Program Generation for ML Modules (Short Paper)

Takahisa Watanabe
Department of Computer Science
University of Tsukuba
Tsukuba, Japan
takahisa@logic.cs.tsukuba.ac.jp

Yukiyoshi Kameyama
Department of Computer Science
University of Tsukuba
Tsukuba, Japan
kameyama@acm.org

Abstract

Program generation has been successful in various domains which need high performance and high productivity. Yet, programming-language supports for program generation need further improvement. An important omission is the functionality of generating modules in a type safe way. Inoue et al. have addressed this issue in 2016, but investigated only a few examples. We propose a language as an extension of (a small subset of) MetaOCaml in which one can manipulate and generate code of a module, and implement it based on a simple translation to MetaOCaml. We show that our language solves the performance problem in functor applications pointed out by Inoue et al., and that it provides a suitable basis for writing code generators for modules.

CCS Concepts • **Software and its engineering** → **General programming languages**; • **Theory of computation** → *Type theory*;

Keywords Program Generation, Modules, Type Safety, Program Transformation

ACM Reference Format:
Takahisa Watanabe and Yukiyoshi Kameyama. 2018. Program Generation for ML Modules (Short Paper). In *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'18).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3162072

1 Introduction

Multi-stage programming (MSP) is an attractive way to generate efficient code tailored to specific hardware, environment, or run-time parameters. After a number of studies for developing languages and systems for multi-stage programming (Scheme's quasi quotation, hygienic macro, Template Haskell etc.), research on type systems for MSP has lead to full-blown programming languages for MSP: MetaOCaml [5, 12] and Scala Lightweight Modular Staging (LMS) [10]. In these languages, type safety of code generators has stronger implication than one would have expected: it subsumes the type safety of all generated code regardless of run-time parameters.[1] Thus, these languages provide a solid basis for writing safe (or relatively safer) code generators. Recent successful examples include query engines [7], stream fusion [6] and generic programming [15].

Sticking to type safety sometimes leads to rather restricted expressivity of the language. One might want to generate not only code of expressions, but also code of types, declarations, and other syntactic objects such as modules, all of which have been considered difficult under statically typed MSP languages. In particular, guaranteeing type safety of generated code against types which do not exist at compile time would be difficult. Modules in ML-like languages involve declaration of types and values, hence, type safety for programs which generate (code of) a module is also challenging.

Inoue et al. [4] proposed the shift of MSP research from term generation to module generation. They have investigated the efficiency problem of indirect accesses in ML-style modules, and shown that the problem may be solved in a hypothetical extension with module generation. They also showed that their solution can be converted to a program written in the existing MetaOCaml, and at the end of the paper, they have questioned whether there exist compelling examples which really need such an extension.

In this paper, we investigate the same problem as theirs, and give a solution from a different angle. Specifically, we propose a lightweight extension of (a subset of) MetaOCaml where we can naturally and smoothly express manipulation of the code of a module, including splicing module components in another expression. This extension is arguably useful to express solutions for several inefficiency problems caused by module abstractions. Our extension is lightweight in the sense that we can translate away the extended functionality into the existing MetaOCaml.

The rest of this paper is organized as follows: §2 introduces the motivating example and the problem to be addressed in this paper, and the earlier work on the same problem. We show our solution to the motivating example in §3 and then

[1] Strictly speaking, type safety in these languages is guaranteed only for a certain sublanguage of these languages, typically a language without computational effects.

PEPM'18, January 8–9, 2018, Los Angeles, CA, USA

ACM ISBN 978-1-4503-5587-2/18/01...$15.00
https://doi.org/10.1145/3162072

```
module type EQ = sig
  type t
  val eq: t → t → bool
end
module type SET = sig
  type elt
  type set
  val member: elt → set → bool
end
module MakeSet (Eq: EQ)
 : SET with type t = Eq.t =
struct
  type elt = Eq.t
  type set = Eq.t list
  let rec member elt = function
  | [] → false
  | elt' :: set' →
    Eq.eq elt elt'
     || member elt set'
end
module IntSet = MakeSet (struct
  type t = int
  let eq = (=)
end)
```

Figure 1. MakeSet Functor (no code generation)

introduce our language which allows module generation in §4. §5 shows an implementation of our language through a translation to MetaOCaml, and the result of performance measurement is shown in §6. §7 gives concluding remark and future work.

2 Motivating Example

The module system in ML-like languages provides a powerful abstraction to structure a large program. It has been an active target of scientific research, and has found many interesting extensions with compelling applications such as first-class modules, modular implicits [14], and tagless final embedding [1]. On the practical side, the MirageOS[2] is one of the most successful library operating systems which uses OCaml modules to implement operating system drivers.

A big problem of module abstraction is performance penalty in functor[3] applications. Inoue et al. addressed this problem and gave a solution for a few examples, using program generation techniques. We shall illustrate the problem and their solutions below using the same example as theirs.

Inoue et al.'s leading example is a module which represents a set, shown in Fig. 1. In the code, the first nine lines define types for modules[4]. A module of type `EQ` consists of a concrete type for `t` and an implementation of `eq` whose type is `t → t → bool`. A module of type `SET` consists of two types `elt` (for elements) and `set` (for the set of elements), and a function

[2]https://mirage.io

[3]In ML, a *functor* is a function from modules to a module.

[4]In ML, the type of modules is called a *signature*.

```
module type EQ_CODE = sig
  type code t
  val eq: t code → t code → bool code
end
module type SET = (* the same as before *)
module MakeSetGen (Eq: EQ_CODE)
 : (SET with type t = Eq.t) code =
 <struct
   type elt = ~(Eq.t)
   type set = elt list
   let rec member elt = function
   | [] → false
   | elt' :: set' →
     ~(Eq.eq <elt> <elt'>) || member elt
         set'
 end>
module IntSet = MakeSetGen (struct
  type code t = <int>
  let eq (x : int code) y = <(~x) = (~y)>
end)
```

Figure 2. Inoue et al.'s solution (slightly modified)

`member` (for the membership function). `MakeSet` is a functor which is given a module of type `EQ` (which specifies the type of elements of the set) and returns a module of type `SET`. What MakeSet actually does is to implement the finite set as a list, and provides an implementation of the membership function. The last four lines apply the functor `MakeSet` to a module of type `EQ`, which has `int` and the equality function[5] over `int` as its components. By this application we obtain a concrete module of type `SET` whose element is of type `int`.

Although the above usage of modules provides an elegant, modular framework for introducing sets, it has a serious performance penalty compared with a monolithic implementation of the module `IntSet` which does not use functors. The problem of the code in Fig. 1 lies in the phrase `Eq.eq elt elt'` where Eq.eq is a reference to the function (=) (an indirect access to the compiled code of the function). Every time this phrase is evaluated, the actual content of Eq.eq is dereferenced, and this overhead is not negligible if the module component is dereferenced repeatedly. Inoue et al. have observed that the abstraction overhead can be eliminated by MetaML-style program generation for modules.

Fig. 2 shows their solution for the above problem written in a hypothetical extension of MetaOCaml. Since this code uses MetaOCaml, an extension of OCaml with the functionality of quasi-quotation for terms, we explain its basic operators first. `<e>` is the code for the term `e`, for instance `<3 + 5>` is a code for the term `3 + 5`. The term `~e` is used for splicing. for instance, the code `<2 + ~x * 4>` will evaluate to `<2 + (3 + 5) * 4>` if the value of `x` is `<3 + 5>`. The type of `<3 + 5>` is `int code`, not `int`. Although not included in our example,

[5] In OCaml the notation (=) represents the prefix version of the function =, thus (=) x y is equivalent to x = y.

the term `run e` is used for compiling and running code. For instance, the code `run <3 + 5>` will evaluate to `6`. The type of `run <3 + 5>` is `int`.

The program in Fig. 2 uses these annotations in two ways. The first, traditional way is the usage in `let eq (x : int code) y = <(~x) = (~y)>` where the annotations are used for terms. For instance, the term `eq <2+3> <3+1>` evaluates to the code value `<(2+3) = (3+1)>`. The second, untraditional way is the usage in `module MakeSetGen (Eq: EQ_CODE) = < struct ... end>` where a concrete module appears inside brackets, namely we use code of a module. Since the current implementation of MetaOCaml does not allow code of a module, the above program can only be written in a hypothetical extension. If there were such an extended language, the program in Fig. 2 can solve the problem of indirect access; the functor `MakeSetGen` receives an argument of type `EQ_CODE`, which is intuitively a code value of a module of type `EQ`, and returns a code value of a module where the actual equality function (provided by the argument) is spliced at the place of `Eq.eq` in the code `~(Eq.eq <elt> <elt'>)`. This splicing is done at program-generation time, and the result of program generation is a code value of a module. By running it, we obtain a module that does not suffer from the performance problem.

Inoue et al.'s paper left us the following two questions: (1) They found another solution obtained by translating the above program to a program in the standard MetaOCaml. They then asked if their hypothetical extension is really needed in realistic (and compelling) applications. (2) The program in Fig.2 has a typing problem which is not fully settled in their paper. Is there any type-sound language to support module generation? The present paper tries to answer these two questions.

3 Our Solution

We propose an extension of (core) MetaOCaml which allows generation and manipulation of code of a module. Our extension is intentionally very small so that MetaOCaml programmers can easily understand it. Fig. 3 shows a solution to the leading example in the previous section written in our language, which we shall explain below.

The solution uses first-class modules (standard in OCaml and MetaOCaml) plus three new operators \$, % and **run_module** for manipulating code of a module. First-class modules are standard since OCaml 3.12, which allow manipulation of modules as first-class values that can be passed to and returned from functions. A module m is turned to an expression by `(module m : M)` where M is the type of m, and the expression e is turned back to a module by `(val e)`. As modules are turned to expressions, the functor `MakeSet` in Fig. 1 can be represented by the function `makeSet`.

The \$ operator converts code of a module to a module of code. Consider the program phrase `$eq.t`. Since `eq` has

```
let makeSet (type a)
  (eq: (module EQ with type t = a) code) =
  <(module struct
      type elt = %($eq.t)
      type set = %($eq.t) list
      let rec member elt = function
      | [] → false
      | elt' :: set' →
        ~($eq.eq) elt elt'
          || member elt set'
    end : SET with type elt = a) >
module IntSet = run_module
 (val makeSet <(module struct
      type t = int
      let eq = (=)
    end: EQ with type t = int)>)
```

Figure 3. Our Solution for MakeSet

type `(module EQ) code`,[6] it refers to code of a module of type EQ. It is not possible to extract its components such as the type `t` and the value `eq`, since MetaOCaml does not allow destruction of code values in any way. Hence we need the new operator \$ to covert `eq` to a module of code, namely, a module whose value component is code. Then we can extract each component by simply applying the dot notation to `$eq`, and `$eq.eq` refers to the `eq` component. (Note that `$eq.eq` is parsed to `($eq).eq`.)

We think that the existence of the \$ operator in our language is harmless by the following reasons. First, we have concrete semantics for our language, via the translation to core MetaOCaml. Second, the \$ operator already exists in our intended semantics. Let us assume that our target language does not have any computational effects (which is in fact true in our current setting), and we interpret the type `A code` as `unit → A`, which should be one possible interpretation. Then the type `(A * B) code` is isomorphic to the type `(A code) * (B code)`, and similarly the type `{x : A; y : B} code` is isomorphic to the type `{x : A code; y : B code}`. Since modules in MetaOCaml are internally represented as records[7], it is natural to expect that the type for code of a module is isomorphic to the type for a module of code, namely, a module whose components have code types.[8] The \$ operator is a syntactic operator for one direction of this "isomorphism". Admittedly, this naïve argument is insufficient as a mathematical justification, and we think that it is sufficient as an intuitive guidance for our language design.

The % operator is the typed version of Cross-Stage Persistence (CSP). MetaOCaml allows a present-stage value such as `fun x → x * x` to be embedded in code (a future-stage

[6] We ignore the sharing constraint "with type t = a" in this discussion, since it is not quite relevant.

[7] This holds if we ignore dependency between components of modules.

[8] Similarly we can consider "isomorphism" between code of X and X of code for X=tuple, list, and other similar data structures.

```
(module (sig              module (sig
   type t = int             type t = int
   val v1: t                val v1: t code
   val v2: t → t            val v2: (t → t) code
 end)) code               end)
```

Figure 4. Types for code of a module and a module of code values

value), by which the value goes across the stage boundary. (Note the difference from splicing which allows us to embed code such as `<fun x → x * x>` in another code.) Taha and Nielesen [13] used the notation `%e` for CSPing the value of `e`, and we borrow it to denote CSP for types.[9] The phrase `type set = %($eq.t) list` in Fig. 3 means that, if `$eq.t` evaluates to `int * bool`, then the type `set` is `(int * bool) list`. If the `%` operator is omitted, then the phrase `$eq.t` is not evaluated since it appears inside brackets. The `%` operator lets the phrase be evaluated and then replaced by the result of evaluation. The novelty of our `%` operator is to allow CSP for types. Note that we have no other operators that manipulate types, in particular, we do not allow code of types in our language. This is crucial in our design which differentiate our language from Inoue et al.'s.

The **run_module** operator runs code of a module and it is similar to the **run** primitive in MetaOCaml, which works for code of a term.

We claim that the program in Fig. 3 is natural and easy to write and understand. First it perfectly fits the MetaML-style program generation framework. In this framework, given an ordinary program, one only has to add annotations such as `<...>` and `~(...)` to appropriate places in the program, based on binding-time analysis (the process is called *staging*). In our language, one need to add % and $ in addition to these annotations, but still the resulting program after staging is quite similar to the original program.

Besides the basics of MetaOCaml and ML-style modules, all we need to understand our language are the first-class modules and the new operators $, % and **run_module**. but as the program in Fig. 3 shows, they are really lightweight. The first one simply converts code of a module to a module of code, and the second embeds the type dereferenced by a type component of a module and it is similar to the implicit CSP for values in MetaOCaml. The last one is similar to the run primitive.

4 Our Language

Our language is an extension of core MetaOCaml which does not include computational effects such as references and exceptions, but includes first-class modules. We omit the explanation of the standard part; see Harper and Lillibridge [3] and Leroy [8] for module calculi.

[9] We could have designed our language to elide the operator % in the source code just like MetaOCaml, but we made it an explicit operation to make the point clear.

$$
\begin{aligned}
m &::= \textbf{struct}\ c;\ c\ \cdots ; c\ \textbf{end} \mid \textbf{val}\ g\\
c &::= \textbf{val}\ x : \sigma = e \mid \textbf{type}\ t = \sigma\\
e &::= x \mid p(e, \cdots, e) \mid x.x \mid \$x.x \mid \textbf{fun}\ (x : \sigma) \to e\\
&\quad \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid e\ @\ e \mid \texttt{<}e\texttt{>} \mid !\,e \mid \sim e\\
g &::= e \mid \textbf{fun}\ (x : \tau) \to g \mid \textbf{let}\ x = g\ \textbf{in}\ g \mid g\ @\ g\\
&\quad \mid (\textbf{module}\ m : M) \mid \texttt{<}g\texttt{>} \mid \textbf{run\_module}\ g
\end{aligned}
$$

Figure 5. Syntax for terms

Fig. 5 defines the syntax of modules (m), module components (c), simple expressions (e), and general expressions (g). We distinguish simple expressions from general expressions to avoid nested modules (a module which contains another module as its component) for simplicity. Simple expressions are standard lambda expressions such as variables and primitive operations $p(e, \cdots, e)$, selection for module component $x.x$, expressions for staging ($<e>$, $!e$ and $\sim e$), or the new expression $\$x.x$ introduced in this work. A general expression may contain a module expression and related expressions. The expression (**module** $m : M$) is an expression converted from a module m where the type of module M is made explicit. The expression $<g>$ may be code of a module. We also have an expression **run_module** g for running a module. The example in Fig. 3 can be written as a general expression in this syntax.

The type system of our language is based on Davies' $\lambda\bigcirc$[2] which allows manipulation of open code and no stage errors may happen for typeable terms. Although $\lambda\bigcirc$ does not prevent open code from being executed (the scope extrusion problem [9, 13]), the situation is not worse than the current MetaOCaml and Scala LMS. It is left for future work to make our language to be scope safe.

For simplicity we restrict the number of stages to two where the stage 0 is the present stage and the stage 1 is the next (future) stage. Each typing judgment is associated with its stage such as $E \vdash^i e : \sigma$ for $i = 0, 1$. Due to lack of space we cannot list the typing rules. Instead, we show a few examples of typing rules and a typing derivation. Let the types of Fig. 4 be M_1 (left) and M_2 (right). Then M_1 is the type for code of a module and M_2 is the type for a module consisting of a type and code values. Then, the $ operator has the following typing rule where the superscript 0 indicates the stage:

$$\frac{(x\ :\ M_1)^0 \in E}{E\ \vdash^0\ \$x\ :\ M_2}$$

Namely, the $ operator merely converts a variable[10] of type M_1 to an expression of type M_2. Using the above rule, we

[10] We have restricted the syntax so that the argument of $ must be a variable.

can derive the type for each component of the module, for instance, the type of the component `v2` is derived as follows:

$$\frac{E \ \vdash^0 \ \$x \ : \ M_2}{E \ \vdash^0 \ \$x.v2 : (\$x.t \longrightarrow \$x.t) \ \textbf{code}}$$

The typing rule of our % operator is similar to the one for CSP in MetaOCaml:

$$\frac{E \ \vdash^0 \ \sigma \ :: \ \star}{E \ \vdash^1 \ \% \, \sigma :: \star}$$

where $\star$ is the (unique) kind, namely, $\sigma :: \star$ means σ is a well-formed type. Using this rule, we can derive well-formedness of a type component as follows:

$$\cfrac{\cfrac{\cfrac{E \ \vdash^0 \ \$x : M_2}{E \ \vdash^0 \ \$x.t :: \star}}{E \ \vdash^1 \ \%(\$x.t) :: \star}}{E \ \vdash^1 \ (\textbf{type} \ t = \%(\$x.t)) \ \textbf{wf}}$$

Since the last judgment of the above derivation has the stage 1, **type** $t = \%(\$x.t)$ is used in the future stage (as a type component of code of a module). The type component **type** $elt = \%(\$eq.t)$ in Fig. 3 can be typed similarly.

The typing rule for the **run_module** operator is similar to the run primitive in MetaOCaml:

$$\frac{E \ \vdash^0 \ g \ : \ (\textbf{module} \ M) \ \textbf{code}}{E \ \vdash^0 \ \textbf{run\_module} \ g \ : \ \textbf{module} \ M}$$

Intuitively, it receives a code value of a module, and returns a compiled code of the module.

In summary, our type system assigns types to terms based on the (intuitive) isomorphism given in Fig. 4. The rest of the rules in our type system is standard and omitted.

5 Implementation by Translation

We have implemented our language by translating the added functionality away, namely we translate a program in our language to a MetaOCaml program. Its main role is to eliminate code of a module `<(module m : M)>` , which is not allowed in MetaOCaml.

The translation is parameterized by stages, namely, we have the present-stage translation $[\![\cdot]\!]^0$ and the future-stage one $[\![\cdot]\!]^1$. The former is homomorphic over constructors except $[\![<e>]\!]^0 := [\![e]\!]^1$. The future-stage translation $[\![\cdot]\!]^1$ is used for translating expressions inside brackets. A module and its component are translated at the stage 1 as follows:

$$\begin{aligned}
[\![(\textbf{module} \ m : M)]\!]^1 &:= (\textbf{module} \ [\![m]\!]^1 : [\![M]\!]^1) \\
[\![\textbf{type} \ t :: \star = \sigma]\!]^1 &:= \textbf{type} \ t :: \star = [\![\sigma]\!]^1 \\
[\![\textbf{val} \ x : \sigma = e]\!]^1 &:= \textbf{val} \ x : [\![\sigma]\!]^1 \textbf{code} = <[\![e]\!]^1>
\end{aligned}$$

A value component (third line) is converted to a code value, while a type component (second line) is kept intact. The new primitives $ and % are simply dropped by the translation: $[\![\$x_1.x_2]\!]^1 := x_1.x_2$ and $[\![\% \ \sigma]\!]^1 := [\![\sigma]\!]^1$. Fig. 6 shows the result of translating the example of `MakeSet` in Fig. 3.

```
module type EQ_CODE = sig
  type t
  val eq: (t → t → bool) code
end
module type SET_CODE = sig
  type elt
  type set
  val member: (elt → set → bool) code
end
let makeSet (type a)
  (eq: (module EQ_CODE with type t = a)) =
  (module struct
    module Eq = (val eq)
    type elt = Eq.t
    type set = Eq.t list
    let member =  <
      let rec member elt' = function
      | [] → false
      | (elt :: set) →  ~(Eq.eq) elt' elt
          || member elt' set
      in member >
  end : SET_CODE with type elt = a)
```

Figure 6. Result of the translation of MakeSet in Fig. 3

Translating the **run_module** operator away is not straightforward, since we need to “propagate” the **run**-primitive to each module component. Nevertheless we can handle this case by making the translation dependent on the type of the source term. We explain it by an example. Suppose g has the following type.

$$\begin{array}{l}
(\textbf{module} \ (\textbf{sig} \\
\quad \textbf{type} \ \mathrm{t} \\
\quad \textbf{val} \ \mathrm{v1} : \ \sigma_1 \\
\quad \textbf{val} \ \mathrm{v2} : \ \sigma_2 \\
\ \textbf{end})) \ \textbf{code}
\end{array}$$

Then we translate (**run_module** g) as follows:

$$\begin{array}{l}
[\![\textbf{run_module} \ \mathrm{g}]\!]^0 = (\textbf{module struct} \\
\qquad \textbf{module} \ \mathrm{G} = (\textbf{val} \ [\![\mathrm{g}]\!]^0) \\
\qquad \textbf{type} \ \mathrm{t} = \mathrm{G.t} \\
\qquad \textbf{let} \ \mathrm{v1} = \textbf{run} \ \mathrm{G.v1} \\
\qquad \textbf{let} \ \mathrm{v2} = \textbf{run} \ \mathrm{G.v2} \\
\quad \textbf{end})
\end{array}$$

where **run** is a primitive for executing code in MetaOCaml, and **val** for turning an expression to a module. We use a nested module (a module which has a module component) to make the result of the translation be a single module. Note that each value component of the outer module has the **run** primitive. Clearly, the above translation cannot be defined unless we use the type information of g.

The translation is not quite correct for modules whose components have dependency. Consider the following code snippet:

```
let m = <(module struct
   let v1 = 10 + 20 + 30
   let v2 = v1 + v1
end)>
```

By translating this program, we would get:

```
let m = (module struct
   let v1 = <10 + 20 + 30>
   let v2 = <v1 + v1>
end)
```

but then v2 has a free reference to v1 at the stage 1. To solve the problem, we refine the translation by inserting let expressions in the following way:

```
let m = (module struct
   let v1 = <10 + 20 + 30>
   let v2 = <let v1 = ~(v1) in
              v1 + v1>
end)
```

Now v2 has no free references and can be used in other code by splicing. Thus, we have resolved dependency via let-insertion. The above strategy fixes the problem of the naïve translation, yet it has a problem; generated code may become excessively large for some cases. It is our ongoing work to improve the size of generated code, which will be reported elsewhere.

We have implemented our language using the above translation. The performance of our implementation is shown in the next section.

6 Experiments and Performance

We have conducted an experiment for micro benchmarks using the implementation of our language. The case study shown in this section was taken from Suzuki et al.'s normalizer for language-integrated query (SQL-query language integrated with a functional language) [11]. They used the tagless-final embedding for domain-specific languages, and expressed each normalization step as a functor, and used a recursive functor to iterate normalization steps. Thus, functor applications are repeatedly used in their normalizer and their overhead is not negligible. We have implemented a simplified normalizer in our language, and measured its performance. Here we show its core part, and the complete code is shown in the first author's page[11].

```
module type S = sig
  type int_t
  type obs_t
  val int: int → int_t
  val add: int_t → int_t → int_t
  ...
end
let suppressAddZeroPE =
 fun (m: (module S with ..) code) →
 <(module struct
    type int_t = $m.int_t * bool
    type obs_t = int
    let int = fun n1 →
      (~($m.int) n1, n1 = 0)
    let add = fun n1 → fun n2 →
      match (n1, n2) with
      | (n1, b1), (n2, b2) →
        if (b1 && b2) then
          ( ~($m.int) 0, true)
        else
          ~($m.add) n1 n2
   ...
   end: S with type obs_t = int)>
let rec fix depth m =
  if depth <= 0 then  m
  else fix (depth - 1)
       (suppressAddZeroPE m)
```

[11]http://logic.cs.tsukuba.ac.jp/~takahisa/module-generation.html

The function suppressAddZeroPE realizes a program transformation for the zero-suppression optimization such as x+0 to x. It is given an expression of type (module S) code (which specifies the signature of the object language) and returns code of a module after performing the optimization. We apply the transformation iteratively to obtain the fully optimized form, hence we use a recursive function in the code. The number of iteration is given by the parameter depth. Note that this kind of control is easily implemented in our language, while it may be difficult for a fully-automatic static analyzer which would try to infinitely inline a module (or never inline it).

Our translator turns the above code into a MetaOCaml program (a code generator). Then we run it in MetaOCaml to obtain an OCaml program, and by executing it we obtain the final result. We have measured the execution time (the last step) of the above implementation, and that of a naïve implementation of the same program transformation which does not use code generation for modules. (We do not include the time for code generation in measurement.) The result is shown in Table 1 where the unit is second, and we run it on MacOS X 10.11.2, Memory 8GB, BER MetaOCaml N104 (OCaml 4.04.0), byte code compiler.

Table 1. Performance Measurement

depth	2	4	6	8	10
Naïve	0.0501	0.0933	0.1284	0.1692	0.2167
Ours	0.0064	0.0108	0.0159	0.0174	0.0232

The first row shows the value of the depth parameter (the number of iterations), and the second and the third show the performance of the naïve one and that of our implementation, resp. The result shows that our results run seven to eight times faster than the naïve implementation.

Clearly this is a biased example to our approach in that functor application (or function application) occurs a huge number of times. Also, the use of recursive functors in OCaml may have had severe performance penalty. Nevertheless, the result is encouraging to pursue the language for module generation.

7 Conclusion

We have proposed an extension of (core) MetaOCaml where one can write program generators that can manipulate and generate code of a module in the type-safe way. We believe that our extension naturally fits the style of MetaML-like multi-stage programming, thus allowing one to write module manipulation easily and naturally. We have shown that the `MakeSet` example and a simplified example of Suzuki et al.'s tagless-final program transformation are expressible in our language, and that the performance of generated modules is improved.

We briefly state future work. First, we put several restrictions to our language and eliminating these restrictions will be an interesting research topic. For instance, we allow only two stages, which means that we cannot write a code generator which generates yet another code generator. We also do not allow nested modules in the source language. Second, we sticked to the MetaML-style type-safe approach to program generation, and did not allow run-time generation and manipulation of code of types. Several authors including Inoue et al.[4] have already argued that allowing it may further improve expressivity of generators and performance of generated code. It is left for future investigation.

Acknowledgments

The authors would like to thank the program committee and anonymous reviewers of PEPM'18 for valuable comments and criticism. Special thanks go to Kohei Suenaga for numerous constructive comments on earlier versions of this paper. The second author is supported in part by the JSPS KAKENHI No. 15K12007.

References

[1] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.* 19, 5 (2009), 509–543. https://doi.org/10.1017/S0956796809007205

[2] Rowan Davies. 1996. A Temporal-Logic Approach to Binding-Time Analysis. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996.* 184–195. https://doi.org/10.1109/LICS.1996.561317

[3] Robert Harper and Mark Lillibridge. 1994. A Type-Theoretic Approach to Higher-Order Modules with Sharing. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994.* 123–137. https://doi.org/10.1145/174675.176927

[4] Jun Inoue, Oleg Kiselyov, and Yukiyoshi Kameyama. 2016. Staging Beyond Terms: Prospects and Challenges. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '16).* ACM, New York, NY, USA, 103–108. https://doi.org/10.1145/2847538.2847548

[5] Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml - System Description. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings (Lecture Notes in Computer Science)*, Michael Codish and Eijiro Sumii (Eds.), Vol. 8475. Springer, 86–102. https://doi.org/10.1007/978-3-319-07151-0_6

[6] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream fusion, to completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 285–299. http://dl.acm.org/citation.cfm?id=3009880

[7] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. 2014. Building Efficient Query Engines in a High-Level Language. *PVLDB* 7, 10 (2014), 853–864.

[8] Xavier Leroy. 2000. A Modular Module System. *J. Funct. Program.* 10, 3 (May 2000), 269–303. https://doi.org/10.1017/S0956796800003683

[9] Junpei Oishi and Yukiyoshi Kameyama. 2017. Staging with control: type-safe multi-stage programming with control operators. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2017, Vancouver, BC, Canada, October 23-24, 2017.* 29–40. https://doi.org/10.1145/3136040.3136049

[10] Tiark Rompf. 2016. Lightweight modular staging (LMS): generate all the things! (keynote). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2016, Amsterdam, The Netherlands, October 31 - November 1, 2016*, Bernd Fischer and Ina Schaefer (Eds.). ACM, 1. https://doi.org/10.1145/2993236.2993237

[11] Kenichi Suzuki, Oleg Kiselyov, and Yukiyoshi Kameyama. 2016. Finally, Safely-extensible and Efficient Language-integrated Query. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '16).* ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/2847538.2847542

[12] Walid Taha. 2003. A Gentle Introduction to Multi-stage Programming. In *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers.* 30–50. https://doi.org/10.1007/978-3-540-25935-0_3

[13] Walid Taha and Michael Florentin Nielsen. 2003. Environment Classifiers. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03).* ACM, New York, NY, USA, 26–37. https://doi.org/10.1145/604131.604134

[14] Leo White, Frédéric Bour, and Jeremy Yallop. 2014. Modular implicits. In *Proceedings ML Family/OCaml Users and Developers workshops, ML/OCaml 2014, Gothenburg, Sweden, September 4-5, 2014.* 22–63. https://doi.org/10.4204/EPTCS.198.2

[15] Jeremy Yallop. 2017. Staged generic programming. *PACMPL* 1, ICFP (2017), 29:1–29:29. https://doi.org/10.1145/3110273