# Type-Safe Generation of Modules in Applicative and Generative Styles

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

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

## Abstract

The MetaML approach for multi-stage programming provides the static guarantee of type safety and scope safety for generated code, regardless of the values of static parameters. Modules are indispensable to build large-scale programs in ML-like languages, however, they have a performance problem. To solve this problem, several languages proposed recently allow one to generate ML-style modules. Unfortunately, those languages had the problems of limited expressiveness, incomplete proofs, and code explosion.

This paper proposes two-stage programming languages for module generation, which solve all the above issues. Our languages accommodate two styles: first-class modules with generative functors and second-class modules with applicative functors. Module generation in both styles is shown to have their own merits by concrete examples. We present type systems, and type-preserving translations from our languages to plain MetaOCaml. We also show the results of performance measurements, which confirms the effectiveness of our languages.

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

***Keywords:*** Program Generation, Modules, Type Safety, Program Transformation

**ACM Reference Format:**
Yuhi Sato and Yukiyoshi Kameyama. 2021. Type-Safe Generation of Modules in Applicative and Generative Styles. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '21), October 17–18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3486609.3487209



*GPCE '21, October 17–18, 2021, Chicago, IL, USA*

ACM ISBN 978-1-4503-9112-2/21/10…$15.00
https://doi.org/10.1145/3486609.3487209

## 1 Introduction

Modules provide a high-level abstraction mechanism to programming languages. They are indispensable to build large-scale programs in ML-family languages, just like classes in many object-oriented languages, and type classes in Haskell. Coq[1] and MirageOS[2] are two successful examples of large-scale applications in OCaml, a dialect of ML-family languages, both of which extensively use modules. While modules are useful as building blocks of large programs and allow separate compilation, the independent nature of modules sometimes has a performance problem, as a function call in a different module needs indirection. Although each software system including MirageOS solves this problem in its own way, a uniform and natural solution is called for, and several authors proposed to apply the program-generation technique to modules to improve the performance.

Program generation, or multi-stage programming, is to separate the execution of programs into two or more stages. At the first stage, code is generated, and at the second stage, the generated code is executed. The result of the second stage may also be code, and the same pattern may continue at the third and later stages. The generated code at earlier stages can be specialized in some input data (*static* data). The merit of this separation is that the specialized code with the other input data (*dynamic* data) is expected to run faster than the original, unspecialized code. Among various studies on program generation, the MetaML approach [16, 19] uses quasi-quotation to represent generated code, and focuses on giving the static guarantee of type safety of *any* generated code. MetaOCaml[3], a multi-stage extension of OCaml, is one of the most successful languages in the MetaML-style program generation [10, 17]. MetaOCaml allows the generation of terms only, since the foundational type theory for MetaML-style languages [8, 18] targets them only.

This paper addresses the efficiency problem for ML-style modules, and proposes two languages that allow generation of ML-style modules in a type-safe way. We are not the first to propose such a language. Inoue et al. [9] were the first to argue an imaginary extension of MetaOCaml which allows generating code of modules. Later, Watanabe et al. [20] and Sato et al. [15] proposed concrete languages

[1] https://coq.inria.fr
[2] https://mirage.io
[3] http://okmij.org/ftp/ML/MetaOCaml.html

for generating and manipulating code of modules as well as implemented the language, and showed that applying program generation to ML-style modules reduces the overhead of modules for several examples. Sato et al. solved a code explosion problem in Watanabe et al.'s study that the size of generated code may increase exponentially in certain programs. Unfortunately, these proposals are premature, and have several problems. First, they are implemented by translating to MetaOCaml, however, no formal properties on the translations, such as type preservation, were proved. Hence, it is not certain whether all terms in their languages are translated. Second, while their languages can generate first-class modules with generative functors, there exist examples that need module generation with second-class modules and applicative functors, which are the standard functors in the current OCaml. Finally, several important features in modular programming such as abstract types and nested modules are not supported by existing languages.

In this paper, we propose two programming languages and study their type-theoretic foundation, which solve the above problems. Namely, our languages allow one to generate ML-modules in the applicative style with second-class modules, and in the generative style with first-class modules. We show that there is a useful program which can be written in the applicative style, but not in the generative style. We formalize a type system for the language in the applicative style, and show that a translation from it to plain MetaOCaml preserves typing. The translation is implemented on top of MetaOCaml, and our experiment shows that both the performance problem by module separation and the code explosion problem are resolved by our system.

Various works in the literature proposed languages for generating high-level abstractions such as modules. Racket's module system with *submodules* [7] is one of the closest to ours, which enables phase separation for language extensions, and coexists with the hygienic macro system. Our languages are hygine, and have nested modules and phase separation, hence they share many features. A major difference between the two is that we take the purely generative[4] approach, which allows us to prove the static assurance of type safety[5] for all generated code. In fact, types help a lot in code generation, and abstract types are key for the abstractions by ML-style modules. In this paper, we devote ourselves to build a type-theoretic foundation of our languages to eliminate the problems above, and to our knowledge, this paper gives the first type system for module generation, which has a complete definition and a translation to MetaOCaml. It is an interesting future work to investigate how our languages can be used for language extensions as studied in the literature.

[4]In the purely generative language, once code values are generated, they cannot be inspected or analyzed.

[5]Strictly speaking, type safety holds for our language without the `run` primitive.

Our contribution is summarized as follows.

- We define two languages that allow module generation in two stages. The first one has first-class modules and generative functors as in the existing works, while the second one has second-class modules and applicative functors, which has never been studied as the target of module generation in the literature.
- Our languages have important features in ML-style modular programming such as abstract types and nested modules, which are missing in the earlier works.
- By concrete examples, we show that module generation in the applicative style is useful to eliminate indirect calls beyond module boundaries.
- We formulate precise type systems for our languages. The traditional type system for applicative functors has involved typing rules to cope with paths, and we successfully generalize it to the two-stage language for module generation. We prove that translations from our languages to plain MetaOCaml preserve typing.
- We implement the translations, and conduct the performance measurements, which show that our two languages can eliminate the overhead of functor applications and are free from code explosion.

The rest of this paper is organized as follows: Section 2 explains several important concepts about ML-modules. Section 3 shows concrete examples written in our languages and explains how our languages are useful in writing code and improving efficiency. Section 4 formally introduces our languages and their type systems, and Section 5 gives type-preserving translations to plain MetaOCaml. Section 6 shows the results of experiments on a microbenchmark. Section 7 states the related work, and Section 8 gives conclusion and future work.

## 2 Background: ML-modules

Modules are a language feature in OCaml to package relevant definitions and separate specifications and implementations, which allow us to develop large-scale applications in a type-safe way that achieves reusability and maintainability. This section illustrates the basics and key features of modules by examples.

### 2.1 Structures

*Structures* correspond to implementations of modules, which are defined by a sequence of *components*. The *components* consist of definitions of types, values, and modules.

Figure 1 shows an example of a structure that represents a set of integers. The structure `IntSet` is defined by the expression **struct** ... **end**, and has two type components, `elt_t` and `set_t`, that represent the type of the element and the type of the set, respectively. In this case, the type `elt_t` is defined as the type `int`, and the `set_t` is defined as the list of `elt_t`.

```
module IntSet =
  struct
    type elt_t = int
    type set_t = elt_t list
    let rec member elt set =
      match set with
      | []       → false
      | hd :: tl → elt = hd
                   || member elt tl
  end
```

**Figure 1.** Structure for the Set Module

```
module type SET =
  sig
    type elt_t
    type set_t
    val member: elt_t→set_t→bool
  end
```

**Figure 2.** Signature for the Set Module

In addition, it has the value component `member` which returns whether the argument `set` contains the argument `elt`. Components can be referenced from outside the structure by the *dot notation*. For example, we write `IntSet.member` to refer to the function `member`. Module structures may be simply called modules.

### 2.2 Signatures

*Signatures* correspond to a specification, or an interface, of modules. Signatures achieve data abstraction to eliminate programs that depend on an implementation of modules. Therefore, signatures make it easy to modify or replace an implementation of modules, which improves the maintainability of programs.

Continuing with the example in Figure 1, users of `IntSet` should not know the implementation details. To hide the fact that the set is implemented by the list, we can define the signature `SET` for the structure `IntSet` as shown in Figure 2. The signature `SET` is defined by the expression **sig** ... **end**, which contains a sequence of specifications for the components in the `IntSet`. `elt_t` and `set_t` are *abstract types* that hide an implementation of corresponding type components. The function `member` takes values of types `elt_t` and `set_t`, and returns a value of type `bool`.

A structure can be *sealed* in OCaml. For instance, by defining `module IntSet' = (IntSet : SET)`, the equivalence `IntSet'.elt_t = int` does not hold.

### 2.3 Functors

*Functors* are modules parameterized by modules and correspond to functions over modules, which achieve reusability. Figure 3 shows an example of a functor that makes a module

```
module type EQ =
  sig
    type t
    val eq: t → t → bool
  end
module MakeSet (Eq: EQ): SET =
  struct
    type elt_t = Eq.t
    type set_t = elt_t list
    let rec member elt set =
      match set with
      | []       → false
      | hd :: tl → Eq.eq elt hd
                   || member elt tl
  end
```

**Figure 3.** Functor for Set

```
module IntEq: EQ =
  struct
    type t = int
    let eq x y = Int.equal x y
  end
module StringEq: EQ =
  struct
    type t = string
    let eq x y = String.equal x y
  end
module IntSet = MakeSet(IntEq)
module StringSet = MakeSet(StringEq)
```

**Figure 4.** Integer Set and String Set by Functor

of sets. The signature `EQ` has the type component `t`, and the function `eq`, which is intended to be an equality function over the type `t`. The functor `MakeSet` takes a module `Eq` with the signature `EQ`, and creates a `SET` module using `Eq` as the elements of the set. Figure 4 re-defines the structure `IntSet` in Figure 1 by applying the functor `MakeSet` to `IntEq`. `IntEq` is a structure that contains the concrete components for integers, sealed with `EQ`. We can get another `SET` structure by applying `MakeSet` to another structure with the signature `EQ`. `StringSet` is obtained by applying `MakeSet` to a string structure.

### 2.4 Modules vs Classes

ML-style modules provide a useful abstraction to programs, just as classes in object-oriented languages do. However, there are significant differences between the two.

The first, and most notable, difference is that classes can have states, while modules without side effects cannot. In ML, stateful objects should be created by another means.

Another difference is that the signature of a module can have abstract types. The signature `SET` in Figure 2 has the type components `elt_t` and `set_t`, which are left unspecified. They can be used in `SET` as if they are actual types.

Abstract types make it possible to write many interesting programs or programming patterns such as modular implicits [21], and tagless-final embedding [2].

### 2.5 Generative Functors vs Applicative Functors

Two semantics for functors have been studied in the literature [4]. *Generative functors* are standard in Standard ML: for a functor $F$ and a module $M$, applying $F$ to $M$ twice (i.e. $F(M)$ twice) would generate modules whose signatures are *not* compatible. A canonical example for the usefulness of this semantics is the functor `SymbolTable` in Figure 5, which is taken from Dreyer's thesis [4]. This example represents a symbol table implemented with a hash table. The signature `SYMBOL_TABLE` hides a concrete type of the `symbol` and an internal hash table, and exposes two components: `string2symbol` and `symbol2string` to interconvert between `symbol` and `string`. The generative functor `SymbolTable` makes a structure sealed with `SYMBOL_TABLE`. The parenthesis `()` is used to specify a generative functor in OCaml. The components `string2symbol` and `symbol2string` access to the internal hash table `table`. The notable point lies in the implementation of `symbol2string`. The exception `Failure` should never be raised while the symbol `n` is obtained by `string2symbol` in the same structure, as the corresponding string can be found in the `table`. In generative semantics, type checking can guarantee that no exceptions will be raised. For example, assuming the structures `ST1` and `ST2` instantiated by the functor `SymbolTable`, `ST1.symbol` is not equal to `ST2.symbol`. Thus, a symbol obtained by `ST2.string2symbol` is never given to `ST1.symbol2string`.

On the other hand, *applicative functors* are standard in OCaml: applying $F$ to $M$ twice would always generate modules whose signatures are compatible. The functor `MakeSet` in Figure 3 is appropriate for applicative semantics. For example, assuming two structures generated by the same functor application `MakeSet(IntEq)`, since they have the same type and equality function for integers, there is no reason to distinguish them.

### 2.6 First-Class Modules vs Second-Class Modules

In ML, the module language exists in a separate layer from the core language for expressions, and hence modules are *second-class* objects. There is an extension of ML that treats modules as ordinary values, which are called *first-class modules*. First-class modules allow one to dynamically dispatch a module with conditional expressions and define a function that takes a module and returns it. Functions over first-class modules need to have generative semantics in the sense that the modules returned by applying such functions to first-class modules must have fresh signatures, that cannot be equal to other signatures.

OCaml (MetaOCaml) supports both first- and second-class modules. Second-class modules are *packed* into first-class modules and *unpacked* from first-class modules. OCaml uses the syntax `(module m:S)` to pack the module `m` with the signature `S` into a value of type `(module S)`, and the syntax `(val m)` unpacks `m` to a module. Components inside first-class modules can only be accessed via unpacked modules.

```
module type SYMBOL_TABLE =
  sig
    type symbol
    val string2symbol: string → symbol
    val symbol2string: symbol → string
  end
module SymbolTable (): SYMBOL_TABLE =
  struct
    type symbol = int
    let table =
      (* allocate internal hash table
           *)
      Hashtbl.create initial_size
    let string2symbol x =
      (* lookup (or insert) x *)
    let symbol2string n =
      match Hashtbl.find table n with
      | Some x → x
      | None   →
         raise (Failure "bad symbol")
  end
module ST1 = SymbolTable ()
module ST2 = SymbolTable ()
```

**Figure 5.** Symbol Table

## 3 Examples of Module Generation

This section shows concrete examples of module generation in the languages for applicative functors with second-class modules, and the one for generative functors with first-class modules. We will show different examples for each language, which reveals the merits of each language.

### 3.1 Examples with Applicative Functors and Second-Class Modules

We show examples in the language with applicative functors and second-class modules, which is called $\lambda^{<A>}$ ($A$ for applicative).

Our first example is `MakeSet`, the standard example for ML-modules. To manipulate and generate code of modules, $\lambda^{<A>}$ provides two multi-stage constructors in addition to the constructors added to Watanabe et al.'s language. To illustrate these constructors, we first give an example of the `MakeSet` program in $\lambda^{<A>}$ (Figure 6). Following Watanabe et al.'s language, our $\lambda^{<A>}$ provides a constructor \$ to extract code of a component from code of a module. For instance, if `Eq` is code of a module that has a value component `eq` of type $\tau$, then (\$`Eq`)`.eq` is the code of this value component of type $\tau$ **code**. Similarly, we can extract the type component: suppose `Eq` is code of a module that contains a type component `t = int`, (\$`Eq`)`.t` refers to the type `int`.

```
module MakeSet =
  functor (Eq : EQ mcod) →
    ⟪(struct
      type elt_t = $Eq.t
      type set_t = elt_t list
      let rec member elt set =
        match set with
        | []        → false
        | hd :: tl  → ~($Eq.eq) elt
             hd || member elt tl
    end: SET)⟫
module IntEq =
  ⟪(struct
    type t = int
    let eq = (=)
  end: EQ)⟫
module IntSet = Runmod(MakeSet(IntEq)
    : SET)
```

**Figure 6.** MakeSet Functor in $\lambda^{<A>}$

```
module MakeSet =
  functor (Eq : EQ') →
  (struct
   type elt_t = Eq.t
   type set_t = elt_t list
   let member = genlet
     <let rec member elt set =
      match set with
      | []     → false
      | hd::tl→ ~(Eq.eq) elt hd
                || member elt tl
     in member>
  end: SET')
module IntEq =
  (struct
    type t = int
    let eq = genlet <(=)>
  end: EQ')
module IntSet =
  struct
    module X = MakeSet(IntEq)
    type elt_t = X.elt_t
    type set_t = X.set_t
    let member = Runcode.run X.member
  end
```

**Figure 7.** MakeSet Translated from $\lambda^{<A>}$ to MetaOCaml

The key point in $\lambda^{<A>}$ is to distinguish code of modules from code of core expressions to avoid expressions that cannot be translated. To do so, we introduce a type **mcod**, brackets ⟪⟫, and an escape ≈, for code of modules. For example, assuming a structure $m$ has the type $M$, $\langle\!\langle m\rangle\!\rangle$ has the type $M$ **mcod**. Furthermore, if $X$ is bound to $\langle\!\langle m\rangle\!\rangle$, then $X$ can be spliced into other code of a module such as $\langle\!\langle ...(\approx X)...\rangle\!\rangle$. Continuing with the example in Figure 6, `MakeSet` has the type `functor(Eq: EQ mcod) -> SET mcod`, and `IntEq` has the type `SET mcod`. The result of the functor application `MakeSet(IntEq)` is given to `Runmod`, then `IntSet` has the type `SET`.

Figure 7 shows a program translated from Figure 6, where the definitions of `SET'` and `EQ'` are omitted. Through the translation, constructors for module generation are eliminated as the `genlet` primitive is inserted into the body of the component `member`. The role of `genlet` in the latest MetaOCaml[6] is to perform let-insertion to share generated code. Sato et al. [15] proposed to use `genlet` to avoid code explosion.

Our second example shows the usefulness of applicative functors. We borrow an example from Leroy's paper [11], which implements dictionaries by a functor `MakeDict`:

```
module type DICT =
  sig
    type key
    type 'a dict
    val empty: 'a dict
    val add: key→'a→'a dict→'a
        dict
    val find: key→'a dict→'a
  end
module MakeDict =
  functor(Key: EQ) →
    (struct
      type key = Key.t
      type 'a dict = (key*'a) list
      ...
    end: DICT)
```

The parameter `Key` of `MakeSet` is the key of this dictionary. We can extend the functor with the operation `domain` that returns the set of keys of a dictionary. The simplest way is to make a module for the set of keys inside the functor `MakeDict` using the functor `MakeSet`:

```
module MakeDict =
 functor(Key: EQ) →
  (struct
    ...
    module KeySet = MakeSet(Key)
    let domain dict=..KeySet.member..
  end: DICT)
```

To eliminate the abstraction overhead of functor applications, we can rewrite the above program in $\lambda^{<A>}$ as follows:

```
module MakeDict =
 functor(Key: EQ mcod) →
  ⟪(struct
    ...
    module KeySet= ≈(MakeSet(Key))
    let domain dict=..KeySet.member..
  end: DICT)⟫
```

[6] http://okmij.org/ftp/ML/MetaOCaml.html

Suppose functors are given the *generative* semantics, then the set of keys returned from `domain` cannot be used with other sets obtained by applying `MakeSet` to the same module for an element type. The types of their sets are incompatible. For example, we consider a functor `MakePrioQueue` implementing priority queues that use sets in the same way as `MakeDict`.

```
module MakePrioQueue =
 functor(Elt: EQ mcod) →
  ⟪(struct
     type elt = Elt.t
     type queue = elt list
     module EltSet= ≈(MakeSet(Elt))
     let contents queue =
     (* set of elements in queue *)
     ...
  end: PRIOQUEUE)⟫
```

Then, we give the module `IntEq` to the two functors:

```
module IntDict =
  Runmod(MakeDict(IntEq): DICT)
module IntPrioQueue =
  Runmod(MakePrioQueue(IntEq):
      PRIOQUEUE)
```

`IntDict` and `IntPrioQueue` contain the same set of integers, but the types of their sets are not compatible. Therefore, the following expression causes a type error.

```
IntDict.domain d =
  IntPrioQueue.contents q
```

A possible solution to this problem is to hoist `MakeSet` from `MakeDict` and `MakePrioQueue`, and to share the functor application `MakeSet(IntEq)`. In this case, `MakeDict` and `MakePrioQueue` take an extra argument for the set hoisted out, in addition to the argument `Key` (or `Elt`). Unfortunately, this solution has some problems:

- All programs that use `MakeDict` or `MakePrioQueue` require modifications to the functor arguments, even if some programs do not use the operations on the sets.
- Hoisting the functor application `MakeSet(IntEq)` to a common point requires a non-local program transformation.

In applicative semantics, there are no such problems. Therefore, we think that applicative functors are useful for module generation. Besides the above merit, since applicative functors and second-class modules are common in OCaml, existing OCaml programs can be staged with a little cost if one works in $\lambda^{<A>}$.

### 3.2 Examples with Generative Functors and First-Class Modules

In this subsection, we show examples that use generative functors and first-class modules. We call the language as $\lambda^{<G>}$ ($G$ for generative), which is a refined version of our predecessors [15, 20]. The most useful aspect of $\lambda^{<G>}$ is that a program can choose code of modules. Figure 8 shows a program where an implementation of a logger is dynamically dispatched to the main application depending on a command-line argument. In this example, there are only two choices: `consoleLogger` for printing logs to a console, or `fileLogger` for writing logs to a file.

For this setting, we may be able to generate the main application inlined for all possible combinations at compile-time, such as `consoleLogApp` and `fileLogApp`. In general, however, applications have many runtime parameters such as command-line arguments, and due to combinatorial explosion, it is difficult to generate all possible combinations at compile time. Hence, generating code of modules at runtime is useful for specializing applications which have many parameters.

Another aspect to be noted is that the abstraction overhead can be eliminated from programs suitable for generative functors such as the example of `SymbolTable` in Section 2.5. Functors represented as ordinary functions return modules with fresh abstract types.

$\lambda^{<G>}$ provides two multi-stage constructors for modules in addition to Watanabe et al.'s \$ and **run_module**. One is the type **mcod** for code of modules. The other is brackets ⟪⟫ for modules. Note that escapes for modules are not introduced because the syntax becomes too complex. Since traditional MetaOCaml can generate code of expressions, one might think that a language for generating first-class modules can be implemented as a lightweight extension to MetaOCaml. Unfortunately, it is not the case for Watanabe et al.'s translation and ours, since code of ordinary expressions and code of modules have different semantics, and they need to be distinguished.

Figure 9 shows the `MakeSet` functor expressed in $\lambda^{<G>}$. It is translated to the program in Figure 10 by our translation.

There is a trade-off between $\lambda^{<A>}$ and $\lambda^{<G>}$. In the language $\lambda^{<A>}$, the dictionary example in the previous section can be expressed. However, it does not allow a program that dynamically selects a module to be specialized, since dependencies between second-class modules are solved statically. $\lambda^{<G>}$ is opposite to $\lambda^{<A>}$. OCaml supports the two styles in a single language, and this paper is the first to propose a language extension of module generation for both styles.

## 4 Proposed Languages

We propose two languages for module generation. Due to space limitations, we present the language $\lambda^{<A>}$ here. Readers are encouraged to refer to the full version of this paper[7].

The language $\lambda^{<A>}$ is a two-stage language as an extension of core MetaOCaml plus module generation. It has ordinary simply-typed lambda terms, second-class modules, and

[7] http://logic.cs.tsukuba.ac.jp/~yuhi

```
module type LOG =
  sig
    val info: string → unit
    val error: string → unit
  end
let consoleLogger =
  《(module struct
    let print level msg =
      Printf.printf "[%s] %s\n" level
           msg
    let info msg = print "INFO" msg
    let error msg = print "ERROR" msg
  end: LOG)》
let fileLogger =
  《(module struct
    ...
  end: LOG)》
let makeApp (logger: (module LOG)
    mcod) =
  《(module struct
    let start () =
    ~($logger.info) "Start app";
    ...
  end: APP)》
let () =
  let logger =
    if Sys.argv.(1) = "console" then
       consoleLogger
    else fileLogger in
  let app = makeApp logger in
  let module App = (val (run_module
     app: APP)) in
  App.start () ;;
```

**Figure 8.** Choosing Logger Depending on Runtime Options

```
let makeSet (eq: (module EQ) mcod)=
  《(struct
    type elt_t = $eq.t
    type set_t = elt_t list
    let rec member elt set =
      match set with
      | []    → false
      | hd::tl→ ~($eq.eq) elt hd
                || member elt tl
  end: SET)》
module IntSet = (val (run_module
  (makeSet 《(module struct
    type t = int
    let eq = (=)
  end: EQ)》)))
```

**Figure 9.** MakeSet Functor in $\lambda^{<G>}$

```
let makeSet (eq: (module EQ')) =
  (struct
    module Eq = (val eq)
    type elt_t = Eq.t
    type set_t = elt_t list
    let member = genlet
      <let rec member elt set =
        match set with
        | []    → false
        | hd::tl→ ~(Eq.eq) elt hd
                || member elt tl
      in member>
  end: SET')
module IntSet = (val
  (module struct
    module S =
      (val (makeSet (module struct
        type t = int
        let eq = genlet <(=)>
      end: EQ')))
    type elt_t = S.elt_t
    type set_t = S.set_t
    let member=run S.member
  end: EQ))
```

**Figure 10.** MakeSet Translated from $\lambda^{<G>}$ to MetaOCaml

multi-stage constructors for code generation. The design of the language is based on Leroy's applicative-functor calculus [11], the classic type system $\lambda\circ$ [3], and Watanabe et al.'s calculus [20]. We confine ourselves to a minimal language to express our results. Extending our language by more features such as sharing constraints and computational effects is left for future work.

### 4.1 Syntax

Figure 11 defines the syntax for terms in $\lambda^{<A>}$. We use metavariables $m$ for module expressions, $s$ for a sequence of structure components, $c$ for structure components, $p$ for access paths, $e$ for core expressions, and $P$ for programs. Also, $x$, $t$, and $X$ are names for values, types, and modules, respectively. Duplicate component names are prohibited by typing rules. In this language, base types and primitives are unspecified. Complete programs $P$ are sequences of structure components. For simplicity, we sometimes omit **prog** and **end** in program examples.

We introduce brackets 《》, an escape ≈, and **Runmod**, for module expressions. Since the module expressions are *second class*, which means that they exist on a different layer than that of the terms. the multi-stage constructors for them should be distinguished from those for terms, which are <>, ~, and **run**. Following Watanabe et al.'s calculus, we introduce the \$ constructor to extract a component contained in code of a module as code. For example, $\$p.x$ extracts the value component $x$ as code from code of a module accessed with the path $p$, and its code can be spliced into other code. $\$p.x$ reads $\$(p).x$. A functor definition and a restriction by a signature are defined by **functor** $(X : M) \longrightarrow m$ and $(m : M)$, respectively. They use unfamiliar syntax for Camel users, but OCaml supports them. The key to applicative functors is that the access paths include a path application $p_1(p_2)$, which

$$
\begin{aligned}
m &::= X \mid p.X \mid \textbf{struct}\ s\ \textbf{end} \mid (m : M) \mid m\ (p) \\
&\quad \mid \textbf{functor}\ (X : M) \to m \\
&\quad \mid \langle\!\langle m \rangle\!\rangle \mid \approx m \ \mid \textbf{Runmod}\ (m : M) \mid \$p.X \\
s &::= \epsilon \mid c\ s \\
c &::= \textbf{let}\ x : \tau\ =\ e \mid \textbf{type}\ t\ =\ \tau \mid \textbf{module}\ X = m \\
p &::= X \mid p.X \mid p_1(p_2) \mid \$p.X \\
e &::= x \mid p.x \mid <e> \mid \sim e \mid \textbf{run}\ e \mid \$p.x \\
&\quad \mid \textbf{fun}\ x\ \to\ e \mid e\ e \mid \textbf{let}\ x\ =\ e\ \textbf{in}\ e
\end{aligned}
$$

**Figure 11.** Syntax of terms

$$
\begin{aligned}
M &::= \textbf{sig}\ S\ \textbf{end} \mid \textbf{functor}\ (X : M_1) \to M_2 \mid M\ \textbf{mcod} \\
S &::= \epsilon \mid C\ S \\
C &::= \textbf{val}\ x : \tau \mid \textbf{type}\ t \mid \textbf{type}\ t\ =\ \tau \mid \textbf{module}\ X : M \\
\tau &::= t \mid p.t \mid \tau\ \to\ \tau \mid \tau\ \textbf{code} \mid \$p.t
\end{aligned}
$$

**Figure 12.** Syntax of types

is needed to test type equality among modules obtained by functor applications. Along with this, the syntax of a functor application $m(p)$ is restricted to a path argument only.

### 4.2 Type System

Figure 12 defines the syntax of types. The syntax of types is standard except the following. We use metavariables $M$ for module types, $S$ for a sequence of signature components, $C$ for signature components, and $\tau$ for types of core expressions. We introduce the type $M$ **mcod** to distinguish code of modules from code of core expressions. The signature components include an abstract type component (**type** $t$) and a manifest type component (**type** $t\ =\ \tau$). The $\$p.t$ refers to the type component $t$ within code of a module specified with the path $p$. Our language has no explicit syntax for embedding types to code of a module, which is contrasting to Watanabe et al.'s language.

The typing environment $E$ is a possibly empty sequence of signature components $C$ annotated with a level $l$. Since $\lambda^{<A>}$ is a two-stage language, $l$ is either 0 or 1. The other typing environment $\Delta$ is a subsequence of $E$. It is used in the translation only, and explained later.

The first group of typing judgment consists of $\vdash\ P$ **wt** for well typedness of $P$, and $E; \Delta \vdash^l\ M$ **wf**, $E; \Delta \vdash^l\ S$ **wf**, $E; \Delta \vdash^l\ C$ **wf**, and $E; \Delta \vdash^l\ \tau$ **wf** for well formedness. They claim that the target types or expressions are well typed or well formed under the assumptions $E; \Delta$ at the stage $l$. We omit the definition in this paper.

The second group of judgements consist of $E; \Delta \vdash^l\ e\ :\ \tau$ for an expression $e$ of type $\tau$, $E; \Delta \vdash^l\ m\ :\ M$ for a module expression $m$ of module type $M$, and $E; \Delta \vdash^l\ s\ :\ S$ for a structure $s$ with signature $S$. Most typing rules to derive a judgment $E; \Delta \vdash^l\ e\ :\ \tau$ are standard in the type systems for MetaML-like languages [1, 18]. Nevertheless, we explain the rules for basic multi-stage constructs $<e>$, $\sim e$, and **run** $e$ shown below.

$$
\frac{E; \Delta \vdash^1\ e\ :\ \tau}{E; \Delta \vdash^0\ <e>:\ \tau\ \textbf{code}}\ (\text{E-C}) \qquad \frac{E; \Delta \vdash^0\ e\ :\ \tau\ \textbf{code}}{E; \Delta \vdash^1\ \sim e\ :\ \tau}\ (\text{E-E})
$$

The rule `E-C` assigns the type $\tau$ **code** to the code expression $<e>$ if $e$ has type $\tau$. The levels of judgments are used to distinguish level-1 expressions, such as $e$ in this rule, from level-0 expressions, such as $<e>$. The rule `E-E` is opposite to `E-C`; the 'escape' expression $\sim e$ splices the code value of $e$ into another code. For example, we can derive

$$
\cdot; \cdot \vdash^0\ \textbf{let}\ x\ =\ <3>\ \textbf{in} <\sim x + 7>: \texttt{int}\ \textbf{code}
$$

assuming that integer constants and addition are added,

We did not introduce environment classifiers [18] to keep our type system compact. As a drawback, the scope-extrusion problem may occur; the run-expression may receive an open code as the value of $e$, causing a runtime error. However, it should not be difficult to include environment classifiers in our type system.

The judgment for module expressions $E; \Delta \vdash^l\ m\ :\ M$ has the following typing rules for multi-stage constructors:

$$
\frac{E; \Delta \vdash^1\ m\ :\ M}{E; \Delta \vdash^0\ \langle\!\langle m \rangle\!\rangle\ :\ M\ \textbf{mcod}}\ (\text{M-C}) \quad \frac{E; \Delta \vdash^0\ m\ :\ M\ \textbf{mcod}}{E; \Delta \vdash^1 \approx m\ :\ M}\ (\text{M-E})
$$

The rule `M-C` assigns a type to code of a module $\langle\!\langle m \rangle\!\rangle$, and the rule `M-E` to a splice of a module $\approx m$. They are quite similar to the counterpart rules `E-C` and `E-E` for core expressions.

The following rule `M-R` assigns a type to a run expression for modules:

$$
\frac{E; \Delta \vdash^0\ m\ :\ M\ \textbf{mcod}}{E; \Delta \vdash^0\ \textbf{Runmod}\ (m : M)\ :\ M}\ (\text{M-R})
$$

Our type system has a few involved rules.

The first one is the typing rule for the \$-operator:

$$
\frac{E; \Delta \vdash^0\ p\ :\ (\textbf{sig}\ S_1\ (\textbf{val}\ x : \tau)\ S_2\ \textbf{end})\ \textbf{mcod}}{E; \Delta \vdash^0\ \$p.x\ :\ \tau'\ \textbf{code}}\ (\text{E-DotCode})
$$

To understand the rule, assume $\tau' = \tau$. Then, the rule says that given $p$ of a module-code type and $x$ has type $\tau$ in the module, $\$p.x$ has the type $\tau$ **code**. This reflects our intention that the \$-operator converts code of a module to a module consisting of code values. In the precise formulation, $\tau'$ is $\tau[z \leftarrow \$p.z \mid z \in \textbf{Dom}(S_1)]$ **code** which involves a substitution; see Leroy's calculus [11].

The second involved rule is the rule M-Strengthening to derive the equivalence of module types:

$$
\frac{E; \Delta \vdash^0\ p\ :\ M}{E; \Delta \vdash^0\ p\ :\ M/p^0}\ (\text{M-Strengthening})
$$

The *strengthening* operation replaces abstract type components with manifest type components with a path. For instance, assuming a module `A` has type `sig type t end`, this operation translates its type to `sig type t = A.t end`. Also, assuming the result type of functor application (path application) `F(A)` is `sig type t end`, its strengthened type is `sig type t = F(A).t end`. Intuitively, this operation gives a module type an identity. We use a notation $M/p^l$, which is based on Leroy's style [11], to strengthen the module type $M$ with the path $p$ at the level $l$. The level $l$ in $M/p^l$ is for the operation / rather than the path $p$, which plays the role of a flag that indicates whether it is inside **mcod**.

We emphasize that strengthening is needed for the applicative semantics only, which complicates the formal development, compared with the generative semantics. In the multi-stage languages, we need to consider the case when the path $p$ contains the \$-operator.

### 4.3 Examples of Typing Derivation

We explain how the types for the modules and the functor in Figure 6 is derived.

Let $E = (\texttt{Eq} : \texttt{EQ}\ \textbf{mcod})^0$ where `EQ` is the signature defined in Figure 3. The \$-operator turns code of a module (EQ) to a module consisting of code, hence we have:

$$\dfrac{\dfrac{\dfrac{}{E; \cdot \vdash^0 \texttt{Eq} : \texttt{EQ}\ \textbf{mcod}}}{E; \cdot \vdash^0 \texttt{\$Eq.eq} : (t \rightarrow t \rightarrow \texttt{bool})\ \textbf{code}}}{\dfrac{E; \cdot \vdash^1 \sim(\texttt{\$Eq.eq}) : t \rightarrow t \rightarrow \texttt{bool}}{\vdots \atop E; \cdot \vdash^1 \texttt{member} : \texttt{elt\_t} \rightarrow \texttt{elt\_t}\ \texttt{list} \rightarrow \texttt{bool}}}$$

In the last step of the above derivation, we used several standard typing rules for terms. We also have all the type components of `MakeSet` are suitably typed, and together with the above derivation, we can derive the following type:

$$\dfrac{\dfrac{E; \cdot \vdash^1 \textbf{struct type}\ \texttt{elt\_t = \$Eq.t...}\ \textbf{end} : \texttt{SET}}{E; \cdot \vdash^1 (\textbf{struct type}\ \texttt{elt\_t = \$Eq.t...}\ \textbf{end} : \texttt{SET}) : \texttt{SET}}}{E; \cdot \vdash^0 \langle\!\langle (\textbf{struct type}\ \texttt{elt\_t = \$Eq.t...}\ \textbf{end} : \texttt{SET}) \rangle\!\rangle : \texttt{SET}\ \textbf{mcod}}$$

Then, we can derive $\cdot; \cdot \vdash^0$ `MakeSet : EQ_SET_FUN` where `EQ_SET_FUN` is **functor** (`Eq : EQ` **mcod**) $\rightarrow$ `SET` **mcod**.

Similarly, we can derive the type for `IntEq` in Figure 6 as $\cdot; \cdot \vdash^0$ `IntEq : EQ` **mcod**. By combining them, we can derive the type for `IntSet` as follows:

$$\dfrac{\dfrac{\cdot; \cdot \vdash^0 \texttt{MakeSet} : \texttt{EQ\_SET\_FUN} \quad E; \cdot \vdash^0 \texttt{IntEq} : \texttt{EQ}\ \textbf{mcod}}{\cdot; \cdot \vdash^0 \texttt{MakeSet(IntEq)} : \texttt{SET}\ \textbf{mcod}}}{\cdot; \cdot \vdash^0 \textbf{Runmod}\ (\texttt{MakeSet(IntEq)} : \texttt{SET}) : \texttt{SET}}$$

## 5 Translation to MetaOCaml

We define a translation from $\lambda^{<A>}$ to plain MetaOCaml, while we omit the one from $\lambda^{<G>}$ to plain MetaOCaml, which is much simpler than the former.

We first show an example of the translation. Consider the following program written in $\lambda^{<A>}$.

```
module A = 《 struct
  type t = int
  let x:t = 1
  module B = struct
    val y:t = 2
  end
end 》
module A' = Runmod(A : sig
  type t = int
  val x:t
  module B: sig
    val y:t
  end
end)
```

They are translated as follows:

```
module A = struct
  type t = int
  let x:t code = <1>
  module B = struct
    val y:t code = <2>
  end
end
module A' = struct
  module X = A
  type t = X.t
  let x:t = run X.x
  module B = struct
    module Y = X.B
    let y:t = run Y.y
  end
end
```

In the translation, code of a module is translated to a module, and the type of its value components such as `x` translates to a code type, for instance, `int` is translated to `int code`. On the other hand, the type components do not change, for instance, the abstract type `t` is equal to `int` before and after the translation. The translation is applied recursively if code of a module being translated has nested modules (e.g. `B`). For **Runmod**$(A : S)$, our translation expands all the components of $A$, and re-constructs a new module $A'$ by collecting them after adding the `run`-primitive to the value components such as `run X.x`.

### 5.1 Translation Rules

We explain the key rules of our translation. For a type $\tau$ and a level $l$, the translated type $[\![\ \tau\ ]\!]^l$ is defined as follows:

$$[\![\ t\ ]\!]^l = t$$

$$[\![\ p.t\ ]\!]^l = [\![\ p\ ]\!]^l.\ t$$

$$[\![\ \tau_1 \rightarrow \tau_2\ ]\!]^l = [\![\ \tau_1\ ]\!]^l \rightarrow [\![\ \tau_2\ ]\!]^l$$

$$[\![\ \tau\ \textbf{code}\ ]\!]^0 = [\![\ \tau\ ]\!]^0\ \textbf{code}$$

$$[\![\ \$p.t\ ]\!]^l = [\![\ p\ ]\!]^l.\,t$$

Essentially, types remain the same through the translation except that the symbol $ is eliminated (in the last rule).

The translation for terms is denoted by $[\![\ \cdot\ ]\!]^l_\Delta$, which is parameterized by the level $l$ (for $l = 0, 1$) and the environment $\Delta$ explained later. The key rule for value components is shown below:

$$\begin{aligned}&[\![\ \mathbf{let}\ x : \tau\ =\ e\ ]\!]^1_\Delta \\ &\qquad = \mathbf{let}\ x : [\![\ \tau\ ]\!]^1\ \mathbf{code}\ =\ \mathbf{genlet}\ <[\![\ e\ ]\!]^1_\Delta >\end{aligned}$$

A value component at the level 1 is in a code of a module, hence we change its type to a code type, and add the `genlet` primitive to avoid the code-duplication problem.

For expressions, the translation is homomorphic for most cases except the following rules:

$$[\![\ x\ ]\!]^1_\Delta = \begin{cases}\sim x & (x \in \mathrm{Dom}(\Delta)) \\ x & (otherwise)\end{cases}$$

$$[\![\ p.x\ ]\!]^1_\Delta = \begin{cases}\sim([\![\ p\ ]\!]^1.\,x) & (\mathrm{head}(p) \in \mathrm{Dom}(\Delta)) \\ [\![\ p\ ]\!]^1.\,x & (otherwise)\end{cases}$$

$$[\![\ \$p.x\ ]\!]^0_\Delta = [\![\ p\ ]\!]^0.\,x$$

In the first rule, $\Delta$ is the set of variables bound in the same module (which appears as a code of the module), hence we must add an escape (splice) to the variable after the translation. Similarly for the second rule. In the last rule, we unconditionally eliminate $ from the paths.

The rules for $[\![\ m\ ]\!]^l_\Delta$ for a module expression $m$ is given as follows:

$$[\![\ \langle\!\langle m \rangle\!\rangle\ ]\!]^0_\Delta = [\![\ m\ ]\!]^1_\Delta$$

$$[\![\ \approx m\ ]\!]^1_\Delta = [\![\ m\ ]\!]^0_\Delta$$

$$\begin{aligned}[\![\ \mathbf{Runmod}\ (m : \mathbf{sig}\ S\ \mathbf{end})\ ]\!]^0_\Delta = \mathbf{struct}&\\ &\mathbf{module}\ X = [\![\ m\ ]\!]^0_\Delta \\ &S\ /\!\!/\ X \\ \mathbf{end}&\end{aligned}$$

$$[\![\ \$p.X\ ]\!]^0_\Delta = [\![\ p\ ]\!]^0.\,X$$

The first rule translates code of a module to a module, but the stage is raised to 1, reflecting the fact that $m$ is in the code. The next rule eliminates the escape for code of a module. The third rule translates a runmod-term which is complicated. Its result depends on the signature $S$ of the target module since we need to build a new module. The new module contains a nested module with a fresh name and components. The expression $S\ /\!\!/\ X$ when $S$ is a value component is defined as:

$$((\mathbf{val}\ x : \tau)\ S)\ /\!\!/\ X = (\mathbf{let}\ x : \tau\ =\ \mathbf{run}\ X.x)\ S\ /\!\!/\ X$$

which retrieves the $x$-component from the module $X$, runs it, and binds $x$ (in the new module) to its result. The **run**-primitive is propagated to nested modules shown below:

$$\begin{aligned}&((\mathbf{module}\ X' : M)\ S)\ /\!\!/\ X \\ &\ = (\mathbf{module}\ X' = [\![\ \mathbf{Runmod}\ (X.X' : M)\ ]\!]^0_\epsilon)\ S\ /\!\!/\ X\end{aligned}$$

Other module expressions are kept intact and the environment $\Delta$ is not important for now. We omit the other clauses of the translation.

### 5.2 Translation Preserves Typing

We can prove that the following form of simple type preservation holds for the translation. If $E; \Delta \vdash^0 e\ :\ \tau$ is derivable in our type system, then $[\![\ E\ ]\!]_\Delta \vdash^0\ [\![\ e\ ]\!]^0_\Delta\ :\ [\![\ \tau\ ]\!]^0$ is derivable. We assume the following two: First, the target language is a subset of MetaOCaml, which is the same as our language without multi-stage constructors for modules. However, the type system in the target language is defined without the second environment $\Delta$, which is obtained by simply removing $\Delta$ from our type system. Second, the target type system has a typing rule for genlet. The typing rule for genlet is defined below.

$$\frac{E \vdash^0\ e\ :\ \tau\ \mathbf{code}}{E \vdash^0\ \mathbf{genlet}\ e\ :\ \tau\ \mathbf{code}}$$

**Theorem 5.1.** *If $E; \Delta \vdash^l\ e_0\ :\ \tau_0$ is derivable in $\lambda^{<A>}$, then $[\![\ E\ ]\!]_\Delta \vdash^l\ [\![\ e_0\ ]\!]^l_\Delta\ :\ [\![\ \tau_0\ ]\!]^l$ is derivable.*

The proof is obtained by simply applying the standard technique of proving type preservation. The most difficult part is to formulate the typing rules and translation rules correctly in the presence of applicative functors and generation of code and modules.

## 6 Implementation and Performance

Although the main purpose of this paper is to give a solid type-theoretic foundation to the languages for module generation, the performance of our language is not irrelevant, as the main objective of module generation is to eliminate the overhead of functor applications (or module boundary).

We have implemented our languages using the translations in the previous section and compared the performance of our implementation with those of the existing work. We take a benchmark program used in existing studies such as Watanabe et al. [20]. It implements simplification rules for arithmetic expressions in a domain-specific language (DSL). The DSL is embedded in OCaml by the tagless-final embedding [2], which extensively uses the module system. Simplifications for DSL expressions such as $0 + n \longrightarrow n$ and $0 \cdot n \longrightarrow 0$ are expressed by functors, and it is desirable to remove the overhead of functor applications.

The following functor `Simp` implements one of simplifications:

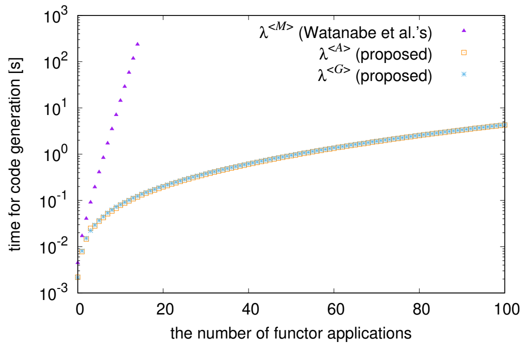


**Figure 13.** Time for Code Generation

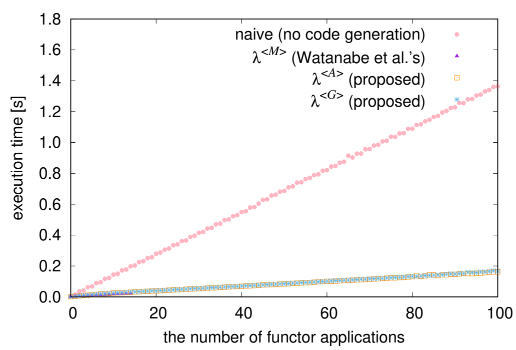


**Figure 14.** Execution Time for Generated Code

```
module Simp= functor(M : S mcod)→
⟪ (struct
  type int_t = $M.int_t * bool
  let add n1 n2 =
   match (n1, n2) with
    (x1,b1),(x2,b2) →
    if b1&&b2 then (~($M.int)0,true)
    else if b1 then (x2,false)
    else if b2 then (x1,false)
    else (~($M.add) x1 x2,false)
  ...
end : S) ⟫
```

The argument of `Simp` is code of a structure `M`, which expresses arithmetic expressions before simplification. The functor `Simp` simplifies an addition expression when at least one of its arguments is zero. To do this, the type `M.int_t` is interpreted as a pair of an arithmetic expression and a boolean flag to indicate the expression is zero or not. Since `M` has the type `S mcod`,`$M.add` refers to the code of the add-component of `M` correctly, which is spliced into the resulting code, and the overhead of an indirect function call eliminated. Our test program applies the functor `Simp` to a structure many times such as `Simp(Simp(...(M)...))`, and we measured the performance of the resulting module.

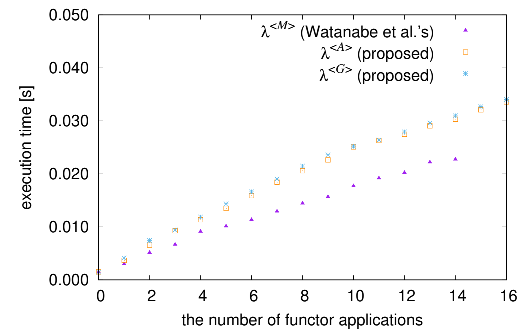


**Figure 15.** Execution Time for Generated Code (zoomed)

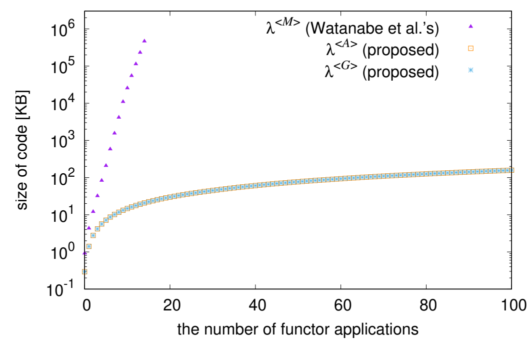


**Figure 16.** Size of Generated Code

We have measured the performance of the following four variations:

- a naïve OCaml program (no code generation).
- a MetaOCaml program with Watanabe et al.'s translation.
- a MetaOCaml program in $\lambda^{<G>}$ with our translation.
- a MetaOCaml program in $\lambda^{<A>}$ with our translation.

We did not include the result by Sato et al.'s translation, since for this benchmark the resulting program by their translation is identical to the one in $\lambda^{<G>}$ with our translation.

We have conducted the experiments on Ubuntu 18.04 LTS, Xeon E3-1225 v6@3.3GHz, Memory 32GB, BER MetaOCaml N107 (OCaml 4.07.1), byte code compiler, and all the results are the average of 10 trials. For these programs, we have measured the time for code generation and compilation (Figure 13), the execution time of generated code (Figure 14 and 15), and the size of generated code (Figure 16) where the size of code is the file size in bytes,

The results can be summarized as follows. First, all approaches based on module generation outperform the naïve program (Figure 14), thus we confirmed that it eliminates the overhead of functor applications. Second, both of our two languages have slightly better performance in the execution time than Watanabe et al.'s work, while ours outperform theirs in the other factors (the code-generation time and the size of generated code). Third, in all experiments, the generative and applicative styles have quite similar performance. Hence, and we can select a suitable style depending on our applications without worrying about the loss of performance.

## 7 Related Work

In this section, we compare several closely related works with our work.

One can say that our work provides a means to *inline* functor applications based on the programmer's control. There are fully automatic tools or systems to inline functor applications including Flambda [6] and MLton [8]. By aggressively inlining functor applications, they can eliminate redundant indirections. While automatic tools are easier to use than human-controlled tools, we think the former does not always subsume the latter, just like fully automatic partial evaluation does not always subsume programmer-controlled program generation. In particular, when we want to inline functor applications conditionally, based on domain-specific knowledge, or when we want to control the number of inlining (how much we inline the functor applications), human intervention is needed.

Squid [13] is a multi-stage programming framework for Scala, and guarantees that generated code is well-typed and well-scoped. Parreaux and Shaikhha [12] proposed a library for class generation built on top of the Squid and gave practical use cases. Unfortunately, it is difficult to simply apply their use cases to our approach. First, classes can have states, but modules without side effects cannot. Second, their library provides a way to dynamically generate fields of classes, but our language cannot. Achieving them in a type-safe way and giving large-scale practical use cases are left for future work.

The MetaML-style approach taken in this work is *purely generative*. Many studies take opposite approaches based on introspection or reflection which allow the inspection or decomposition of generated code. One of such approaches is Racket's module system [7] mentioned in Section 1. Another related work is SugarJ [5], which provides a library-based approach to module systems for model-driven development. Although their motivation is quite different from ours, there is some technical similarity in that their modules provide phase separation, and they consider transformers over modules just like functors over modules in our case.

## 8 Conclusion

In this paper, we have studied typed two-stage programming languages for generating and manipulating code of modules. We have designed two languages that allow generating modules in different styles, given a precise type-theoretic formulation, and defined a type-preserving translation to plain MetaOCaml. We have also implemented program translators based on our translations and shown that despite the complexity of the language and the type system, the module generation with applicative functors and second-class modules has almost identical performance compared with the existing studies based on generative functors and first-class modules. Thus, a programmer can choose the style without worrying about the performance.

One may wonder if realistic programs use deeply nested functor applications as is used in our benchmark program, but there exist such cases. For instance, MirageOS contains many functor applications, and its web service has functor applications of depth up to 10 [14]. The actual MirageOS may contain more indirections than this, hence we can expect that the benefit of module generation would be even greater.

There are many directions for future work. First, we want to make our languages more expressive to cover polymorphism, computational effects, and sharing constraints in modules. Second, FLambda is a fully automatic inlining system that works for functor applications, too. While we believe that human-controlled generation of modules and terms such as our work can outperform the former, unifying these approaches is doable and will be interesting from the practical aspect. Finally, ML-style modules are not the only high-level abstractions for programming languages, and there are many works in the literature that investigated the generation of high-level abstractions such as classes. By comparing our work and others, we want to improve our languages and also help improve other works.

[8] http://www.mlton.org

## Acknowledgments

We would like to thank the anonymous reviewers and the PC members of GPCE 2021 for their valuable comments. The second author is supported in part by Grant-in-Aid for Scientific Research (B) No. 18H03218.

## References

[1] Cristiano Calcagno, Eugenio Moggi, and Walid Taha. 2004. ML-Like Inference for Classifiers. In *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings (Lecture Notes in Computer Science)*, David A. Schmidt (Ed.), Vol. 2986. Springer, 79–93. https://doi.org/10.1007/978-3-540-24725-8_7

[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] 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

[4] Derek Dreyer. 2005. *Understanding and Evolving the ML Module System.* Ph.D. Dissertation. USA. AAI3166274.

[5] Sebastian Erdweg and Klaus Ostermann. 2017. A Module-System Discipline for Model-Driven Software Development. *Art Sci. Eng.*

*Program.* 1, 2 (2017), 9. https://doi.org/10.22152/programming-journal.org/2017/1/9

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

[7] Matthew Flatt. 2013. Submodules in racket: you want it when, again?. In *Generative Programming: Concepts and Experiences, GPCE'13, Indianapolis, IN, USA - October 27 - 28, 2013*, Jaakko Järvi and Christian Kästner (Eds.). ACM, 13–22. https://doi.org/10.1145/2517208.2517211

[8] Yuichiro Hanada and Atsushi Igarashi. 2014. On Cross-Stage Persistence in Multi-Stage Programming. 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, 103–118. https://doi.org/10.1007/978-3-319-07151-0_7

[9] 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

[10] 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*. 86–102. https://doi.org/10.1007/978-3-319-07151-0_6

[11] Xavier Leroy. 1995. Applicative Functors and Fully Transparent Higher-Order Modules. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 142–153. https://doi.org/10.1145/199448.199476

[12] Lionel Parreaux and Amir Shaikhha. 2020. Multi-Stage Programming in the Large with Staged Classes. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Virtual, USA) *(GPCE 2020)*. Association for Computing Machinery, New York, NY, USA, 35–49. https://doi.org/10.1145/3425898.3426961

[13] Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. 2017. Squid: type-safe, hygienic, and reusable quasiquotes. In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala, SCALA@SPLASH 2017, Vancouver, BC, Canada, October 22-23, 2017*, Heather Miller, Philipp Haller, and Ondrej Lhoták (Eds.). ACM, 56–66. https://doi.org/10.1145/3136000.3136005

[14] 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

[15] 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@POPL 2020, New Orleans, LA, USA, January 20, 2020*, Casper Bach Poulsen and Zhenjiang Hu (Eds.). ACM, 1–13. https://doi.org/10.1145/3372884.3373160

[16] Tim Sheard. 1998. Using MetaML: A Staged Programming Language. In *Advanced Functional Programming, Third International School, Braga, Portugal, September 12-19, 1998, Revised Lectures (Lecture Notes in Computer Science)*, S. Doaitse Swierstra, Pedro Rangel Henriques, and José Nuno Oliveira (Eds.), Vol. 1608. Springer, 207–239. https://doi.org/10.1007/10704973_5

[17] 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

[18] Walid Taha and Michael Florentin Nielsen. 2003. Environment Classifiers. *SIGPLAN Not.* 38, 1 (Jan. 2003), 26–37. https://doi.org/10.1145/640128.604134

[19] Walid Taha and Tim Sheard. 1997. Multi-Stage Programming. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997*, Simon L. Peyton Jones, Mads Tofte, and A. Michael Berman (Eds.). ACM, 321. https://doi.org/10.1145/258948.258990

[20] 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

[21] 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