Module Generation without Regret

Yuhi Sato
Department of Computer Science
University of Tsukuba
Tsukuba, Japan
yuhi@logic.cs.tsukuba.ac.jp

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

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

Abstract

Modules are an indispensable mechanism for providing abstraction to programming languages. To reduce the abstraction overhead in the usage of modules, Watanabe et al. proposed a language for generating and manipulating code of modules, and implemented it via a translation to plain MetaOCaml. Unfortunately, their solution has a serious problem of code explosion if functors are repeatedly applied to modules. Another problem in their solution is that it does not allow nested modules.

This paper proposes a refined translation for a two-stage typed language with module generation where nested modules are allowed. Our translation does not suffer from the code-duplication problem. The key idea is to use the genlet operator in latest MetaOCaml, which performs let insertion at the code-generation time to allow sharing of code fragments. To our knowledge, our work is the first to apply genlet to code generation for modules. We conduct an experiment using a microbenchmark, and the result shows that our method is effective to reduce the size of generated code that would have been exponentially large.

CCS Concepts • **Software and its engineering → General programming languages**.

Keywords Program Generation, Modules, Type Safety, Program Transformation

ACM Reference Format:
Yuhi Sato, Yukiyoshi Kameyama, and Takahisa Watanabe. 2020. Module Generation without Regret. In *Proceedings of the 2020 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '20), January 20, 2020, New Orleans, LA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3372884.3373160

PEPM '20, January 20, 2020, New Orleans, LA, USA

ACM ISBN 978-1-4503-7096-7/20/01…$15.00
https://doi.org/10.1145/3372884.3373160

1 Introduction

Program generation breaks the trade-off between productivity and performance, and has been studied intensively [14]. Multi-Stage Programming (MSP) languages such as MetaML provide a way to generate programs against runtime parameters while static safety assurance is guaranteed. Unfortunately, it is difficult to guarantee its safety beyond term generation, and generating a module is not allowed in MetaOCaml, a multi-stage extension of OCaml.

A module system in ML is highly valuable for providing high-level abstraction as well as from developing practical applications. Large programs can be developed efficiently using modules, as they allow us to build each software component independently, and to compose them in a safe way to achieve reusability and maintainability. On the practical side, MirageOS[1] is a successful example of large-scale applications which use a number of modules. In the implementation of MirageOS, OS components such as device drivers and protocols are implemented as independent libraries, which contains thousands of modules. On the research side, interesting extensions using modules have been proposed: modular implicits [18], extensible language-integrated query [15] and tagless-final embedding [2].

Inoue et al. [7] were the first to propose a language extension for generating code of a module in the MSP style. They investigated the abstraction overhead in ML-style modules, and pointed out that the problem may be solved in a hypothetical extension with module generation. Watanabe et al. [17] proposed a language $\lambda^{<M>}$ for generating the code of a module, and implemented the language. They have also conducted an experiment to show that the abstraction overhead in modules can be reduced. Unfortunately, their approach has another problem; code generated by their method can become so large that it may not compile.

In this paper, we proposed a language and its implementation that solve the problems left open by Watanabe et al. First, we introduce a new translation from $\lambda^{<M>}$ to MetaOCaml to solve the code-explosion problem in Watanabe et al.'s approach. Our key idea is to use the genlet primitive in MetaOCaml for let-insertion, to avoid code duplication. Second, we show that our language allows nested modules in code generation. Third, we briefly mention the type system

[1] https://mirage.io

of our language. Furthermore, we discuss several interesting topics on modules such as the code of a functor.

The rest of this paper is organized as follows: §2 shows the benefits of modular programming and describes the previous work for module generation and its problems. We introduce our solution in §3. §4 defines our source language that allows module generation. §5 defines the translation from the source language to MetaOCaml and mentions the type system. The experiments on microbenchmark and their results are shown in §6. We discuss our translation in §7 and describe the related work in §8. §9 give conclusion and future work.

2 Motivating Example

This section introduces the motivating example and illustrates the benefits of modular programming. It also explains a summary of previous studies on module generation and their problems.

2.1 Modular Approach

The module system allows us to build clean, maintainable and large-scale applications. Our motivating example is a simple implementation of an echo server:

```
module Main (Log:LOG) (Tcp:TCP) = struct
  module Server = MakeServer (Tcp)

  let echo msg =
    Log.info ("[echo] Received: "^msg);
    Log.info ("[echo] Send: "^msg);
    msg

  let start =
    let tcp = Tcp.create "localhost" 7777 in
    Server.start tcp echo
end
```

This program creates a new network service which accepts a connection on the TCP port 7777, receives a message, and echoes the message back to the client and outputs logs. We briefly explain an ML module. A module (*structure*) consists of *components* that are declarations of values, types, and nested modules, and a parameterized module is called a *functor*. The type of a module is called a *signature*, which is an interface to the module. In the above functor `Main`, the implementations of modules `Log` and `Tcp` are abstracted by signatures `LOG` and `TCP` respectively. The functor `Main` makes the nested module `Server` which is a server library on the given module `Tcp`, and passes the component (function) `echo` to the function `Server.start` as a callback. To run the echo server, we apply the functor to two modules that implement the signatures, create a new module `M`, and call its `start` function:

```
let module M = Main (PrintLogger) (Tcp) in
  M.start ;;
```

There are several reasons to use modules and functors to build applications in this way. First, it helps us to write reusable, testable and maintainable programs. The functor provides a common implementation. The signature supports a loosely coupled implementation which is independent of the module implementation. Therefore, we can easily swap the implementation. For example, to change the destination of logs from the standard output to a file, we create a new module `FileLogger` that implements the signature `LOG`, and apply the `Main`. Also, a mockup version of the module `Tcp` that does not actually communicate is useful for unit testing.

Second, large-scale applications may be built easily with modules and functors. By dividing a large program into smaller modules, we can ask an expert to develop each module independently of the others. To assemble these modules, nested modules are essential. In the real world, the website of MirageOS is built with up to 10 functor applications, and the Mirage repository hosts hundreds of libraries which use modules [13].

2.2 Module Generation

Modular programs sometimes suffer from performance penalty; modules obtained by functor applications may have runtime overhead due to indirection. In the above example, the function `Log.info` is called via an indirect access. Also, the function `Server.start` is called with repeated indirections because the functor `MakeServer` refers to the module `Tcp`. In general, programs including many repeated functor applications possibly have serious performance penalty.

Watanabe et al. [17] gave the language $\lambda^{<M>}$, an extension of MetaOCaml, that allows generation of code of a module. They used first-class modules for generating and manipulating code of a module, that can be passed to and returned from functions.

Figure 1 shows the functor `Main` written in $\lambda^{<M>}$. For the sake of explanation, the functor `Main` is simplified and has the function `echo` only.

Let us explain MetaOCaml very briefly. MetaOCaml provides three multi-stage operators $\langle\rangle$, $\sim$, and **run** . Brackets $\langle e\rangle$ generate code of the expression e and escape $\sim e$ exempts the expression e from the brackets. When the type of e is τ, the type of $\langle e\rangle$ is τ **code**. The expression **run** e compiles the code e and executes it. In Figure 1, the `main` functor (function) receives code of a module and returns a new code of a module. `$` is a new operator added to $\lambda^{<M>}$, which extracts code of a component from code of a module. `$log.info` means that extract the code of the function `info` from the code of the module `log`. Thus, the `~($log.info)` part is replaced with the code of the function `log.info`, and the abstraction overhead is eliminated.

Watanabe et al. implemented $\lambda^{<M>}$ by a translation to MetaOCaml. Its main role is to eliminate a module within brackets, which is not allowed in MetaOCaml. Their idea is to turn the code of a module into a module containing code. Figure 2 shows the functor `Main` translated from Figure 1. The type of the function `echo` is translated from `string`

```
module type MAIN = sig
  val echo : string -> string
end
let main = fun (log: (module LOG) code) ->
  <(module struct
    let echo = fun msg ->
      ~($log.info)("[echo] Received: "^msg);
      ~($log.info)("[echo] Send: "^msg);
      msg
  end : MAIN)>
```

Figure 1. Main functor written in $\lambda^{<M>}$

```
module type MAIN' = sig
  val echo : (string -> string) code
end
let main = fun (log: (module LOG)) ->
  (module struct
    module Log = (val log)
    let echo = <fun msg ->
      ~(log.info) ("[echo] Received: "^msg);
      ~(log.info) ("[echo] Send: "^msg);
      msg>
  end : MAIN')
```

Figure 2. A MetaOCaml program translated from Figure 1

`-> string` to `(string -> string) code`, and the return type of the function `main` is translated from `(module MAIN) code` to `(module MAIN')`. The operators added to $\lambda^{<M>}$ are eliminated by the translation, and we can run the resulting code in Figure 2.

2.3 Code Explosion Problem

Unfortunately, Watanabe et al.'s translation has a serious problem in that the size of generated code may increase exponentially. An illustrative example is the module `printLogger` which implements the signature `LOG`. The code of the module `printLogger` written in $\lambda^{<M>}$ is shown below.

```
module type LOG = sig
  val warn : string -> unit
  val info : string -> unit
end
let printLogger = <(module struct
  let print = fun level -> fun msg ->
    let t = Unix.localtime (Unix.time ()) in
      Printf.printf "[%s] %d:%d:%d\n%s\n"
      level t.tm_hour t.tm_min t.tm_sec msg

  let warn = fun msg -> print "WARN" msg
  let info = fun msg -> print "INFO" msg
end : LOG)>
```

The module `printLogger` exposes two functions, `warn` and `info`, resp. to display the log level and the timestamp, resp. The function `print` is an auxiliary function for formatting the log. The above program is translated to the following one:

```
module type LOG = sig
  val warn : (string -> unit) code
  val info : (string -> unit) code
end
let printLogger = (module struct
  let print = <fun level -> fun msg ->
    let t = Unix.localtime (Unix.time ()) in
      Printf.printf "[%s] %d:%d:%d\n%s\n"
      level t.tm_hour t.tm_min t.tm_sec msg>

  let warn = <fun msg ->
    let print = fun level -> ... in
    print "WARN" msg>
  let info = <fun msg ->
    let print = fun level -> ... in
    let warn = print "WARN" msg in
    print "INFO" msg>
end : LOG)
```

The problem here is that the function `warn` is defined in the function `info` even though it is not used. To avoid free references in the result, Watanabe et al.'s translation inserts all let-binding up to the i-th function into the $i+1$-th binding. Hence, it inserts a total of $n \cdot (n-1)/2$ let-bindings where n is the value of components. The program of the function `Main.echo` is shown below, which is created by applying the functor `main` to the module `printLogger`.

```
# let module Main = (val main printLogger) in Main.echo;;
- : (string -> string) code = .<
(* echo *)
fun msg_23 ->
  (* info *)
  (fun msg_16 ->
      let print_20 level_17 msg_18 =
        let t_19 = Unix.localtime (Unix.time ()) in
        Stdlib.Printf.printf "[%s] %d:%d:%d\n%s\n"
        level_17 t_19.Unix.tm_hour t_19.Unix.tm_min t_19.
             Unix.tm_sec msg_18 in
      let warn_22 msg_21 = (print_20 "WARN") msg_21 in
      (print_20 "INFO") msg_16)
    ("[echo] Received: "^msg_23);
  (* info *)
  (fun msg_16 ->
      let print_20 level_17 msg_18 =
        let t_19 = Unix.localtime (Unix.time ()) in
        Stdlib.Printf.printf "[%s] %d:%d:%d\n%s\n"
        level_17 t_19.Unix.tm_hour t_19.Unix.tm_min t_19.
             Unix.tm_sec msg_18 in
      let warn_22 msg_21 = (print_20 "WARN") msg_21 in
      (print_20 "INFO") msg_16)
    ("[echo] Send: "^msg_23);
  msg_23>.
```

The problem in this code is that the function `info` is defined twice in the function `echo`. An ideal code would define a let-binding for the function `info` locally and dereference the let-bound variable twice.

In the worst case, the code size increases exponentially in the number of functor applications. [2] In the previous example, as a function is used via a module twice, the size of the generated code is (approximately) doubled. If the function is used twice in another functor, the size of the code would be (approximately) four times as large as the original one.

[2] Since Watanabe et al. used first-class modules to represent modules as expressions, functors are represented by normal functions over first-class modules.

We want to eliminate the overhead from large applications which use many functor applications, as the code-explosion problem becomes more serious. Generating such a huge code takes a lot of time and space, and may cause compilation failures. Compiler optimization is not useful, since the code is generated before compilation. The problems above are summarized as follows.

- The size of the translated code is proportional to the square of the number of components.
- Code duplication occurs when the same component is referenced multiple times from outside of the module.
- The size of the generated code is proportional to the exponential in the number of functor applications.

2.4 Other Problems

We found a few other problems in Watanabe et al.'s study.

First, their source language was too liberal to be translated to plain MetaOCaml which does not allow code of modules. Consider the following example:

```
<let x = 10 + 20 in
  (module struct let y = 1 end : S)> ;;
```

where S is an appropriate signature. It is translated to:

```
let x = 10 + 20 in
 (module struct let y = <1> end : S') ;;
```

The subexpression `10+20` is executed at the future stage in the first expression, while it is executed at the present stage in the second expression, violating the distinction of stages.

Second, their language did not allow nested modules as follows:

```
module M = struct
  module N = struct x = 10 end
  let y = N.x + N.x
end
```

where the module `N` is a nested module which may be referred to in the components of `M`. Nested modules are useful in expressing a certain class of programs, as described above.

3 Our Proposal

We introduce a refined translation for the language with module generation to solve the code-explosion problem described in the previous sections. Our translation performs dynamic let-insertion, which allows code fragments to be shared among different components of modules. In this section, we explain how the translation works using examples. The formal definition of our language and translation will be given in the next section.

Let-insertion is a well-known technique for code sharing in program transformation (partial evaluation, in particular) [3]. It can be implemented in various ways, and here we review two most relevant approaches for let-insertion.

3.1 Static let-Insertion by shift and reset

The first approach uses the delimited-control operators shift and reset [4], which are available in Scheme/Racket, SML, OCaml, Scala, and other modern programming languages. In OCaml and MetaOCaml, they are implemented as an external library [9]. In this approach, a let expression is packaged with shift, and the destination for let-insertion is marked by reset. The let expression is inserted at run time. Note that the destination of let-insertion is determined statically in this approach.

The let-insertion technique via shift and reset has been studied in program generation [8]. Unfortunately, the technique is insufficient to solve the problem in Watanabe et al.'s translation. Since the translator does not know (before code generation) how many times a functor is applied to modules, it is hard to find the optimal destination for let-insertion statically.

Let us investigate why a static let-insertion does not solve the code-duplication problem. Consider the following program with a first-class module:

```
let mcod = <(module struct
  let x1 = 10 + 20
  let x2 = x1 + x1
end : S)> ;;
```

where S is a suitable signature. Watanabe et al.'s translation removes the code of modules, and the above code is translated to:

```
let mcod0 = (module struct
  let x1 = <10 + 20>
  let x2 = <~x1 + ~x1>
end : S') ;;
```

But the component `x2` alone does not make sense because of free occurrences of `x1`, and we need to supply the value of `x1` when we use `x2`. Instead of naively inlining the code for `x1` to get `(10+20)+(10+20)`, we insert a let expression to obtain:

```
let mcod0 = (module struct
  let x1 = <10 + 20>
  let x2 = <let t = 10 + 20 in t + t>
end : S') ;;
```

which is a duplication-free code. So far, so good.

As the next step, we apply the following functor `fnctr` to `macd0` (Watanabe et al. use the first-class modules, hence, a functor becomes a normal function):

```
let fnctr mcod = (module struct
  let x = <~mcod.x1 + ~mcod.x2>
end : T) ;;
fnctr mcod0 ;;
```

For the sake of simplicity, the unpacking of module `mcod` is omitted. The result of the last line is the following module:

```
(module struct
  let x = <(10+20) + (let t = 10+20 in t+t)>
end : T)
```

The final result still contains the code `10+20` twice, which shows the static let-insertion does not completely solve the problem. An ideal result would be the following.

```
(module struct
  let x = <let t = 10 + 20 in t + (t + t)>
end : T)
```

We can expect that the last result can be obtained by let-insertion, but the destination of let-insertion is the outermost position of nested functor applications. In general, we may want to apply functors to modules multiple times, and the ideal destination may be quite distant from the original position of a let expression.

3.2 Dynamic let-Insertion by genlet

The second way uses the genlet primitive [11] in MetaOCaml[3] which performs let-insertion. Unlike let-insertion via shift and reset, let-insertion by genlet works in code generators only. The notable point in genlet is that a program need not specify the destination of let-insertion, which is determined dynamically when the code is actually generated.

Let us consider the following example using genlet:

```
let x = genlet <10 + 20> in <~x + ~x>
```

The genlet primitive is a normal function, which generates a fresh future-stage variable, a let binding that binds it to the argument of genlet, and returns the code that refers to this variable. The generated let binding is inserted somewhere in the code, which is decided dynamically. An intermediate term in this execution is the following.

```
<let t = 10 + 20 in
   ~(let x = <t> in <~x + ~x>)>
```

which evaluates to the code below.

```
<let t = 10 + 20 in t + t>
```

The resulting code has no duplicated occurrences of `10 + 20`.

The destination of let-insertion by genlet is the outermost location that causes no scope-extrusion problem, namely, free variables in the argument of genlet should not go beyond their binders. In summary, genlet is useful to avoid code duplication in program generation.

The next question is whether genlet is useful for module generation, and how we can solve the code-duplication problem with modules.

Actually, our solution is very simple; for each value component in a module, we insert the genlet primitive at the topmost position of the right-hand side of a value component, and that's all. For instance, we rewrite the previous module `mcod0` to the following one:

```
let mcod1 = (module struct
  let x1 = genlet <10 + 20>
  let x2 = genlet <~x1 + ~x1>
end : S') ;;
```

[3]Available in BER MetaOCaml version N107 and later.

where two occurrences of `genlet` are introduced. Other parts of the program are kept intact.

Although simple, the reason why our solution works is rather complicated. Let us consider the execution of `mcod1` alone. The right-hand side of each value component of a module is evaluated one by one, and the function `genlet` is called twice. For the `x1` component, we get `<let t1 = 10 + 20 in t1>` as its value. The result of the execution of the `x2` component is rather unexpected, as it returns `<let t1 = 10 + 20 in let t2 = t1 + t1 in = t2>`. This is quite different from the result of a simple-minded computation for `x2`, which is `<let t2 = (let t1 = 10 + 20 in t1) + (let t1 = 10 + 20 in t1) in t2>`.

The reason why we got non-duplicating code for `x2` is somewhat complicated. [4] For the `x1` component, we get `<let t1 = 10 + 20 in t1>` as its value, which is not surprising. When we evaluate `genlet <e>`, we do not immediately get `<let t = e in ...t>`; rather, it returns an internal data structure (a triple) consisting of a set of free variables, the body `<e>`, and a list of let bindings to be inserted in future. In other words, `genlet <e>` is evaluated only partially and the let-insertion is delayed, similar to lazy evaluation. When we retrieve the value of the triple at the top level (for instance, the value is printed), let bindings in this triple are inserted at the topmost positions which do not cause the scope-extrusion problem. Coming back to the evaluation of the term `genlet <~x1 + ~x1>`, the value of `x1` is a triple which contains potential dynamic let-insertion. Hence, there are two nested dynamic let-insertion, and its result has nested let-bindings such as `<let t1 = 10 + 20 in let t2 = t1 + t1 in ...>`.

Our finding in this paper is the above machinery of genlet works as well in the presence of modules and functors. To see it, we consider the evaluation of the term `fnctr mcod1`, which simulates a functor application using first-class modules. When we evaluate the term, again the dynamic let-insertion triggered by genlet in `mcod1` is delayed until the result of the whole term is printed. When we print it, dynamic let-insertion by two genlet is actually performed, and we get the following ideal code as nested let bindings:

```
(module struct
  let x = <let t1 = 10 + 20 in
            let t2 = t1 + t1 in t1 + t2>
end : T)
```

Since let bindings for `t1` and `t2` are nested, it is clear that let insertion was performed after the evaluation of `mcod1`.

This feature of genlet has been considered useful in code generation, but as far as we know, it has not been studied whether genlet can work beyond module boundaries, until the work presented in this paper.

[4]Our explanation here is essentially due to Kiselyov's explanation for genlet, available from the BER MetaOCaml repository on GitHub.

4 Source Language

Our source language is a two-stage programming language which is an extension of core MetaOCaml. It includes standard lambda-calculus with primitive operators, let expression, and multi-stage operators for code generation. In this work, we confine ourselves to a minimal language to express our results. We assume that no name collision occurs among the components of modules and signatures. Extending our language to a more realistic one is left for future work.

Figure 3 defines the syntax of terms where the index i is either 0 (present stage) or 1 (future stage). We use metavariables m^i for stage-i modules (structures), s^i for sequences of stage-i components, c^i for stage-i components of modules, e for expressions, e^i for stage-i expressions, e^i_m for stage-i first-class modules, x for variables, X for module names, and t for types. A module m^i is either a module name X, a structure, or an unpacking expression ($\mathbf{val}\ e^i_m$). For convenience, we slightly extend the unpacking operator in OCaml so that $(\mathbf{val}\ e^i_m).x$ is a valid syntax, but it can be easily resolved by introducing the declaration $\mathbf{module}\ X = (\mathbf{val}\ e^i_m)$, and relacing it to $X.x$ for a fresh name X. We sometimes omit the index i in e^i.

The syntax for terms is mostly standard except the following. The term $m.x$ is a reference to the x-component of the module m, and \$ and **run_module** are new operators introduced by Watanabe et al. If x is bound to code of a first-class module, $\$x.y$ refers to the code of the y-component of the module, thus the \$-operator turns code of module to a module of code. The **run_module** runs the code of a module. $\langle\rangle$, $\sim$, and **run** are the standard multi-stage operators in MetaOCaml, while we take a different syntax for code of a module $\mathbf{<}\ (\mathbf{module}\ m^i : M)\ \mathbf{>}$. This distinction is introduced since we do not allow $\sim$ (the splicing operator) to be applied to the code of a module. In addition, because the term e^1 does not include a module, there is no other way to make the code of a module, and our syntax rejects expressions that cannot be translated as the example in §2.4. Our syntax is natural because it provides two operators that make the code and two operators that decompose the code.

Figure 4 defines the syntax of types. We use metavariables M for types of a module, S for a sequence of types of components, C for types of components, and σ for types of expressions. Types consist of those types in the simply-typed lambda calculus, code types, code-of-module types, reference to type components, and CSP (Cross-Stage Persistence) for types $\%\sigma$. Whereas CSP for terms in MetaOCaml allows one to embed a present-stage value into future-stage code, Watanabe et al. use CSP for types to embed a present-stage type into future-stage modules. In this work we also introduced the type $(\mathbf{module}\ M)\ \mathbf{mcod}$ to distinguish the code type for modules than that for terms ($\tau\ \mathbf{code}$) in order to disallow code of functors.[5]

[5] see §7 about this choice.

$$
\begin{aligned}
m^i &::= X \mid \mathbf{struct}\ s^i\ \mathbf{end} \mid (\mathbf{val}\ e^i_m)\\
s^i &::= \epsilon \mid c^i\ s^i\\
c^i &::= \mathbf{type}\ t \mid \mathbf{type}\ t\ =\ \sigma \mid \mathbf{module}\ X = m^i \mid \mathbf{let}\ x{:}\sigma\ =\ e^i\\
e &::= e^0 \mid e^1\\
e^0 &::= \langle e^1\rangle \mid \mathbf{run}\ e^0\\
&\quad \mid \mathbf{<}\ e^1_m\ \mathbf{>} \mid (\mathbf{run\_module}\ e^0 : M) \mid x \mid X.x \mid e^0_m\\
&\quad \mid p(e^0, \cdots, e^0) \mid \mathbf{fun}\ x \to e^0 \mid e^0\ e^0 \mid \mathbf{let}\ x\ =\ e^0\ \mathbf{in}\ e^0\\
e^1 &::= \sim e^0 \mid \$x.x \mid x \mid X.x\\
&\quad \mid p(e^1, \cdots, e^1) \mid \mathbf{fun}\ x \to e^1 \mid e^1\ e^1 \mid \mathbf{let}\ x\ =\ e^1\ \mathbf{in}\ e^1\\
e^i_m &::= (\mathbf{module}\ m^i : M)
\end{aligned}
$$

Figure 3. Syntax for terms

$$
\begin{aligned}
M &::= \mathbf{sig}\ S\ \mathbf{end}\\
S &::= \epsilon \mid C\ S\\
C &::= \mathbf{type}\ t \mid \mathbf{type}\ t\ =\ \sigma \mid \mathbf{module}\ X : M \mid \mathbf{val}\ x : \sigma\\
\sigma &::= t \mid X.t \mid \$x.t \mid \sigma\ \to\ \sigma \mid \tau\ \mathbf{code} \mid \%\sigma \mid (\mathbf{module}\ M)\\
&\quad \mid (\mathbf{module}\ M)\ \mathbf{mcod}\\
&\text{where } \tau \text{ is } \mathbf{module}\text{-free}
\end{aligned}
$$

Figure 4. Syntax for types

A typing environment is a sequence of declarations $(\mathbf{type}\ t)^i$, $(\mathbf{type}\ t\ =\ \sigma)^i$, $(\mathbf{module}\ X : M)^i$, and $(\mathbf{val}\ x : \sigma)^i$ where i is a level (stage). In this work i is either 0 or 1.

Our type system is based on Leroy's type system for modules [12], and the classic type system $\lambda\circ$ for multi-stage language (see [5] for instance), however, we have made our own modifications. Here we only explain the most typical typing rules, and the complete typing rules are shown in Appendix A.

The well-formedness rule for nested modules is:

$$\frac{E \vdash^i M\ \mathbf{wf} \qquad E, X^i \vdash^i S\ \mathbf{wf} \qquad X^i \notin \mathbf{Dom}(E)}{E \vdash^i (\mathbf{module}\ X^i : M)\ S\ \mathbf{wf}}$$

where $E \vdash^i$ means a typing judgment at the stage i and **wf** for well-formedness. At the stage i, only the elements annotated with i in the environment E may be dereferenced. M **wf** requires that all component names of M are mutually distinct, or else the condition $X^i \notin \mathbf{Dom}(E)$ is not satisfied.

The typing rule for code of a module is shown below:

$$\frac{E \vdash^1 (\mathbf{module}\ m^1 : M)\ :\ (\mathbf{module}\ M)}{E \vdash^0 \mathbf{<}\ (\mathbf{module}\ m^1 : M)\ \mathbf{>}\ :\ (\mathbf{module}\ M)\ \mathbf{mcod}}$$

The rule for dereferencing a component in code of a module is as follows:

$$\frac{(\mathbf{val}\ x : (\mathbf{module}\ (\mathbf{sig}\ S\ \mathbf{end}))\ \mathbf{mcod})^i \in E \quad S = S_1\ (\mathbf{val}\ y : \sigma)\ S_2}{E \vdash^i \ \$x.y\ :\ \sigma[t \leftarrow (\mathbf{val}\ x).t \mid t \in \mathbf{Dom}(S_1)]\ \mathbf{code}}$$

Since the $-operator turns code of a module to a module of code, its y-component should have the **code** type. This rule has an additional complexity in that we need to substitute a type variable t by $(\mathbf{val}\ x).t$ in the type σ in the conclusion. To see its reason, consider the following example: let x be $\mathbf{<}\ (\mathbf{module\ struct\ type}\ t = int\ \mathbf{let}\ y : t\ =\ 10\ \mathbf{end} : S)\ \mathbf{>}$. If the typing rule does not do substitution, the type of $\$x.y$ is t, which is a free type variable. By the substitution, it will become $(\mathbf{val}\ x).t$, which refers to the t-component of the first-class module bound to x (the type `int` in this case).

5 Our Translation

We introduce a refined translation from our source language to plain MetaOCaml by improving Watanabe et al.'s one in two respects. The first is to fix bugs. Indeed their source language contains code of functors, which cannot be (easily) translated to programs in plain MetaOCaml. We have refined their source language and define a type-preserving translation here. The second improvement is the better performance of translated code, which is the main subject of this paper. In this section, we explain the key part of our translation and the differences from Watanabe's translation. The complete definition of our translation is shown in Appendix B.

5.1 Definitions

We use the notation $[\![\ \cdot\ ]\!]^i$ for our translation, which is parameterized by the level i (for $i = 0, 1$). $[\![\ e^0\ ]\!]^0$ is the result of the translation for a level-0 (present-stage) expression e^0, and similarly $[\![\ e^1\ ]\!]^1$ for a level-1 (future-stage) expression e^1. $[\![\ \langle e^1 \rangle\ ]\!]^0$ is translated to $\langle [\![\ e^1\ ]\!]^1 \rangle$. We may omit the index i in an expression e^i. Our translation sometimes needs an additional parameter d, which shall be explained shortly.

Let us explain the key rules for translation. First, we translate code of a module to a module of code as follows:

$$[\![\ \mathbf{<}\ (\mathbf{module}\ m : M)\ \mathbf{>}\ ]\!]^0 = (\mathbf{module}\ [\![\ m\ ]\!]^1 : [\![\ M\ ]\!]^1)$$

The outermost brackets are eliminated and the components of the module will be translated at level 1. (The translation rules for the level 0 do not do much work, and are omitted.)

Next, we define a level-1 translation of a type declaration in a module as follows:

$$[\![\ \mathbf{val}\ x : \sigma\ ]\!]^1 = \mathbf{val}\ x : [\![\ \sigma\ ]\!]^1\ \mathbf{code}$$
$$[\![\ \mathbf{type}\ t\ ]\!]^1 = \mathbf{type}\ t$$
$$[\![\ \mathbf{type}\ t\ =\ \sigma\ ]\!]^1 = \mathbf{type}\ t\ =\ [\![\ \sigma\ ]\!]^1$$
$$[\![\ \mathbf{module}\ X : M\ ]\!]^1 = \mathbf{module}\ X : [\![\ M\ ]\!]^1$$

The type of a value component is translated to the **code** type. A type component and a nested module are passed through by the translation.

The level-1 translation for terms in a module is also interesting and defined as follows:

$$[\![\ \mathbf{let}\ x : \sigma\ =\ e\ ]\!]^1 = \mathbf{let}\ x : [\![\ \sigma\ ]\!]^1 \mathbf{code} = \mathbf{genlet}\ \langle [\![\ e\ ]\!]^1 \rangle$$
$$[\![\ \mathbf{type}\ t\ ]\!]^1 = \mathbf{type}\ t$$
$$[\![\ \mathbf{type}\ t\ =\ \sigma\ ]\!]^1 = \mathbf{type}\ t\ =\ [\![\ \sigma\ ]\!]^1$$
$$[\![\ \mathbf{module}\ X = m\ ]\!]^1 = \mathbf{module}\ X = [\![\ m\ ]\!]^1$$

A value component is translated into code of the value and genlet is inserted in front of it, to allow code sharing.

So far, the translation is compositional and simple, but we need one twist here. Since the translation for code of modules moves brackets from outside of a module to inside a component, it must manage the change of stages for a reference in a value component, otherwise the result will have a dangling reference. For this purpose, we introduce an additional parameter d in $[\![\ \cdot\ ]\!]^i_d$, where d is the set of declarations that may be referenced in the translation, hence their levels must be adjusted through the translation. For a sequence s of components at level 1, we accumulate the component c to the set d as follows:

$$[\![\ \epsilon\ ]\!]^1_d = \epsilon$$
$$[\![\ c\ s\ ]\!]^1_d = \begin{cases} [\![\ c\ ]\!]^1_d\ [\![\ s\ ]\!]^1_d & (c \text{ is a type component}) \\ [\![\ c\ ]\!]^1_d\ [\![\ s\ ]\!]^1_{c,d} & (otherwise) \end{cases}$$

Because variables (component names) in d have level 1 before the translation, and level-0 expressions are bound to them, we need to splice them. Hence, the rules for variables are:

$$[\![\ x\ ]\!]^1_d = \begin{cases} \sim x & (x \in \mathrm{Dom}(d)) \\ x & (otherwise) \end{cases}$$
$$[\![\ X.x\ ]\!]^1_d = \begin{cases} \sim (X.x) & (X \in \mathrm{Dom}(d)) \\ X.x & (otherwise) \end{cases}$$
$$[\![\ \$x_1.x_2\ ]\!]^1_d = (\mathbf{val}\ x_1).x_2$$

where $Dom(d)$ is the set of variable names in the domain of d. The third rule above is the one for dereferencing x_2 in the code x_1. After the translation, x_1 becomes a first-class module that consists of code of a component, thus we should translate it to $(\mathbf{val}\ x_1).x_2$ to look up the value of x_2. Although MetaOCaml does not allow such an expression, we can translate it to $X.x_2$ after declaring $\mathbf{module}\ X = (\mathbf{val}\ x_1)$.

The **run_module** operator is translated depending on the type of the target module. The translation eliminates the **run_module** operator and defines a new module. The new module contains an unpacked module X where X is a fresh name, and components of module X.

$$[\![\ (\mathbf{run\_module}\ e : \mathbf{sig}\ S\ \mathbf{end})\ ]\!]^0_d$$
$$= (\mathbf{module\ struct}$$
$$\quad \mathbf{module}\ X = (\mathbf{val}\ [\![\ e\ ]\!]^0_d)$$
$$\quad (X, S) \triangleright \bullet$$
$$\mathbf{end} : \mathbf{sig}\ [\![\ S\ ]\!]^0\ \mathbf{end})$$

The rule $(X, S) \triangleright \bullet$ applies a **run**-primitive to each value component in module X of type S. The **run_module** is applied to a nested module and the **run**-primitive is propagated.

$$
\begin{aligned}
(X, \epsilon) \triangleright \bullet &= \epsilon \\
(X,\ (\mathbf{val}\ x : \sigma)\ S) \triangleright \bullet &= \mathbf{let}\ x : \sigma\ =\ \mathbf{run}\ X.x \\
&\quad (X, S) \triangleright \bullet \\
(X,\ (\mathbf{type}\ t\ =\ \sigma)\ S) \triangleright \bullet &= \mathbf{type}\ t\ =\ X.t \\
&\quad (X, S) \triangleright \bullet \\
(X_1,\ (\mathbf{module}\ X_2 : M)\ S) \triangleright \bullet &= \mathbf{module}\ X_2 = \\
&\quad (\mathbf{val}\ [\![\ (\mathbf{run\_module}\ (\mathbf{module}\ X_1.X_2 : M) : M)\ ]\!]^0_\epsilon) \\
&\quad (X_1, S) \triangleright \bullet
\end{aligned}
$$

5.2 Translation Preserves Typing

We can prove that the following form of simple type preservation holds for our translation. If $E \vdash^0 e\ :\ \sigma$ is derivable in our type system, then $[\![\ E\ ]\!]^0_d \vdash^0 [\![\ e\ ]\!]^0_d\ :\ [\![\ \sigma\ ]\!]^0_d$ is derivable for the empty set d. We assume here that the translation for a typing environment E is defined similarly for the sequence of typing components C, and that the target type system has a typing rule for genlet, which is the same as the one for the standard let. The proof is straightforward but lengthy, so omitted in this paper.

6 Performance

We have implemented our language through the translation in the previous section, and conducted a few experiments against microbenchmarks. The result is quite positive for our claims in that the code-explosion problem in the Watanabe et al.'s study is solved, or at least, drastically reduced as long as we have experimented.

The microbenchmarks created by Watanabe et al. express a domain-specific optimization for arithmetic expressions such as an expression $0 + n \ \longrightarrow\ n$ using the tagless-final embedding [2]. Figure 5 shows the core part of the benchmark written in $\lambda^{<M>}$. The tagless-final style uses module types to embed syntax and typing rules of the object language. The module type `S` specifies a type `int_t` representing a numeric type in the object language, `int` representing a numeric literal, and the functions `add`, `sub`, `mul` and `div` correspond to four arithmetic operations. The function `suppressAddMulZero` is a program translator in such an object language. It is given an expression of type `(module S) code` and returns code of a module after performing the optimization. By applying it repeatedly, a fully optimized module can be obtained. In this section, the depth refers to the number of repeated functor applications. For the complete implementation, see Watanabe et al.'s paper [16].

The code-explosion problem shows up if we use Watanabe et al.'s translation for the above program. The function `suppressAddMulZero`, given a module `m`, splices the components of `m` into (the code of) a new module. For example, in

```
module type S = sig
  type int_t
  val int: int -> int_t
  val add: int_t -> int_t -> int_t
  val sub: int_t -> int_t -> int_t
  val mul: int_t -> int_t -> int_t
  val div: int_t -> int_t -> int_t
end

let suppressAddMulZero = fun (m: (module S)
    code) ->
  <(module struct
    type int_t = $m.int_t * bool
     let int = fun n1 ->
      if n1 = 0 then (~($m.int) 0, true)
                else (~($m.int) n1, false)
    let add = fun n1 -> fun n2 ->
      match (n1,n2) with
        (n1,b1),(n2,b2) ->
          if (b1 && b2) then (~($m.int) 0,
              true)
                        else ~($m.add) n1 n2
    let sub = fun n1 -> fun n2 ->
      if n1 = n2 then (~($m.int) 0, true)
                 else ~($m.sub) n1 n2
    let mul = fun n1 -> fun n2 ->
      match (n1,n2) with
        (n1,b1),(n2,b2) ->
          if (b1 || b2) then (~($m.int) 0,
              true)
                        else ~($m.mul) n1 n2
    let div = fun n1 -> fun n2 ->
      match (n1,n2) with
        (n1,_),(n2,_) ->
          (~($m.div) n1 n2, false)
  end: S)>

let rec fix depth m =
  if depth <= 0 then m
  else fix (depth-1) (suppressAddMulZero m)
```

Figure 5. Core Part of The Benchmarks

the `int` component, the `m.int` code is spliced twice. Thus, as the depth increases, the size of the generated code increases exponentially.

We use the following programs for experiments.

1. A MetaOCaml program translated from the benchmark program by our translator.
2. A MetaOCaml program translated from the benchmark program by Watanabe et al's translator.
3. A naive OCaml program that expresses the benchmark without code generation.

For these programs, we measure the memory usage, the execution time of generated code and the time for code generation and compilation. The measurement result is the average of 10 trials. We conduct these experiments on Ubuntu 18.04

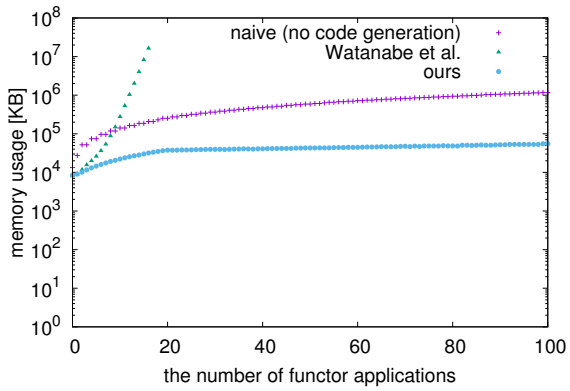

Figure 6. Memory Usage

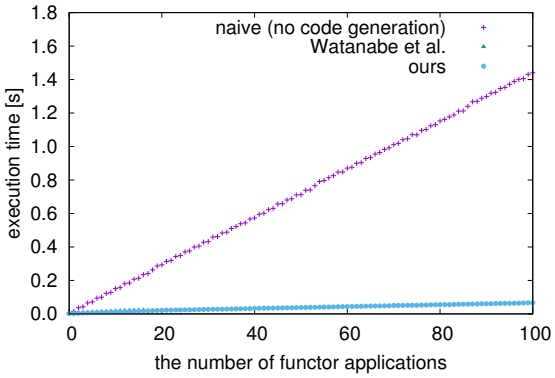

Figure 7. Execution Time

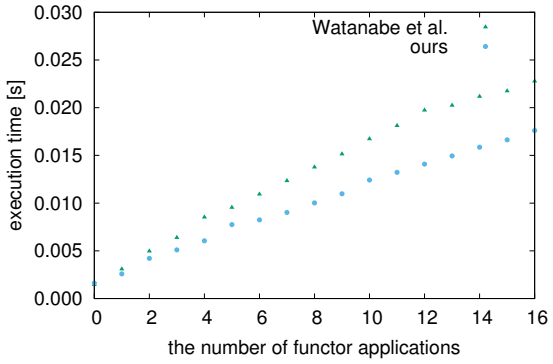

Figure 8. Execution Time (zoom in)

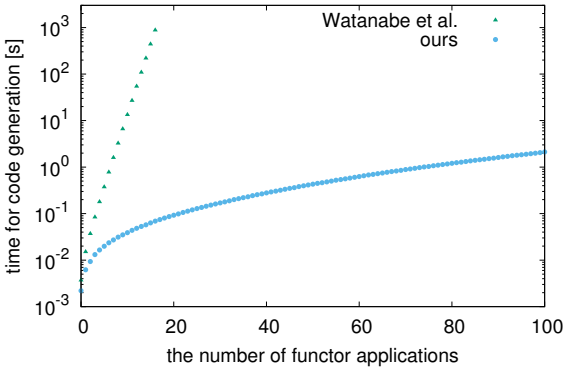

Figure 9. Time for Code Generation

LTS, Xeon E3-1225 v6@3.3GHz, Memory 32GB, BER MetaOCaml N107 (OCaml 4.07.1), byte code compiler. The memory usage is measured using the GNU time command for compiled executables, defined by the maximum resident set size of the process during its lifetime.

Figure 6 shows the memory usage where the horizontal (vertical, resp.) axis is the number of functor applications (memory usage on a logarithmic scale, resp.). The program translated by Watanabe et al.'s consumes the memory exponentially, and the experiment was only performed up to depth 16. On the other hand, the memory usage of ours is linear. The naive program without code generation uses a recursive module to repeatedly apply a functor for normalization. The recursive module contains several nested modules, and these are captured each time the functor is applied. Therefore, the memory usage of the naive program is larger than ours which creates at most 100 modules.

Figure 7 shows the execution time (excluding the time for code generation). The result shows that the generated modules run faster than the naive one. Our program is about 30% faster than Watanabe et al.'s (Figure 8). Our non-duplicating code reduces the number of steps in program execution and encourages optimization by the compiler.

Figure 9 shows the time for code generation and compilation where the vertical axis is a logarithmic scale. The time of Watanabe et al.'s program increases exponentially, while ours has a gentle slope.

In our benchmark, functors are applied to modules quite a few times, but it is not an unrealistic experiment. Since the implementation of MirageOS contains a number of functor applications, a unikernel that runs web service has functor applications of depth up to 10. At depth 10, the execution time is 0.15 seconds for the naive program (without code generation), while it is only 0.012 seconds for our program. MirageOS actually contains more indirections than this, because it contains a large number of components and nested modules. Hence, we expect that the benefit of efficient code generation for modules will be greater.

7 Discussion

7.1 Code of Functors

This design choice was motivated by the following consideration. We first point out that removing this restriction may be possible, but needs an extra cost. Let us consider an expression `<fun m -> (module ...)>` of type `((module`

`A) -> (module B))` code, which is code of a functor in our representation. Since plain MetaOCaml prohibits modules within brackets, the brackets should be translated away, and one possible solution is to use the unstaging translation by Kiselyov [10]. There is, however, a problem with this solution, in that the unstaging translation translates a bracket-expression to a thunk, and it would severely degrade the performance of generated code and also complicate the whole translation.

We think that applications for code of functors are not sufficiently appealing, in the way that the cost mentioned above is justified in the context of program generation. As is discussed in this paper, functor applications are a major source of indirections and penalize performance, and the purpose of making use of the code-generation technique is to optimize (inline) functor applications. Since MetaOCaml is generative, we cannot manipulate code inside brackets, hence the code of functors may not be further optimized.

7.2 Remaining Duplicated Code

Although our translation eliminates most code duplication, it still allows duplicated code to be generated at the top level. For example, if the module `printLogger` is at the top level (i.e., its code is run), the function `info` and `warn` contain the code of the function `print`. Namely, each component has no duplicated code, but duplicated may exist between top-level components. We think that this is not a serious problem, as such duplication appears only at the top level, and the performance problem is solved using our solution as is shown in the previous section.

8 Related Work

In this section, we explain several closely related work to our work. For comparison, we pick up three previous works, Macros, Flambda and MLton, which can eliminate module overhead at compile time. We also mention Functoria, which helps developers build applications that use many modules, as well as other research on using genlet.

Macros [20] are an extension of OCaml which allows type-safe compile-time metaprogramming. It provides constructors such as quoting `<<e>>` and splicing `$e` and can manipulate code fragments similar to MetaOCaml. Macros fully support the OCaml language including the module system, and the abstraction overhead of modules can be eliminated in a similar way as ours. However, our approach can generate code specialized for the runtime environment, such as the number of CPU cores and memory size.

Flambda [6] is an optimizer of the OCaml compiler which inlines a program whenever possible. Since functor application is the target of inlining for Flambda, an indirection discussed in this paper might be eliminated by Flambda, too. Also, MLton [1] is an optimizing compiler for the Standard ML, which aggressively inlines functors. While both of these two studies are fully automated, our approach has its own merit in that one is given full control as to how and what code is generated.

Functoria [13] is a domain-specific language mainly used in MirageOS, which can manipulate modules and functors to build modular applications. Its main purpose is to scrap the boilerplate associated with programs which use modules. Functoria generates an OCaml program from a configuration that describes how to combine modules. Since MetaOCaml does not allow code of modules to be generated as values, Functoria currently uses an ad hoc approach to generate code of modules as strings. We hope that our work improves the implementation of Functoria in the future.

In this study, we have extensively used the genlet primitive in MetaOCaml, which is not yet used in many applications, but has huge potential in realizing code sharing in various forms. As another application of using genlet, a recent study by Yallop and Kiselyov [19] makes use of genlet for generating mutually-recursive definitions.

9 Conclusion

In this paper, we have proposed a refined translation for an extension of core MetaOCaml that can generate and manipulate code of modules. We have conducted experiments for micro-benchmarks and shown that our translation gives an space- and time- efficient code for applications which need repeated applications of functors to modules. Our approach provides a way to explicitly perform the domain-specific optimizations, in the programmer's responsibility, without relying on a compiler. We think that MirageOS is one of such applications where approximately 10-times nested functor applications are used in practice.

Our contributions in this paper are summarized as follows. First, we experimentally confirmed that dynamic let insertion by genlet can go across the boundaries of modules and functors, and can be used to avoid the code duplication problem in a relatively large codebase. Second, we have solved the code-explosion problem in the previous study by Watanabe et al., and opened a way to generate code using high-level programming which makes heavy use of module abstraction. Third, we have extended $\lambda^{<M>}$ to allow code generation of submodules, gave a complete formulation including **run_module** , and studied its type system.

In future work, we plan to extend our source language to a more realistic one, such as side effects and polymorphism, mutually-recursive modules, and also develop practical applications using our language for generation of code of modules. Investigating theoretical foundation of genlet is also an interesting future work.

A Typing Rules

$\boxed{E \vdash^i M\ \mathbf{wf}}$

$$\frac{E \vdash^i S\ \mathbf{wf}}{E \vdash^i \mathbf{sig}\ S\ \mathbf{end}\ \mathbf{wf}}\ (\text{WF-Sig})$$

$\boxed{E \vdash^i S\ \mathbf{wf}}$

$$\frac{}{E \vdash^i \epsilon\ \mathbf{wf}}\ (\text{WF-Empty})$$

$$\frac{E \vdash^i \sigma\ \mathbf{wf} \qquad E, x^i \vdash^i S\ \mathbf{wf} \qquad x^i \notin \mathbf{Dom}(E)}{E \vdash^i (\mathbf{val}\ x^i : \sigma)\ S\ \mathbf{wf}}\ (\text{WF-Val})$$

$$\frac{E,\ \mathbf{type}\ t \vdash^i S\ \mathbf{wf} \qquad t^i \notin \mathbf{Dom}(E)}{E \vdash^i (\mathbf{type}\ t)\ S\ \mathbf{wf}}\ (\text{WF-TypeAbs})$$

$$\frac{E \vdash^i \sigma\ \mathbf{wf} \qquad E,\ \mathbf{type}\ t\ =\ \sigma \vdash^i S\ \mathbf{wf} \qquad t^i \notin \mathbf{Dom}(E)}{E \vdash^i (\mathbf{type}\ t\ =\ \sigma)\ S\ \mathbf{wf}}\ (\text{WF-Type})$$

$$\frac{E \vdash^i M\ \mathbf{wf} \qquad E, X^i \vdash^i S\ \mathbf{wf} \qquad X^i \notin \mathbf{Dom}(E)}{E \vdash^i (\mathbf{module}\ X^i : M)\ S\ \mathbf{wf}}\ (\text{WF-Mod})$$

$\boxed{E \vdash^i \sigma\ \mathbf{wf}}$

$$\frac{(\mathbf{type}\ t\ =\ \sigma) \in E}{E \vdash^i t\ \mathbf{wf}}\ (\text{T-Var})$$

$$\frac{(\mathbf{module}\ X : \mathbf{sig}\ S\ \mathbf{end})^i \in E \qquad (\mathbf{type}\ t\ =\ \tau) \in S}{E \vdash^i X.t\ \mathbf{wf}}\ (\text{T-Dot})$$

$$\frac{(\mathbf{val}\ x\colon (\mathbf{module}\ \mathbf{sig}\ S\ \mathbf{end})\ \mathbf{mcod})^0 \in E \quad (\mathbf{type}\ t = \tau) \in S}{E \vdash^i \$x.t\ \mathbf{wf}}\ (\text{T-DotCode})$$

$$\frac{E \vdash^i \sigma_1\ \mathbf{wf} \qquad E \vdash^i \sigma_2\ \mathbf{wf}}{E \vdash^i \sigma_1 \to \sigma_2\ \mathbf{wf}}\ (\text{T-Arr})$$

$$\frac{E \vdash^1 \tau\ \mathbf{wf}}{E \vdash^0 \tau\ \mathbf{code}\ \mathbf{wf}}\ (\text{T-Code})$$

$$\frac{E \vdash^0 \sigma\ \mathbf{wf}}{E \vdash^1 \%\,\sigma\ \mathbf{wf}}\ (\text{T-Csp})$$

$$\frac{E \vdash^i M\ \mathbf{wf}}{E \vdash^i (\mathbf{module}\ M)\ \mathbf{wf}}\ (\text{T-Mod})$$

$$\frac{E \vdash^1 (\mathbf{module}\ M)\ \mathbf{wf}}{E \vdash^0 (\mathbf{module}\ M)\ \mathbf{mcod}\ \mathbf{wf}}\ (\text{T-ModCode})$$

Figure 10. Typing rules for types

$\boxed{E \vdash^i m : M}$

$$\frac{(\mathbf{module}\ X : M)^i \in E}{E \vdash^i X : M}\ (\text{M-Var})$$

$$\frac{E \vdash^i s : S}{E \vdash^i \mathbf{struct}\ s\ \mathbf{end} : \mathbf{sig}\ S\ \mathbf{end}}\ (\text{M-Str})$$

$$\frac{E \vdash^i e_m : (\mathbf{module}\ M)}{E \vdash^i (\mathbf{val}\ e_m) : M}\ (\text{M-Val})$$

$\boxed{E \vdash^i s : S}$

$$\frac{}{E \vdash^i \epsilon : \epsilon}\ (\text{S-Empty})$$

$$\frac{E \vdash^i e : M \qquad E \vdash^i \sigma\ \mathbf{wf} \qquad E, (\mathbf{val}\ x{:}\sigma)^i \vdash^i s : S \qquad x^i \notin \mathbf{Dom}(E)}{E \vdash^i (\mathbf{let}\ x{:}\sigma = e)\ s : (\mathbf{val}\ x{:}\sigma)\ S}\ (\text{S-Let})$$

$$\frac{E,\ (\mathbf{type}\ t)^i \vdash^i s : S \qquad t^i \notin \mathbf{Dom}(E)}{E \vdash^i (\mathbf{type}\ t)\ s : (\mathbf{type}\ t)\ S}\ (\text{S-TypeAbs})$$

$$\frac{E \vdash^i \sigma\ \mathbf{wf} \qquad E,\ (\mathbf{type}\ t\ =\ \sigma)^i \vdash^i s : S \qquad t^i \notin \mathbf{Dom}(E)}{E \vdash^i (\mathbf{type}\ t\ =\ \sigma)\ s : (\mathbf{type}\ t\ =\ \sigma)\ S}\ (\text{S-Type})$$

$$\frac{E \vdash^i m : M \quad E \vdash^i M\ \mathbf{wf} \quad E, (\mathbf{module}\ X{:}M)^i \vdash^i s : S \quad X^i \notin \mathbf{Dom}(E)}{E \vdash^i (\mathbf{module}\ X{=}m)\ s : (\mathbf{module}\ X{:}M)\ S}\ (\text{S-Mod})$$

$\boxed{E \vdash^i e : \sigma}$

$$\frac{(\mathbf{val}\ x : \sigma) \in E}{E \vdash^i x : \sigma}\ (\text{E-Var})$$

$$\frac{(\mathbf{module}\ X : \mathbf{sig}\ S\ \mathbf{end})^i \in E \qquad S = S_1\ (\mathbf{val}\ x : \sigma)\ S_2}{E \vdash^i X.x : \sigma[t \leftarrow X.t \mid t \in \mathbf{Dom}(S_1)]}\ (\text{E-Dot})$$

$$\frac{(\mathbf{val}\ x\colon (\mathbf{module}\ (\mathbf{sig}\ S\ \mathbf{end}))\ \mathbf{mcod})^i \in E \quad S{=}S_1\ (\mathbf{val}\ y{:}\sigma)\ S_2}{E \vdash^i \$x.y : \sigma[t \leftarrow (\mathbf{val}\ x).t \mid t \in \mathbf{Dom}(S_1)]\ \mathbf{code}}\ (\text{E-DotCode})$$

$$\frac{E \vdash^i \sigma_1\ \mathbf{wf} \qquad E,\ (\mathbf{val}\ x : \sigma_1)^i \vdash^i e : \sigma_2}{E \vdash^i \mathbf{fun}\ x \to e : \sigma_1 \to \sigma_2}\ (\text{E-Fun})$$

$$\frac{E \vdash^i e_1 : \sigma_1 \to \sigma_2 \qquad E \vdash^i e_2 : \sigma_1}{E \vdash^i e_1\ e_2 : \sigma_2}\ (\text{E-App})$$

$$\frac{E \vdash^i e_1 : \sigma_1 \qquad E,\ (\mathbf{val}\ x : \sigma_1)^i \vdash^i e_2 : \sigma_2}{E \vdash^i \mathbf{let}\ x\ =\ e_1\ \mathbf{in}\ e_2 : \sigma_2}\ (\text{E-Let})$$

$$\frac{E \vdash^i M\ \mathbf{wf} \qquad E \vdash^i m : M}{E \vdash^i (\mathbf{module}\ m : M) : (\mathbf{module}\ M)}\ (\text{E-Mod})$$

$$\frac{E \vdash^1 e : \tau}{E \vdash^0 \langle e \rangle : \tau\ \mathbf{code}}\ (\text{E-Bra})$$

$$\frac{E \vdash^0 e : \sigma\ \mathbf{code}}{E \vdash^1 \sim e : \sigma}\ (\text{E-Esc})$$

$$\frac{E \vdash^0 e : \sigma\ \mathbf{code}}{E \vdash^0 \mathbf{run}\ e : \sigma}\ (\text{E-Run})$$

$$\frac{E \vdash^1 e_m : (\mathbf{module}\ M)}{E \vdash^0 <\, e_m\, > : (\mathbf{module}\ M)\ \mathbf{mcod}}\ (\text{E-BraMod})$$

$$\frac{E \vdash^0 e_m : (\mathbf{module}\ M)\ \mathbf{mcod}}{E \vdash^0 (\mathbf{run\_module}\ e_m : M) : (\mathbf{module}\ M)}\ (\text{E-Runmod})$$

Figure 11. Typing rules for terms

B Translation

$\boxed{[\![\ m\ ]\!]^i_d}$

$$
\begin{aligned}
[\![\ X\ ]\!]^i_d &= X \\
[\![\ \textbf{struct}\ s\ \textbf{end}\ ]\!]^i_d &= \textbf{struct}\ [\![\ s\ ]\!]^i_d\ \textbf{end} \\
[\![\ (\textbf{val}\ e)\ ]\!]^i_d &= (\textbf{val}\ [\![\ e\ ]\!]^i_d)
\end{aligned}
$$

$\boxed{[\![\ s\ ]\!]^i_d}$

$$
\begin{aligned}
[\![\ \epsilon\ ]\!]^i_d &= \epsilon \\
[\![\ c\ s\ ]\!]^0_d &= [\![\ c\ ]\!]^0_d\ [\![\ s\ ]\!]^0_d \\
[\![\ c\ s\ ]\!]^1_d &= \begin{cases} [\![\ c\ ]\!]^1_d\ [\![\ s\ ]\!]^1_d & (c \text{ is a type component}) \\ [\![\ c\ ]\!]^1_d\ [\![\ s\ ]\!]^1_{c,d} & (otherwise) \end{cases}
\end{aligned}
$$

$\boxed{[\![\ c\ ]\!]^i_d}$

$$
\begin{aligned}
[\![\ \textbf{type}\ t\ ]\!]^i_d &= \textbf{type}\ t \\
[\![\ \textbf{type}\ t\ =\ \sigma\ ]\!]^i_d &= \textbf{type}\ t\ =\ [\![\ \sigma\ ]\!]^i \\
[\![\ \textbf{module}\ X = m\ ]\!]^i_d &= \textbf{module}\ X = [\![\ m\ ]\!]^i_d \\
[\![\ \textbf{let}\ x : \sigma\ =\ e\ ]\!]^0_d &= \textbf{let}\ x : [\![\ \sigma\ ]\!]^0\ =\ [\![\ e\ ]\!]^0_d \\
[\![\ \textbf{let}\ x : \sigma\ =\ e\ ]\!]^1_d &= \textbf{let}\ x : [\![\ \sigma\ ]\!]^1\ \textbf{code}\ =\ \textbf{genlet}\ \langle [\![\ e\ ]\!]^1_d \rangle
\end{aligned}
$$

$\boxed{[\![\ e\ ]\!]^i_d}$

$$
\begin{aligned}
[\![\ x\ ]\!]^0_d &= x \\
[\![\ x\ ]\!]^1_d &= \begin{cases} \sim x & (x \in \mathrm{Dom}(d)) \\ x & (otherwise) \end{cases} \\
[\![\ X.x\ ]\!]^0_d &= X.x \\
[\![\ X.x\ ]\!]^1_d &= \begin{cases} \sim (X.x) & (X \in \mathrm{Dom}(d)) \\ X.x & (otherwise) \end{cases} \\
[\![\ \textbf{fun}\ x\ \rightarrow\ e\ ]\!]^i_d &= \textbf{fun}\ (x : [\![\ \sigma\ ]\!]^i)\ \rightarrow\ [\![\ e\ ]\!]^i_d \\
[\![\ e_1\ e_2\ ]\!]^i_d &= [\![\ e_1\ ]\!]^i_d\ [\![\ e_2\ ]\!]^i_d \\
[\![\ \textbf{let}\ x\ =\ e_1\ \textbf{in}\ e_2\ ]\!]^i_d &= \textbf{let}\ x\ =\ [\![\ e_1\ ]\!]^i_d\ \textbf{in}\ [\![\ e_2\ ]\!]^i_d \\
[\![\ \langle e \rangle\ ]\!]^0_d &= \langle [\![\ e\ ]\!]^1_d \rangle \\
[\![\ \sim e\ ]\!]^1_d &= \sim [\![\ e\ ]\!]^0_d \\
[\![\ \textbf{run}\ e\ ]\!]^0_d &= \textbf{run}\ [\![\ e\ ]\!]^0_d \\
[\![\ \$x_1.x_2\ ]\!]^i_d &= (\textbf{val}\ x_1).x_2 \\
[\![\ (\textbf{module}\ m : M)\ ]\!]^i_d &= (\textbf{module}\ [\![\ m\ ]\!]^i_d : [\![\ M\ ]\!]^i) \\
[\![\ \texttt{<}\ (\textbf{module}\ m : M)\ \texttt{>}\ ]\!]^0_d &= (\textbf{module}\ [\![\ m\ ]\!]^1_d : [\![\ M\ ]\!]^1) \\
[\![\ (\textbf{run\_module}\ e : \textbf{sig}\ S\ \textbf{end})\ ]\!]^0_d &= (\textbf{module struct} \\
&\quad \textbf{module}\ X = (\textbf{val}\ [\![\ e\ ]\!]^0_d) \\
&\quad (X, S) \triangleright \bullet \\
&\quad \textbf{end} : \textbf{sig}\ [\![\ S\ ]\!]^0\ \textbf{end})
\end{aligned}
$$

Figure 12. Translation for terms

$\boxed{(X, S) \triangleright \bullet}$

$$
\begin{aligned}
(X, \epsilon) \triangleright \bullet &= \epsilon \\
(X,\ (\textbf{val}\ x : \sigma)\ S) \triangleright \bullet &= \textbf{let}\ x : \sigma\ =\ \textbf{run}\ X.x \\
&\quad (X, S) \triangleright \bullet \\
(X,\ (\textbf{type}\ t\ =\ \sigma)\ S) \triangleright \bullet &= \textbf{type}\ t\ =\ X.t \\
&\quad (X, S) \triangleright \bullet \\
(X_1,\ (\textbf{module}\ X_2 : M)\ S) \triangleright \bullet &= \textbf{module}\ X_2 = \\
&\quad (\textbf{val}\ [\![\ (\textbf{run\_module}\ (\textbf{module}\ X_1.X_2 : M) : M)\ ]\!]^0_\epsilon) \\
&\quad (X_1, S) \triangleright \bullet
\end{aligned}
$$

Figure 13. Translation related to **run_module**

$\boxed{[\![\ M\ ]\!]^i}$

$$
[\![\ \textbf{sig}\ S\ \textbf{end}\ ]\!]^i = \textbf{sig}\ [\![\ S\ ]\!]^i\ \textbf{end}
$$

$\boxed{[\![\ S\ ]\!]^i}$

$$
\begin{aligned}
[\![\ \epsilon\ ]\!]^i &= \epsilon \\
[\![\ C\ S\ ]\!]^i &= [\![\ C\ ]\!]^i\ [\![\ S\ ]\!]^i
\end{aligned}
$$

$\boxed{[\![\ C\ ]\!]^i}$

$$
\begin{aligned}
[\![\ \textbf{type}\ t\ ]\!]^i &= \textbf{type}\ t \\
[\![\ \textbf{type}\ t\ =\ \sigma\ ]\!]^i &= \textbf{type}\ t\ =\ [\![\ \sigma\ ]\!]^i \\
[\![\ \textbf{module}\ X : M\ ]\!]^i &= \textbf{module}\ X : [\![\ M\ ]\!]^i \\
[\![\ \textbf{val}\ x : \sigma\ ]\!]^0 &= \textbf{val}\ x : [\![\ \sigma\ ]\!]^0 \\
[\![\ \textbf{val}\ x : \sigma\ ]\!]^1 &= \textbf{val}\ x : [\![\ \sigma\ ]\!]^1\ \textbf{code}
\end{aligned}
$$

$\boxed{[\![\ \sigma\ ]\!]^i}$

$$
\begin{aligned}
[\![\ t\ ]\!]^i &= t \\
[\![\ X.t\ ]\!]^i &= X.t \\
[\![\ \$x.t\ ]\!]^i &= (\textbf{val}\ x).t \\
[\![\ \sigma_1\ \rightarrow\ \sigma_2\ ]\!]^i &= [\![\ \sigma_1\ ]\!]^i\ \rightarrow\ [\![\ \sigma_2\ ]\!]^i \\
[\![\ \sigma\ \textbf{code}\ ]\!]^0 &= [\![\ \sigma\ ]\!]^0\ \textbf{code} \text{ if } \sigma \text{ is } \textbf{module}\text{-free} \\
[\![\ \%\sigma\ ]\!]^1 &= \sigma \\
[\![\ (\textbf{module}\ M)\ ]\!]^i &= (\textbf{module}\ [\![\ M\ ]\!]^i) \\
[\![\ (\textbf{module}\ M)\ \textbf{mcod}\ ]\!]^0 &= (\textbf{module}\ [\![\ M\ ]\!]^1)
\end{aligned}
$$

Figure 14. Translation for types

Acknowledgments

The second author is partly supported by MEXT's Grant-in-Aid for Scientific Research (B) No. 18H03218.

References

[1] 1997. MLton, a whole program optimizing compiler for Standard ML. http://www.mlton.org. (1997).

[2] 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

[3] Olivier Danvy. 1996. *Pragmatic Aspects of Type-Directed Partial Evaluation.* Technical Report. BRICS Report Series RS-96-15. 27 pages.

[4] Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990.* 151–160. https://doi.org/10.1145/91556.91622

[5] 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

[6] Xavier Leroy et al. 2019. Chapter 21. Optimisation with Flambda. The OCaml system release 4.09. (2019). https://caml.inria.fr/pub/docs/manual-ocaml/flambda.html

[7] 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 2016, St. Petersburg, FL, USA, January 20 - 22, 2016.* 103–108. https://doi.org/10.1145/2847538.2847548

[8] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2011. Shifting the stage - Staging with delimited control. *J. Funct. Program.* 21, 6 (2011), 617–662. https://doi.org/10.1017/S0956796811000256

[9] Oleg Kiselyov. 2012. Delimited control in OCaml, abstractly and concretely. *Theor. Comput. Sci.* 435 (2012), 56–76. https://doi.org/10.1016/j.tcs.2012.02.025

[10] Oleg Kiselyov. 2015. Generating Code with Polymorphic let: A Ballad of Value Restriction, Copying and Sharing. In *Proceedings ML Family / OCaml Users and Developers workshops, ML Family/OCaml 2015, Vancouver, Canada, 3rd & 4th September 2015.* 1–22. https://doi.org/10.4204/EPTCS.241.1

[11] Oleg Kiselyov. 2017. Let-insertion as a primitive. (2017). http://okmij.org/ftp/ML/MetaOCaml.html#genlet

[12] Xavier Leroy. 2000. A modular module system. *J. Funct. Program.* 10, 3 (2000), 269–303. http://journals.cambridge.org/action/displayAbstract?aid=54525

[13] Gabriel Radanne, Thomas Gazagnaire, Anil Madhavapeddy, Jeremy Yallop, Richard Mortier, Hannes Mehnert, Mindy Preston, and David J. Scott. 2019. Programming Unikernels in the Large via Functor Driven Development. *CoRR* abs/1905.02529 (2019). arXiv:1905.02529 http://arxiv.org/abs/1905.02529

[14] Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Arvind K. Sujeeth, Manohar Jonnalagedda, Nada Amin, Georg Ofenbeck, Alen Stojanov, Yannis Klonatos, Mohammad Dashti, Christoph Koch, Markus Püschel, and Kunle Olukotun. 2015. Go Meta! A Case for Generative Programming and DSLs in Performance Critical Systems. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA.* 238–261. https://doi.org/10.4230/LIPIcs.SNAPL.2015.238

[15] 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 2016, St. Petersburg, FL, USA, January 20 - 22, 2016.* 37–48. https://doi.org/10.1145/2847538.2847542

[16] Takahisa Watanabe. 2017. Program Generation for ML Modules. (2017). http://logic.cs.tsukuba.ac.jp/~takahisa/module-generation.html

[17] Takahisa Watanabe and Yukiyoshi Kameyama. 2018. Program generation for ML modules (short paper). In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, Los Angeles, CA, USA, January 8-9, 2018.* 60–66. https://doi.org/10.1145/3162072

[18] Jeremy Yallop. 2016. Staging generic programming. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016.* 85–96. https://doi.org/10.1145/2847538.2847546

[19] Jeremy Yallop and Oleg Kiselyov. 2019. Generating mutually recursive definitions. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM@POPL 2019, Cascais, Portugal, January 14-15, 2019.* 75–81. https://doi.org/10.1145/3294032.3294078

[20] Jeremy Yallop and Leo White. 2015. Modular macros. OCaml Users and Developers Workshop. (2015).