

# Compiling Embedded Programs to Byte Code\*

Morten Rhiger

BRICS\*\*

Department of Computer Science

University of Aarhus\*\*\*

E-mail: [mrhiger@brics.dk](mailto:mrhiger@brics.dk)

Home page: <http://www.brics.dk/~mrhiger>

**Abstract.** Functional languages have proven substantially useful for hosting embedded domain-specific languages. They provide an infrastructure rich enough to define both a convenient syntax for the embedded language, a type system for embedded programs, and an evaluation mechanism for embedded programs. However, all existing host languages either interpret embedded programs instead of compiling them or require an expensive pre-compilation phase. In this article we close this gap in an implementation of the functional language OCaml: We provide a library of OCaml byte-code combinators that is reminiscent of quasi-quotation in Lisp and of ‘C and that enables just-in-time compilation of embedded programs. We illustrate these byte-code combinators on a prototypical domain-specific language.

**Keywords.** Just-in-time compilation, OCaml, domain-specific language, embedded language.

## 1 Introduction

Embedded languages have been devised as an alternative to implementing compilers for domain-specific languages [2, 31]. An embedded language adds domain-specific functionality (such as domain-specific values, their types, and operations on them) to an existing general-purpose language. The general-purpose language provides the domain-independent linguistic features (such as a type system and an evaluation strategy) and the means to execute programs (such as a compiler or an interpreter). This style was already envisioned in the 1960’s by Landin who observed that the design of programming languages splits into “the choice of written appearances of programs” and “the choice of abstract entities that can be referred to in the language” [23, 29].

Embedded languages provide practically useful compromises in both language development and application development. The language designer need

---

\* Appears in the proceedings of PADL 2002.

\*\* Basic Research in Computer Science ([www.brics.dk](http://www.brics.dk)), funded by the Danish National Research Foundation.

\*\*\* Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark.

not define and implement new domain-specific languages from scratch and the application programmer need not use general-purpose languages to solve domain-specific problems. Furthermore, embedded languages that share the same host language can easily be combined without the need for foreign-language interfaces.

Declarative languages provide powerful domain-independent linguistic features that make them well suited as host languages. Typed functional languages, such as Haskell [14] and ML [28], are particularly useful as host languages because they provide a rich infrastructure of higher-order functions, polymorphic types, and modules [20]. Like all other languages, an embedded language consists of

- a syntax of valid programs,
- a static semantics (such as, e.g., a type system), and
- a dynamic semantics (such as, e.g., a definitional interpreter or compiler).

Functional languages often enable convenient syntactic representations of embedded programs using either macro systems or a mixture of prefix and infix operators. Recent work have also shown how embedded programs can be type-checked in a statically typed host language using “phantom types” [12, 15, 22, 24, 32]. Currently, however, embedded programs must either be translated into the host languages by a pre-processor [12] or they must be executed by an interpreter implemented in the host language. (If the domain-specific operations cannot be expressed in the host language then the evaluation mechanism for embedded programs is provided by a stand-alone processor. We do not consider this third alternative.) Neither of these approaches is satisfactory. The first because integrating a pre-processor is difficult and inflexible in the presence of the interactive sessions of most functional languages. The second because it introduces an overhead of interpreting embedding programs. It would be desirable to enable direct execution of embedded programs without the burden of pre-processing and without the penalty of interpretation.

In this article we extend OCaml [27], a dialect of ML, with support for generating embedded programs directly as executable code. We provide a collection of combinators for constructing higher-order programs as OCaml byte-code instructions instead of as text and we provide the means to execute such byte-code instructions. The combinators can be used to translate an interpreter for embedded programs into a compiler that directly generates executable byte-code instructions. Existing compilers for embedded languages generate target code as program text which must be then be compiled into executable code [12]. These compilers can essentially be seen as source-to-source optimizing macro transformers. In comparison, we take one step further and generate byte-code executables directly. In addition, our approach also enables source-to-source optimizing macros.

This article uses a prototypical domain-specific language, namely regular expressions, as a running example. However, the technique presented here is directly applicable to other examples. The hitherto largest application is in

specializing the term rewriter of MetaPRL, a theorem prover implemented in OCaml [32].

In Section 2 we embed regular expressions into OCaml using an interpreter. This example highlights the limitations of interpreting embedded programs. In Section 3 we present our main contribution, an implementation of byte-code combinators for OCaml. We return to embedding regular expressions into OCaml using the byte-code combinators in Section 4. In Section 5 we briefly discuss the experiments with the MetaPRL theorem prover. We outline related work in Section 6 and conclude in Section 7.

## 2 Interpreting Embedded Programs

Lex and Yacc-like pre-processing tools are prototypical examples of embedded domain-specific applications. They provide a practically useful alternative to implementing parsers for regular or context-free languages directly as general-purpose programs. To this end, they extend an existing general-purpose language with high-level declarations for grammars. The pre-processor compiles an embedded grammar into a parser implemented in the host language while the remaining program parts supply the code that constructs tokens or parse trees.

To illustrate embedded languages in OCaml, we consider a particularly simple variant of parsing, namely regular-expression matching. The concrete syntax of regular expressions is as follows.

$$r ::= \mathbf{0} \mid \mathbf{1} \mid c \mid r_1 + r_2 \mid r_1 r_2 \mid r^*$$

where  $c$  denotes a character symbol. The language accepted by a regular expression is defined inductively as follows. (Here,  $\epsilon$  denotes the empty string.)

$$\begin{aligned} \mathcal{L}(\mathbf{0}) &= \emptyset \\ \mathcal{L}(\mathbf{1}) &= \{\epsilon\} \\ \mathcal{L}(c) &= \{c\} \\ \mathcal{L}(r_1 + r_2) &= \mathcal{L}(r_1) \cup \mathcal{L}(r_2) \\ \mathcal{L}(r_1 r_2) &= \mathcal{L}(r_1) \mathcal{L}(r_2) \\ \mathcal{L}(r^*) &= \bigcup_{i \geq 0} (\mathcal{L}(r))^i \end{aligned}$$

For the purpose of simplification, we use a regular-expression matcher that backtracks. (An alternative is to translate a regular expression into a deterministic automaton which can be either interpreted or compiled.) Figure 1 presents the abstract syntax of regular expressions and a matching function `accept`. The matcher is applied to a regular expression, a list of characters, and a continuation. If the regular expression matches a portion of the characters then the continuation is applied to the remaining characters. Otherwise, the matcher directly returns false. The clause for  $r^*$  (STAR) is recursive and uses an explicit fixed-point operator. In this clause, the continuation immediately fails if matching does not progress. This extra check ensures that matching pathological regular expressions such as  $\mathbf{1}^*$  terminates [7, 17].

```

type re = ZERO | ONE | CHAR of char
        | CAT of re * re | SUM of re * re | STAR of re

let rec fix f x = f (fix f) x

let rec accept re s k =
  match re with
  | ZERO          -> false
  | ONE           -> k s
  | CHAR c        ->
    (match s with c'::s' -> c = c' && k s'
     | []          -> false)
  | CAT (re1, re2) -> accept re1 s (fun s' -> accept re2 s' k)
  | SUM (re1, re2) -> accept re1 s k || accept re2 s k
  | STAR re1       ->
    fix (fun star s ->
        k s || accept re1 s
          (fun s' -> if s==s' then false else star s'))
    s

let matches re s = accept re s (fun s' -> s' = [])

```

Fig. 1. Interpreted regular expression matching in OCaml

Embedding regular expressions into OCaml using an interpreter is more flexible than the Lex and Yacc-like approach where grammars are compiled into parsers in a pre-processing phase. The reason is that regular expressions, our notion of grammars, are first-class objects, and hence can be manipulated within OCaml. For example, instead of the cumbersome prefix data type constructors we can use infix operators such as the following.

```

let (++) re1 re2 = SUM(re1, re2)
let (@@) re1 re2 = CAT(re1, re2)

```

A regular expression such as  $a(b + c)$  can then be transliterated in OCaml as

```

CHAR 'a' @@ (CHAR 'b' ++ CHAR 'c')

```

which matches the strings  $\{ab, ac\}$ . We can also exploit algebraic identities such as  $\mathcal{L}(\mathbf{0} + r) = \mathcal{L}(r + \mathbf{0}) = \mathcal{L}(r)$  and  $\mathcal{L}(\mathbf{1}r) = \mathcal{L}(r\mathbf{1}) = \mathcal{L}(r)$  to generate improved regular expressions. For example, alternative definitions of  $++$  and  $@@$  read as follows.

```

let (++) re1 re2 =
  match re1, re2 with
  | ZERO, _ -> re2

```

```

| _, ZERO -> re1
| _, - -> SUM(re1, re2)

let (@@) re1 re2 =
  match re1, re2 with
  | ONE, _ -> re2
  | _, ONE -> re1
  | _, - -> CAT(re1, re2)

```

Finally, the following two functions produce regular expressions for sums  $c_1 + c_2 + \dots + c_n$  and concatenations  $c_1c_2 \dots c_n$  from a given string of characters " $c_1c_2 \dots c_n$ ". They serve as macros defining regular expressions for classes of characters and for sequences of characters, respectively. (Here, `explode` maps a string into the list of its characters.)

```

let cclass s =
  List.fold_right (++)
    (List.map (fun c -> CHAR c) (explode s))
    ZERO

let string s =
  List.fold_right (@@)
    (List.map (fun c -> CHAR c) (explode s))
    ONE

```

The regular-expression matcher may be used in a parser to recognize tokens of characters. A fragment of such a parser reads as follows. (Here we assume, for simplicity, that the input is already separated into chunks of characters).

```

let digit = cclass "0123456789"
let alpha = cclass "abcdefghijklmnopqrstuvwxyz"
let alnum = alpha ++ digit

let lex s =
  if matches (string "if" ++ string "then" ++ string "else") s then
    (* Code for generating keywords *)
  else if matches (STAR digit) s then
    (* Code for generating numerals *)
  else if matches (alpha @@ STAR alnum) s then
    (* Code for generating identifiers *)
  ...
  else error "Unexpected token"

```

However, such a function inherits an interpretive overhead from matching regular expressions against a string. In the following section we develop the machinery that enables us to compile regular expressions (and other embedded programs) into efficient OCaml code.

### 3 Run-Time Byte-Code Generation in OCaml

The implementation of OCaml consists of a byte-code compiler and a runtime system with a virtual machine for running byte-code executables. The compiler is implemented in OCaml and the run-time system is implemented in C. The run-time system consists of a byte-code interpreter, a garbage collector, and a set of pre-defined library procedures.

The byte-code compiler consists of modules each implementing a phase. The initial input is a stream of characters, either read from a file (in batch mode) or from standard input (in interactive mode). Each phase produces a refined representation of the source program.

- Lexical analysis and parsing: Together, lexical analysis and parsing read a sequence of characters and produce an abstract-syntax tree.
- Type analysis: This phase type-checks the source program. It produces a type-annotated abstract-syntax tree.
- Semantics-preserving translation: This phase translates OCaml expressions into an extended  $\lambda$ -calculus. The major difference between an OCaml expression and a  $\lambda$ -term is that modules and functors are represented as tuples and higher-order functions in the  $\lambda$ -terms.
- Code generation: This phase produces a list of symbolic byte-code instructions from a  $\lambda$ -term.
- Byte-code emission: This phase writes a list of symbolic byte-code instructions to a file (in batch mode) or into memory (in interactive mode).

Since OCaml is an interactive system, the compiler is present during all stages of execution. It is our intention to use parts of the OCaml compiler to generate byte-code instructions directly for embedded programs. The first step is therefore to translate embedded programs into a suitable representation of OCaml programs. There are several viable representations of OCaml programs corresponding to the input to each of the phases described above. (We describe the translation technique itself in Section 4.) At the extremes, we can represent programs as their text and pass them through all phases, from lexical analysis and parsing to byte-code emission, or we can represent programs as symbolic byte-code instructions and pass them through only the byte-code emission phase. Neither of these approaches are satisfactory, however: Passing programs through the entire compiler is typically too costly and generating byte-code instructions directly is typically too cumbersome.

As a convenient intermediate representation of OCaml programs, we shall instead choose the  $\lambda$ -terms that are input to the code generation phase. These terms are defined by the following data type in the compiler.

```
type lambda =
  Lvar      of Ident.t
| Lconst   of structured_constant
| Lapply   of lambda * lambda list
| Lfunction of function_kind * Ident.t list * lambda
```

```

| Llet      of let_kind * Ident.t * lambda * lambda
| Lprim    of primitive * lambda list
| Lifthenelse of lambda * lambda * lambda
| Lsequence of lambda * lambda
| ...

```

The compiler provides a function

```

val comp_expr :
  env -> lambda -> int
  -> instruction list -> instruction list

```

for compiling  $\lambda$ -terms. Here, `env` is the type of compilation environments and `instruction` is the type of symbolic byte-code instructions. The call

```
comp_expr env exp sz cont
```

compiles an expression `exp` in compilation environment `env`, in a stack frame of size `sz`, and where `cont` is the list of instructions to execute afterwards (i.e., a byte-code representation of the continuation of the expression). The result is a list of instructions that evaluate `exp` and then perform `cont`.

The data type `lambda` defines an abstract-syntax tree of  $\lambda$ -terms. It provides a representation of programs which is more flexible than a representation of symbolic byte-code instructions. For example, identifiers are represented by their name (`Ident.t`) instead of by their position on the run-time stack and functions are represented as  $\lambda$ -abstractions (`Lfunction`) instead of as closures. However,  $\lambda$ -terms are constructed as abstract-syntax trees only to be mapped into byte-code instructions by the function `comp_expr` immediately afterwards. We eliminate the syntax dispatch of `comp_expr` by using a non-standard *Church-encoded* representation of  $\lambda$ -terms. To this end,  $\lambda$ -term are constructed by *byte-code combinators* instead of by data-type constructors.

A byte-code combinator is a triple that encapsulates enough information that it can be reassembled into a sequence of byte-code instructions.

1. A byte-code combinator carries a list of the variables that occur free in the expression it represents. Such a list is used to construct the byte-code instructions for creating closures.
2. If a byte-code combinator represents a variable, then it carries the name of that variable. This information is used in generating byte-code combinators for let-expressions that preserve tail calls. Instead of generating byte-code instructions corresponding to an expression `let x = E in x` they generate simply the byte-code instructions corresponding to `E`.
3. A byte-code combinator carries a function that generates the actual byte-code instructions. It takes two arguments, an environment mapping variable names to stack or environment positions and a byte-code representation of the continuation. It adds code for the current byte-code combinator to the front of the continuation.

Byte-code combinators efficiently support two key operations, namely concatenation of byte-code instructions and instantiation of free variables. When the code-generating function of a complete byte-code combinator is applied to an environment and a continuation, byte-code instructions are generated in a backwards manner using the continuation and the positions of variables are resolved using the environment. There is no copying of the generated byte-code instructions and they are not traversed once they are created. The type of byte-code combinators, `exp`, is as follows.

```
type code = instruction list
type exp  = ide list * ide option * (env * code -> code)
```

For byte-code combinators involving variables and bindings (such as variables,  $\lambda$ -abstractions, and let-expressions) we provide both a low-level first-order interface and a higher-order interface similar to a higher-order abstract syntax [30]. The low-level interface allows direct generation and manipulation of variables,  $\lambda$ -abstractions, and let-expressions. The higher-order interface groups common patterns involving bindings into convenient functions. In addition to the following byte-code combinators, we have implemented byte-code combinators for operations on tuples, lists, integers, booleans, strings, mutable data structures, and global variables.

```
mkint   : int -> exp
mkapp   : exp -> exp -> exp
mklam   : (exp -> exp) -> exp
mklet   : exp -> (exp -> exp) -> exp
mkif    : exp -> exp -> exp -> exp
...
```

Figure 2 show the byte-code combinators for generating integers and applications. The byte-code combinator for applications generates optimized tail calls and it groups together several curried calls into one instruction. These byte-code combinators generate the following instructions.

<code>Kconst <math>p</math>:</code>	Loads the accumulator with the operand $p$ .
<code>Kpush:</code>	Pushes the contents of the accumulator onto the stack.
<code>Kappterm(<math>i, j</math>):</code>	Performs a tail call with $i$ arguments and where the current stack frame contains $j$ elements.
<code>Kapply <math>i</math>:</code>	Performs an ordinary non-tail call with $i$ arguments.
<code>Kreturn <math>n</math>:</code>	Removes $n$ arguments from the stack and then pops a saved return address and a saved environment off the stack.

On the average, the definition of a byte-code combinator requires less than 8 lines of OCaml code and it typically generates between 3 and 6 instructions. Experiments show that the byte-code combinators typically generate up to 170000 byte-code instructions per second [32].

To run a byte-code combinator, its code-generating function is first applied to an empty environment and a continuation consisting only of a return instruction.

```

let mkint i =
  ([],
   None,
   fun(rho, k) ->
     Kconst(Lambda.Const_base(Asttypes.Const_int i)) :: k)

let mkapp (vsa, _, fa) ((vsb, _, fb) as arg) =
  (union(vsa, vsb),
   None,
   function
     (rho, Kreturn n :: k)      ->
       fb(rho, Kpush :: fa(installTmp rho, Kappterm(1, n+1) :: k))
   | (rho, Kappterm(i, j) :: k) ->
       fb(rho, Kpush :: fa(installTmp rho, Kappterm(i+1, j+1) :: k))
   | (rho, k)                  ->
       fb(rho, Kpush :: fa(installTmp rho, Kapply 1 :: k)))

```

**Fig. 2.** Example byte-code combinators for integers and applications

The result is a list of symbolic instructions. Using functions provided by OCaml's interactive run-time system, the list of symbolic instructions is then written to the memory in the form of executable byte-code instructions. We use the following two auxiliary functions as a front-end to the internals of the run-time system.

```

run_code : instruction list -> 'a
run_exp  : exp -> 'a

```

The function `run_code` writes a list of symbolic byte-code instructions to the memory, relocates global pointers in the allocated block, and passes it to the virtual machine for execution. The function `run_exp` instantiates a byte-code combinator and executes the resulting list of instructions. It is defined in terms of `run_code` as follows.

```

let run_exp (_, _, f) = run_code (f ([], [Kreturn 1]))

```

Together, byte-code combinators and `run_exp` support run-time code generation in the same way as Lisp-like S-expressions and `eval` [3].

## 4 Compiling Embedded Programs

We return to the regular-expression matcher from Section 2 but now using the byte-code combinators to compile regular expressions instead of interpreting them. To this end, we derive a compiler for regular expressions from the interpreter presented in Figure 1 using standard partial-evaluation techniques [21].

```

let rec mkaccept re s k =
  match re with
  | ZERO          -> mkfalse
  | ONE           -> k s
  | CHAR c        ->
    mkif (mknnull s)
      mkfalse
      (mkand (mkeqint (mkint (int_of_char c)) (mkhd s))
        (mklet (mktl s) (fun s' -> k s'))))
  | CAT (re1, re2) -> mkaccept re1 s (fun s' -> mkaccept re2 s' k)
  | SUM (re1, re2) ->
    generalize k
    (fun k' -> mkor (mkaccept re1 s k') (mkaccept re2 s k'))
  | STAR re1      ->
    mkapp (mkfix (mklam2 (fun star s ->
      mkor (k s)
        (mkaccept re1 s (fun s' ->
          mkif (mkeqint s s')
            mkfalse
            (mkapp star s'))))))))
    s

let mkmatches re : char list -> bool =
  run_exp (mklam (fun s -> mkaccept re s (fun s' -> mknnull s')))

```

**Fig. 3.** Embedded compiler for regular expressions

Such a compiler is the “generating extension” of the regular-expression matcher and can be obtained by self-applying a partial evaluator [16]. However, when the operations executed at translation time and those executed at run time are cleanly separated, which is indeed the case with the regular-expression matcher, then a generating extension can simply be obtained from the interpreter by replacing run-time operations with code-generating operations. We use byte-code combinators as code-generating primitives.

The compiler for regular expressions is shown in Figure 3. In partial-evaluation terminology, the regular expression is “static”, the list of characters “dynamic”, and continuations are static functions on dynamic arguments (lists of characters) producing dynamic results (booleans). Hence, we have systematically replaced operations on lists of characters by the equivalent byte-code combinators. Apart from the byte-code combinators already mentioned in Section 3, we also use byte-code combinators for generating booleans (`mkfalse` and `mktrue`), short-circuit boolean conjunction and disjunction (`mkand` and `mkor`, defined in terms of `mkif`), integer comparison (`mkeqint`), and fixed-point operators (`mkfix`, defined in terms of the global variable `fix`).

Generating extensions for continuation-passing style programs often face the problem of code duplication: When a static continuation is applied twice its corresponding code may be inlined twice in the generated program. In the regular expression compiler, continuations are passed to both recursive calls in the clause for `SUM`. Therefore, to avoid code duplication, this continuation is bound and inlined in the generated program using the following generalization directive which produces a standard two-level  $\eta$ -redex.

```
let generalize k f =
  mklet (mklam k) (fun k' -> f (mkapp k'))
```

The main function of the regular-expression compiler generates and evaluates the byte-code combinator corresponding to a given regular expression. It has the type

```
mkmatches : re -> char list -> bool
```

which agrees with the type of the interpreter for regular expressions.

As an example, Figure 4 shows the byte-code instructions corresponding to the regular expression `a`. (The ASCII code for the character `a` is 97.) Matching compiled regular expressions using Figure 3 is more efficient than matching interpreted regular expressions using Figure 1. For example, matching the regular expression `language` against the first 600 words in this article takes 3.09 ms (milliseconds) using the interpreted version and 1.33 ms using the compiled version. (The measures were performed on an IBM ThinkPad 600 equipped with a 266MHz Pentium II processor and 96 Mb of RAM and which runs RedHat Linux 2.2.1. We have not measured the space usage. The running times are averaged over 1000 runs.) Matching the regular expression `{a, b, c, d, e, k, l, m, n}*` against the same 600 words takes 23.08 ms using the interpreted version and 13.73 ms using the compiled version.

The generated byte codes are executed by the OCaml virtual machine that itself runs “at native speed.” Thus, in the example presented here, a regular expression is matches by interpreting its byte-code counterpart (using the OCaml virtual machine). In comparison, we can compile the original regular-expression matcher using OCaml’s native code compiler. A regular expression is then matched by interpreting its abstract syntax (using the original matcher). Both cases have “one level” of interpretive overhead. The latter approach yields running times 0.40 ms and 2.61 ms for the regular expressions `language` and `{a, b, c, d, e, k, l, m, n}*`, respectively. In these cases, directly interpreting regular expressions (at native speed) is faster than interpreting their compiled byte-code counterparts (at native speed).

## 5 Application: The MetaPRL term rewriter

One application area for run-time code generation is in the proof-search engines of automated theorem provers. Higher-order logical frameworks provide

```

        closure L1, 0      ; fun s ->
        return 1          ;
L1:    acc 0              ;      if null s
        push              ;
        const 0a          ;
        eqint             ;
        branchifnot L4    ;
        const 0a          ;      then false
        branch L2         ;
L4:    acc 0              ;      else if 97 = hd s
        getfield 0        ;
        push              ;
        const 97          ;
        eqint             ;
        branchifnot L3    ;
        acc 0              ;      then let s' = tl s
        getfield 1        ;
        push              ;
        acc 0              ;      in
        push              ;
        const 0a          ;
        eqint             ;      null s'
        pop 1             ;
        branch L2         ;
L3:    const 0a          ;      else false
L2:    return 1          ;

```

Fig. 4. Byte-code instructions for the regular expression a.

an expressive foundation for reasoning about formal systems. They permit concise problem descriptions and re-use of logical models. MetaPRL is a logical programming environment that combines OCaml with a higher-order logical framework [18]. The MetaPRL tactic prover consists of a proof editor, logic definitions, and a refiner. The logic definitions describes the language of a logic. It also contains *primitive inference rules* that define axioms and theorems of the logic, *rewrites* that define computational equivalences, and *theorems* which provide proofs for derived inference rules and axioms. Inference rules are compiled to a byte-code language. The refiner interprets the byte-code instructions during rule application and term rewriting.

Since reasoning about programs is expensive, it is crucial that proof searching is efficient. To this end, MetaPRL provides specialized implementations for the parts of a logic, such as term representations and proof-search strategies [19]. Furthermore, the set of inference rules is fixed for a particular proof search so

the term rewriter can be specialized to the set of inference rules at the time a proof search is initiated.

We have applied the byte-code combinators presented in this article to specialize MetaPRL's term rewriter. Similar to the above, we have implemented the generating extension of the term rewriter in a straightforward manner. MetaPRL applies the term rewriter rather extensively. For example, reducing a factorial function with an argument of 100 requires several hundred thousands of rewrites of a total of around 1500 rewrites. Thus, it is eminently reasonable to specialize the term rewriter to the rewrites. We are currently working on benchmarks for these experiments.

## 6 Related Work

Elliot, Finne, and de Moor present an embedded compiler for a first-order language [12]. They base their compiler on macros that exploit algebraic identities to generate efficient source programs and on a separate back-end for code generation. In comparison, we have presented a general approach to evaluating embedded programs that enables programs to be interpreted or compiled directly into byte-code executables at run time.

Sperber and Thiemann present a run-time specializer for Scheme that generates byte code [33]. They have composed an existing partial evaluator for Scheme with an existing Scheme compiler producing byte code [35]. Their run-time specializer does not construct intermediate residual programs as text. Instead, they have obtained a set of byte-code combinators directly from the Scheme compiler by deforesting the data type of abstract-syntax trees. The result is a set of byte-code combinators similar to those presented in this article. Sperber and Thiemann use byte-code combinators to built residual programs in a partial evaluator whereas we suggest to use them directly to built generating extensions (i.e. compilers) from interpreters. Conceptually, both Sperber and Thiemann's combinators and the combinators presented in this article can be obtained from the (recursively descending) compilers for Scheme and OCaml, respectively. For practical purposes, however, both sets of combinators have been written by hand. (In fact, Sperber and Thiemann start out with a handwritten compiler for a subset of Scheme.)

Balat and Danvy have designed a run-time specializer for OCaml using type-directed partial evaluation [1]. They have composed standard type-directed partial evaluation with a compiler from normal forms into OCaml byte code. They have exploited the fact that normal forms form a subset of full OCaml to implement a fast dedicated compiler. In contrast, we do not use a compiler but integrate the generation of byte-code instructions into the byte-code combinators.

In dialects of Lisp, quasi-quotation and `eval` provide an interface to generating programs as S-expressions and executing them at run time [3]. S-expressions may be "taken apart" and they may contain nested applications of `eval` and nested quasiquations. Together, S-expressions, quasiquations, and `eval` there-

fore allows for “peep-hole optimizing” code at run-time before executing it and they support multi-stage code generation. In comparison, the byte-code combinators presented in this article do not support multi-level code generation and they may not be taken apart, e.g., by pattern matching. In order to provide an `eval` procedure, the Lisp compiler or interpreter must coexist with the run-time system. In comparison, executing a byte-code combinator only require a simple translation of symbolic instructions into (integer) byte codes. Hence, the OCaml compiler need not coexist at run-time and our solution can be used with compiled OCaml programs.

Traditional macro preprocessors, such as CamlP4 for OCaml and the macro systems of most dialects Lisp, sometimes support program specialization at compile time. However, since macros are expanded at compile time, these macro systems do not provide run-time specialization or run-time code generation. For example, it is not possible for a standard macro preprocessor alone to generate specialized regular-expression matchers based on regular expressions input by the user. This is exactly achieved by some form of run-time code generation such as `eval` or the byte-code combinators presented in this article.

`'C` is an extension of the C language with Lisp-like quasi-quotation for specifying dynamically generated code [13]. The main concerns of `'C` is generating efficient code at run time with as little overhead as possible. `'C` generates machine code instead of byte code as in this work.

Tempo is a run-time specialization systems for C [4]. Based on an initial binding-time signature, Tempo splits the source program into static and dynamic parts. While the dynamic parts do contain unknowns, they can be compiled into binary code by an almost standard C compiler. From the binding-time annotations the static compiler also generates a *template* that specifies how the pre-compiled blocks should be re-assembled at run time. When the source program is split into few large blocks, they may provide enough flow information that an optimizing C compiler can generate efficient code for them. In contrast, byte-code programs, such as those generated by the byte-code combinators that we present, are difficult to optimize since the set of instructions is highly specialized. Instead, byte-code run-time systems often provide a fast virtual machine.

The Fabius system by Leone and Lee is a run-time code generator of machine code for a first-order subset of ML [25, 26]. Fabius specializes curried functions to their first argument using standard partial evaluation techniques: Given an application  $f a$  in the source program, Fabius constructs the generating extension of  $f$  at compile time. The generating extension is then applied to the argument  $a$  at run time to produce the executable code corresponding the specialized version of  $f$  with respect of  $a$ . In comparison, our system generates byte code and is not restricted to specializing curried functions.

There is a strong similarity between an interpreter and its generating extension [6, 29]. Both can be expressed as a fold function over the data type for the input language. The only structural difference is that the interpreter uses evaluating primitives while the compiler uses code-generating primitives for operations on run-time values. We can emphasize this binding-time separation

by parameterizing an interpreter over a collection of primitives. Type-directed partial evaluation precisely expect such binding-time separated programs as input [5]. By instantiating the parameterized interpreter with either evaluating or code-generating primitives we obtain an interpreter or a compiler that generates executable code directly. Furthermore, by instantiating the interpreter with primitives that generate a textual representation of programs (such as a data type of OCaml terms) we obtain a translator that generates OCaml programs. Hence, this parameterized approach is able to express both pre-processing, interpretation, and just-in-time compilation. Using OCaml’s module system we can conveniently represent parameterized interpreters as functors and primitives as modules.

We have also used the byte-code combinators presented in this article as a code-generating back-end for a type-directed partial evaluator [32]. The result is a run-time specializer for typed higher-order programs. Using this system, we have successfully specialized the term rewriter in the back-end of a large theorem prover [18].

In typed languages that support program manipulation it is desirable that only well-typed programs are generated. Static typing in the context of run-time code generation, however, is still somewhat of an open problem. Davies’s  $\lambda^\circ$ -calculus [8], motivated by linear-time temporal logic, provides a type system able to describe standard partial evaluation. However, in  $\lambda^\circ$ , there is no way to express immediate evaluation of sub-terms because generated programs may contain free variables. Davies and Pfenning’s  $\lambda^\square$ -calculus [9], motivated by modal logic, provides a type system for manipulating closed program. In  $\lambda^\square$ , direct evaluation of generated programs can be expressed but these often contain administrative redexes [10]. At any rate, neither  $\lambda^\circ$  nor  $\lambda^\square$  have been designed for a strict language with mutable state such as OCaml.

## 7 Conclusions

The ability to generate code at run time can increase the efficiency of embedded programs while retaining their flexibility. We have designed a set of byte-code combinators for directly generating OCaml byte-code instructions. The implementation is integrated in the latest version of the OCaml compiler. It consists of less than 500 lines of OCaml code.

The byte-code combinators can be used as code-generating primitives for deriving a compiler, or generating extension, from an interpreter. This style of programming amounts to implementing generating extensions by hand and occurs in macro-systems and partial evaluation. We have illustrated this programming style using a regular-expression matcher but it also applies to other embedded languages.

**Acknowledgements.** The author would like to thank the anonymous referees for constructive suggestions and Olivier Danvy for extensive comments.

## References

1. Vincent Balat and Olivier Danvy. Strong normalization by type-directed partial evaluation and run-time code generation. In Xavier Leroy and Atsushi Ohori, editors, *Proceedings of the Second International Workshop on Types in Compilation*, number 1473 in Lecture Notes in Computer Science, pages 240–252, Kyoto, Japan, March 1998.
2. Thomas Ball, editor. *Proceedings of the Second USENIX Conference on Domain-Specific Languages*, Austin, Texas, October 1999.
3. Alan Bawden. Quasiquotation in Lisp. In Olivier Danvy, editor, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, number NS-99-1 in BRICS Note Series, pages 4–12, San Antonio, Texas, January 1999.
4. Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In Steele [34], pages 145–156.
5. Olivier Danvy. Type-directed partial evaluation. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School*, number 1706 in Lecture Notes in Computer Science, pages 367–411, Copenhagen, Denmark, July 1998. Springer-Verlag.
6. Olivier Danvy. Programming techniques for partial evaluation. In Friedrich L. Bauer and Ralf Steinbrüggen, editors, *Foundations of Secure Computation*, NATO Science series, pages 287–318. IOS Press Ohmsha, 2000.
7. Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International Conference on Principles and Practice of Declarative Programming*, Firenze, Italy, September 2001. ACM Press. To appear.
8. Rowan Davies. A temporal-logic approach to binding-time analysis. In Edmund M. Clarke, editor, *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.
9. Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In Steele [34], pages 258–283.
10. Rowan Davies and Frank Pfenning. A modal analysis of staged computation. Technical report CMU-CS-99-153, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1999. To appear in the Journal of the ACM.
11. Premkumar Devanbu and Jeff Poulin, editors. *Proceedings of the Fifth International Conference on Software Reuse*, Victoria, British Columbia, June 1998. IEEE Computer Society Press.
12. Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. In Walid Taha, editor, *Proceedings of the International Workshop on Semantics, Applications, and Implementation of Program Generation*, number 1924 in Lecture Notes in Computer Science, pages 9–27, Montréal, Canada, September 2000.
13. Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. ‘C: A language for high-level, efficient, and machine-independent dynamic code generation. In Steele [34], pages 131–144.
14. Joseph H. Fasel, Paul Hudak, Simon Peyton Jones, and Philip Wadler. Haskell special issue. *SIGPLAN Notices*, 27(5), May 1992.
15. Sigbjørn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. Calling hell from heaven and heaven from hell. In Peter Lee, editor, *Proceedings of the 1999*

- ACM SIGPLAN International Conference on Functional Programming*, pages 114–125, Paris, France, September 1999. ACM Press.
16. Yoshihiko Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. Reprinted from *Systems, Computers, Controls* 2(5), 1971.
  17. Robert Harper. Proof-directed debugging. *Journal of Functional Programming*, 9(4):463–469, July 1999.
  18. Jason Hickey. Nuprl-light: An implementation framework for higher-order logics. In William McCune, editor, *14th International Conference on Automated Deduction*, number 1249 in *Lecture Notes in Artificial Intelligence*, pages 395–399. Springer-Verlag, 1997.
  19. Jason J. Hickey and Aleksey Nogin. Fast tactic-based theorem proving. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 252–266. Springer-Verlag, 2000.
  20. Paul Hudak. Modular domain specific languages and tools. In Devanbu and Poulin [11], pages 134–142.
  21. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International Series in Computer Science. Prentice-Hall, 1993. Available online at <http://www.dina.kvl.dk/~sestoft/pebook/pebook.html>.
  22. Simon Peyton Jones, Erik Meijer, and Daan Leijen. Scripting COM components in Haskell. In Devanbu and Poulin [11], pages 224–233.
  23. Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.
  24. Daan Leijen and Erik Meijer. Domain specific embedded compilers. In Thomas Ball, editor, *Proceedings of the 2nd USENIX Conference on Domain-Specific Languages*, pages 109–122, 1999.
  25. Mark Leone and Peter Lee. Lightweight run-time code generation. In Peter Sestoft and Harald Søndergaard, editors, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Technical Report 94/9, University of Melbourne, Australia, pages 97–106, Orlando, Florida, June 1994.
  26. Mark Leone and Peter Lee. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 31, No 5, pages 137–148. ACM Press, May 1996.
  27. Xavier Leroy. *The Objective Caml system, release 3.01*. INRIA, Rocquencourt, France, March 2001.
  28. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.
  29. Lockwood Morris. The next 700 formal language descriptions. *Lisp and Symbolic Computation*, 6(3/4):249–258, 1993.
  30. Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In Mayer D. Schwartz, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 23, No 7, pages 199–208, Atlanta, Georgia, June 1988. ACM Press.
  31. Chris Ramming, editor. *Proceedings of the First USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, October 1997.
  32. Morten Rhiger. *Higher-Order Program Generation*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, July 2001.

33. Michael Sperber and Peter Thiemann. Two for the price of one: composing partial evaluation and compilation. In Ron K. Cytron, editor, *Proceedings of the ACM SIGPLAN'97 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 32, No 5, pages 215–225, Las Vegas, Nevada, June 1997. ACM Press.
34. Guy L. Steele, editor. *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1996. ACM Press.
35. Peter Thiemann. Combinators for program generation. *Journal of Functional Programming*, 9(5):483–525, 1999.