell \geq 1)\\
\langle \textbf{close}(c),\ \ell,\ E,\ k\rangle &\Longrightarrow \langle c,\ \ell,\ E\uparrow^{\ell},\ \texttt{QAbs}(k)\rangle & (\ell \geq 1)\\
\langle \textbf{push}(c'); c,\ \ell,\ E,\ k\rangle &\Longrightarrow \langle c,\ \ell,\ E,\ \texttt{QApp'}(c', \ell, E, k)\rangle & (\ell \geq 1)\\
\langle \textbf{enter}; c,\ \ell,\ E,\ k\rangle &\Longrightarrow \langle c,\ \ell+1,\ E,\ \texttt{QNext}(k)\rangle & (\ell \geq 1)\\
\langle \textbf{leave}; c,\ \ell+1,\ E,\ k\rangle &\Longrightarrow \langle c,\ \ell,\ E,\ \texttt{QPrev}(k)\rangle & (\ell \geq 1)\\
\\
\langle \texttt{EvArg}(c, E, k),\ v\rangle &\Longrightarrow \langle c,\ 0,\ E,\ \texttt{EvBody}(v, k)\rangle\\
\langle \texttt{EvBody}(\langle E, c\rangle, k),\ v\rangle &\Longrightarrow \langle c,\ 0,\ v :: E,\ k\rangle\\
\langle \texttt{Quote}(k),\ c\rangle &\Longrightarrow \langle k,\ \ulcorner c \urcorner\rangle\\
\langle \texttt{Unquote}(k),\ \ulcorner c \urcorner\rangle &\Longrightarrow \langle k,\ c\rangle\\
\langle \texttt{QAbs}(k),\ c\rangle &\Longrightarrow \langle k,\ \textbf{close}(c)\rangle\\
\langle \texttt{QApp'}(c', \ell, E, k),\ c\rangle &\Longrightarrow \langle c',\ \ell,\ E,\ \texttt{QApp}(c, k)\rangle\\
\langle \texttt{QApp}(c, k),\ c'\rangle &\Longrightarrow \langle k,\ \textbf{push}(c'); c\rangle\\
\langle \texttt{QNext}(k),\ c\rangle &\Longrightarrow \langle k,\ \textbf{enter}; c\rangle\\
\langle \texttt{QPrev}(k),\ c\rangle &\Longrightarrow \langle k,\ \textbf{leave}; c\rangle\\
\langle \texttt{Halt},\ v\rangle &\Longrightarrow v
\end{aligned}$$

Fig. 3. The derived uniform compiler and VM with low-level code generation.

```
(* eval3' : term * int -> inst list *)
let rec eval3' (t, l) = match t, l with
  Var n , 0 -> [Access n]  | Var n, l -> [QVar n]
| App (t0, t1), 0 -> Push (eval3' (t1, 0)) :: eval3' (t0, 0)
| App (t0, t1), l -> PushQApp (eval3' (t1, l)) :: eval3' (t0, l)
...
(* appK3' : cont * value -> value *)
let rec appK3' (k, v) = (* the same as appK2 *) ...
| Cont7 (t2, k), Quot t3 -> appK3' (k, Quot (App (t2, t3)))
...
(* appC3' : compt * env * cont -> value *)
and appC3' (c, e, k) = match c with
  [Access n] -> appK3' (k, List.nth e n)
| [QVar n] -> appK3' (k, Quot (Var n))
| Push c1::c0 -> appC3' (c0, e, Cont1 (c1, e, k))
| PushQApp c1::c0 -> appC3' (c0, e, Cont6 (c1, e, k))
...
(* main3' : term -> value *)
let main3' t = appC3' (eval3' (t, 0), [], Cont0)
```

The resulting instruction set is twice as large as that for uniform compilation. Instructions PushQXXX push onto a continuation stack a marker that represents the corresponding term constructor; the marker is eventually consumed by appK3' to attach a term constructor to the current result: for example, Cont6 and Cont7 are markers for application.

Unfortunately, we fail to derive a VM for low-level code generation. This is simply because the compiler now takes a *pair* of a term and a level but a level is missing around term constructors in appK3' or appC3'!

We think that this failure is inherent in multi-level languages. In a multi-level language, one language construct has different meanings, depending on where it appears: for example, in $\lambda^{\bigcirc}$, a λ-abstraction at level 0 evaluates to a function closure, whereas one at level $\ell > 0$ evaluates to quoted λ-abstraction at level $\ell - 1$. Now, notice that the compiler derived here is still uniform at levels greater than 0 (one term constructor is always compiled to the same instruction, regardless of its level). So, it would not be possible for a VM to emit different instructions without level information, which, however, has been compiled away. If the number of possible levels is bounded, "true" non-uniform compilation would be possible but would require different instructions for *each* level, which would be unrealistic. We conjecture that this problem can be solved by a hybrid of uniform and non-uniform compilation, which is left for future work.

5 Related Work

Implementation of Multi-Level Languages A most closely related piece of work is Wickline et al. [4], who have developed a compiler of ML$^{\Box}$, which is an extension of ML with the constructs of $\lambda^{\Box}$ [8], and the target virtual machine CCAM, an extension of the Categorical Abstract Machine [16]. The CCAM is equipped with, among others, a set of special (pseudo) instructions **emit** I, which emit the single instruction I to a code block and are used to implement generating extensions. The instruction **emit**, however, is not allowed to be nested because such nested emits would be represented by real instructions whose size is exponential in the depth of nesting. They developed a strategy for compiling nested quotation by exploiting another special instruction **lift** to transform a value into a code generator that generates the value and the fact that environments are first-class values in the CAM. In short, their work supports both non-uniform compilation and low-level code generation in one system. Unfortunately, the design of the abstract machine is fairly ad hoc and it is not clear how the proposed compilation scheme can be exported to other combinations of programming languages and VMs. Our method solves the exponential blow-up problem above simply because a compound instruction **emit** I is represented by a single VM instruction I. Although our method does not support non-uniform compilation with low-level code generation, it would be possible to derive a compiler and a VM for one's favorite multi-level language in a fairly systematic manner. It might be interesting future work to incorporate their ideas into our framework to realize non-uniform compilation with low-level code generation.

MetaOCaml[6] is a multi-level extension of Objective Caml[7]. Calcagno et al. [23] have reported its implementation by translation to a high-level language with datatypes for ASTs, gensyms, and run-time compilation but do not take direct low-level code generation into account. We believe our method is applicable to MetaOCaml, too.

As mentioned in Section 1, there are several practical systems that are capable of run-time low-level (native or VM) code generation. Tempo [14] is a compile-time and run-time specialization system for the C language; DyC [15] is also a run-time specialization system for C; ‘C [12] is an extension of C, where programmers can explicitly manipulate, compile, and run code fragments as first-class values with (non-nested) backquote and unquote; Fabius [24] is a run-time specialization system for a subset of ML. They are basically two-level systems but Tempo supports multi-level specialization by incremental self-application [2, 25]. The code-size blowup problem is solved by the template filling technique [14, 26], which amounts to allowing the operand to the emit instruction to be (a pointer to) a *block* of instructions.

Functional Derivation of Abstract and Virtual Machines. Ager et al. describe derivations of abstract and virtual machines from evaluation functions by program transformation [18, 17] and have shown that the Krivine machine [27] is derived from a call-by-name evaluator and that the CEK machine [22] indeed corresponds to a call-by-value evaluator. They also applied the same technique to call-by-need [28], monadic evaluators [29], or strong reduction [19, 17]. How-

[6] `http://www.metaocaml.org/`
[7] `http://caml.inria.fr/ocaml/`

ever, they mainly focus on different evaluation strategy or side-effects and have not attempted to apply their technique to multi-level languages.

6 Conclusions

In this paper, we have shown derivations of compilers and VMs for a foundational multi-level language $\lambda^{\bigcirc}$. We have investigated the two compilation schemes of uniform compilation, which compiles a term constructor to the same instruction regardless of the level at which the term appears, and non-uniform compilation, which generates different instructions from the same term according to its level, and have shown that the former is more suitable for low-level code generation. Our derivation is fairly systematic and would be applicable to one's favorite multi-level language. In fact, although omitted from this paper, we have successfully derived a compiler and a VM for another calculus $\lambda^{\Box}$ [8].

The final derivation step for low-level code generation may appear informal and ad hoc. We are developing a formal translation based on function fusion.

Although it would not be easy to implement our machines for uniform compilation *directly* by the current, real processor architecture, we think they still can be implemented fairly efficiently as a VM. Our future work includes implementation of a uniform compiler and a corresponding VM by extending an existing VM, such as the ZINC abstract machine [30]. We believe our method is applicable to VMs with different architectures, which correspond to different evaluation semantics of the λ-calculus, and is useful to see how they can be extended for multi-level languages.

Acknowledgments We thank anonymous reviewers for providing useful comments and for pointing out missing related work.

References

1. Nielson, F., Nielson, H.R.: Two-Level Functional Languages. Cambridge University Press (1992)
2. Glück, R., Jørgensen, J.: Efficient multi-level generating extensions for program specialization. In: Proc. of PLILP. LNCS 982. (1995) 259–278
3. Davies, R.: A temporal-logic approach to binding-time analysis. In: Proc. of IEEE LICS. (July 1996) 184–195
4. Wickline, P., Lee, P., Pfenning, F.: Run-time code generation and Modal-ML. In: Proc. of ACM PLDI. (1998) 224–235
5. Taha, W., Benaissa, Z.E.A., Sheard, T.: Multi-stage programming: Axiomatization and type-safety. In: Proc. of ICALP. LNCS 1443. (1998) 918–929
6. Moggi, E., Taha, W., Benaissa, Z.E.A., Sheard, T.: An idealized MetaML: Simpler, and more expressive. In: Proc. of ESOP. LNCS 1576. (1999) 193–207
7. Taha, W., Sheard, T.: MetaML and multi-stage programming with explicit annotations. Theoretical Computer Science **248** (2000) 211–242
8. Davies, R., Pfenning, F.: A modal analysis of staged computation. Journal of the ACM **48**(3) (2001) 555–604

9. Taha, W., Nielsen, M.F.: Environment classifiers. In: Proc. of ACM POPL. (2003) 26–37
10. Calcagno, C., Moggi, E., Taha, W.: ML-like inference for classifiers. In: Proc. of ESOP. LNCS 2986. (2004) 79–93
11. Yuse, Y., Igarashi, A.: A modal type system for multi-level generating extensions with persistent code. In: Proc. of ACM PPDP. (2006) 201–212
12. Poletto, M., Hsieh, W.C., Engler, D.R., Kaashoek, M.F.: 'C and tcc: A language and compiler for dynamic code generation. ACM Transactions on Programming Languages and Systems **21**(2) (1999) 324–369
13. Masuhara, H., Yonezawa, A.: Run-time bytecode specialization: A portable approach to generating optimized specialized code. In: Proc. of PADO-II. LNCS 2053. (2001) 138–154
14. Consel, C., Lawall, J.L., Meur, A.F.L.: A tour of Tempo: A program specializer for the C language. Science of Computer Programming **52**(1–3) (2004) 341–370
15. Grant, B., Mock, M., Philipose, M., Chambers, C., Eggers, S.J.: DyC: An expressive annotation-directed dynamic compiler for C. Theoretical Computer Science **248**(1–2) (2000) 147–199
16. Cousineau, G., Curien, P.L., Mauny, M.: The categorical abstract machine. Science of Computer Programming **8**(2) (1987) 173–202
17. Ager, M.S., Biernacki, D., Danvy, O., Midtgaard, J.: From interpreter to compiler and virtual machine: A functional derivation. Technical Report RS-03-14, BRICS (March 2003)
18. Ager, M.S., Biernacki, D., Danvy, O., Midtgaard, J.: A functional correspondence between evaluators and abstract machines. In: Proc. of ACM PPDP. (2003) 8–19
19. Grégoire, B., Leroy, X.: A compiled implementation of strong reduction. In: Proc. of ACM ICFP. (2002) 235–246
20. Kohlbecker, E., Friedman, D.P., Felleisen, M., Duba, B.: Hygienic macro expansion. In: Proc. of ACM LFP. (1986) 151–161
21. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. Higher-Order Symbolic Computation **11**(4) (1998) 363–397
22. Felleisen, M., Friedman, D.P.: Control operators, the SECD machine, and the λ-calculus. In: Proc. Formal Description of Prog. Concepts III. (1986) 193–217
23. Calcagno, C., Taha, W., Huang, L., Leroy, X.: Implementing multi-stage languages using ASTs, gensym, and reflection. In: Proc. of GPCE. LNCS 2830. (2003) 57–76
24. Leone, M., Lee, P.: Optimizing ML with run-time code generation. In: Proc. of ACM PLDI. (1996) 137–148
25. Marlet, R., Consel, C., Boinot, P.: Efficient incremental run-time specialization for free. In: Proc. of ACM PLDI. (1999) 281–292
26. Consel, C., Noël, F.: A general approach for run-time specialization and its application to C. In: Proc. of ACM POPL. (1996) 145–156
27. Krivine, J.L.: A call-by-name lambda-calculus machine. Available online from `http://www.pps.jussieu.fr/~krivine`
28. Ager, M.S., Danvy, O., Midtgaard, J.: A functional correspondence between call-by-need evaluators and lazy abstract machines. Information Processing Letters **90**(5) (2004) 223–232
29. Ager, M.S., Danvy, O., Midtgaard, J.: A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. Theoretical Computer Science **342**(1) (2005) 149–172
30. Leroy, X.: The ZINC experiment: An economical implementation of the ML language. Technical Report 117, INRIA (1990)