Module Generation in Multi-Stage Programming

Yuhi Sato

(Master's Program in Computer Science)

Advised by Yukiyoshi Kameyama

Submitted to the Graduate School of
Systems and Information Engineering
in Partial Fulfillment of the Requirements
for the Degree of Master of Engineering
at the
University of Tsukuba

March 2021

Abstract

Multi-stage programming is a paradigm of program generation and has been studied intensively because it allows us to generate low-level programs specialized in run-time parameters from abstracted programs. Especially, MetaOCaml, a multi-stage programming language, guarantees type safety and scope safety for generated programs, at compile time. However, because it is difficult to guarantee those safety properties, MetaOCaml is restricted to term generation only and cannot generate modules, which are an indispensable mechanism for high-level abstractions. Watanabe et al. proposed a language for module generation and implemented it via a translation to MetaOCaml. Unfortunately, their solution has a serious problem that generated code may become exponentially large, and their language is too liberal to be translated to plain MetaOCaml.

In this thesis, we propose two multi-stage programming languages for module generation and refined translations for these languages. Our translations do not suffer from the code-duplication problem. The key idea is to use the *genlet* primitive in the 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. Our languages correspond to two module styles: first-class modules with generative functors and second-class modules with applicative functors, which are both supported by MetaOCaml and useful for modular programming. We conduct a few experiments using a microbenchmark, and the result shows that our method is effective to reduce the size of generated code to be linear.

Contents

List of Figures

Chapter 1

Introduction

Program generation has been studied intensively and applied in a wide variety of fields [1] since it can break the trade-off between productivity and performance in application developments. Especially, Multi-Stage Programming (MSP) is an effective approach to generate programs for the following reasons. First, MSP languages provide a way to generate programs specialized in run-time parameters. Second, safety of generated code is guaranteed statically (at compile time). Since metaprogramming for generating programs is generally complex and error-prone, it is worth to use systems that provide static safety. For these reasons, MSP is supported in various languages such as OCaml [2], Scala [3, 4], Haskell [5], and C# [6], and is used in various domains such as high-performance computing [7], stream processing pipelines [8], SQL query processor [9], and combinational circuits generation [10]. Unfortunately, because it is difficult to guarantee its safety beyond term generation, generating modules is not allowed in MetaOCaml, a multi-stage extension of OCaml [2].

A module system in OCaml is highly valuable for providing high-level abstraction and 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 contain thousands of modules [11]. On the research side, interesting extensions using modules have been proposed: modular implicits [12], extensible language-integrated query [13], and tagless-final embedding [14].

Inoue et al. [15] were the first to propose a language extension for generating code of modules in the MSP style. They investigated 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. [16] proposed a language $\lambda^{<M>}$ for generating and manipulating code of modules, and implemented the language by a translation to plain MetaOCaml. They have also conducted an experiment to show that the abstraction overhead in modules can be reduced. Unfortunately, their approach has some problems. First, code generated by their method can become so large that it may not compile. Second, their language allows

[1] https://mirage.io

expressions that cannot be translated into MetaOCaml.

In this thesis, we propose languages and its implementations that solve the problems of previous work to generate modules without regret. Our contributions are summarized below.

- We solve the code-explosion problem in Watanabe et al.'s approach. Our key idea is to use a MetaOCaml's primitive called *genlet* which allows performing let insertion to avoid code duplication.
- We give two refined languages for module generation: $\lambda^{<M_A>}$ and $\lambda^{<M_G>}$. The language $\lambda^{<M_A>}$ is based on second-class modules which are standard in OCaml and has functors with applicative semantics, while the language $\lambda^{<M_G>}$ is based on first-class modules and has functors with generative semantics. Our languages provide important features in modular programming such as abstract types and nested modules. All programs typed in our languages are translated into plain MetaOCaml. We prove that the translation for $\lambda^{<M_A>}$ preserves types.

The rest of this thesis is organized as follows: Chapter 2 explains background knowledge such as a module system and multi-stage programming. Chapter 3 explains the motivation for generating modules, and describes the previous work and its problems. Chapter 4 proposes our solution. Chapter 5 defines the language $\lambda^{<M_A>}$ for applicative functors and its translation to MetaOCaml. Chapter 6 defines the language $\lambda^{<M_G>}$ for generative functors and its translation to MetaOCaml. Chapter 7 shows the results of experiments on microbenchmark. Chapter 8 discusses our work. Chapter 9 states the related work. Chapter 10 gives conclusion and future work.

Chapter 2

Background

In this chapter, we state the background of our study. First, we explain a module system in OCaml. Second, we introduce program generation in Multi-Stage Programming.

2.1 Module System in OCaml

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 safe way that achieves reusability and maintainability. This section illustrates the basics and key features of modules with several examples.

2.1.1 Structures

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

Figure 2.1 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`. 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`. In this thesis, structures are sometimes called modules if there is no confusion.

2.1.2 Signatures

Signatures correspond to a specification 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 maintainability of programs.

Continuing with the example in Figure 2.1, 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

```
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 2.1: Structure for Integer Set

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

Figure 2.2: Signature for Integer Set

signature `SET` for the structure `IntSet` as shown in Figure 2.2. 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 defined as *abstract types* that hide an implementation of corresponding type components. On the other hand, *manifest types* expose them. The function `member` takes values of types `elt_t` and `set_t`, and returns a value of type `bool`. The structure `IntSet` can be *sealed* with the `SET` by `module IntSet' = (IntSet : SET)`, and the resulting module forgets the equivalence of `set_t = int list`.

2.1.3 Functors

Functors are modules parameterized by modules and correspond to functions over modules, which achieve reusability. Figure 2.3 shows an example of a functor that makes a module of sets. The functor `MakeSet` is parameterized by the module `Eq` with the signature `EQ`. The `Eq` contains the type `t` of an element of sets and the equality function `eq` on its elements. Using the `MakeSet`, the structure `IntSet` in Figure 2.1 can be defined as shown in Figure 2.4. The structure `IntEq` contains the concrete components for integers, which is sealed with `EQ`. By applying `MakeSet` to `IntEq`, we obtain the structure `IntSet`. If we need a set of strings, we implement the structure `StringEq` that defines a type of an element and an equality function for string, we only have to apply `MakeSet` to `StringEq`.

2.1.4 Generative Functors vs Applicative Functors

Two semantics for functors have been studied in the literature [17]. *Generative functors* are standard in SML: for a functor F and a module M, applying F to M twice would generate modules whose signatures are *not* compatible. A canonical example for the usefulness of

```
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 2.3: 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 2.4: Integer Set and String Set by Functor

this semantics is the functor `SymbolTable` given in Figure 2.5, which is taken from Dreyer's thesis [17]. 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 functions `string2symbol` and `symbol2string` to interconvert between `symbol` and `string`. The generative functor `SymbolTable` makes a structure sealed with `SYMBOL_TALBE` in which `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`, since `ST1.symbol` is not equal to `ST2.symbol`, 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

```
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 2.5: Symbol Table

would always generate modules whose signatures are compatible. The functor `MakeSet` in Figure 2.3 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.

Each style of functors has its own merit, and therefore the latest OCaml (and MetaOCaml) have both.

2.1.5 First-Class Modules vs Second-Class Modules

The module language exists on a separate layer from the language for expressions. The module language is *second-class*, while extensions for treating modules as values in the language for expressions are called *first-class modules*. First-class modules allow us to dynamically dispatch a module with conditional expressions and define a function that takes a module and returns it. Functions over first-class modules have generative semantics in the sense that a function returns a module with a fresh signature.

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.

2.2 Multi-Stage Programming

Multi-Stage Programming (MSP) [18] is a paradigm for runtime program generation in which programmers can write a code generator to generate a program specialized into a particular application domain and run-time parameters. In MSP, *code* means program fragments that can be manipulated programmatically. MetaOCaml is an extension of OCaml that supports MSP, which provides four multi-stage constructors below.

Generating code: $< e >$
: Brackets $< e >$ generate code of the expression e. When the type of e is τ, the type of $< e >$ is τ **code**.

Splicing code: $\sim e$
: Escape $\sim e$ exempts the expression e from the brackets. That is, the escape allows code to be spliced into another code. For example, let x be $< 1 + 2 >$, an expression $<\sim x + 3 >$ is evaluated to $< (1 + 2) + 3 >$.

Executing code: **run** e
: The expression **run** e compiles a value of the code e and executes it. For example, the result of evaluating **run** $< 1 + 2 >$ is 3. Precisely, MetaOCaml provides **run** as the primitive `Runcode.run`.

Cross-Stage Persistence (CSP)
: CSP (Cross-Stage Persistence) is a feature for embedding a present-stage value into a future-stage code. MetaOCaml performs CSP for variables implicitly.

2.2.1 Example of Code Generation in MetaOCaml

We explain program optimization by MetaOCaml using an example of a power function that calculates x^n. Figure 2.6 shows the naive implementation of a power function in OCaml. The comment shows the result of an expression returned by the OCaml (MetaOCaml) top-level, where the type and the value are shown. Given integers `n` and `x`, the function `power` makes `n` recursive calls to compute x^n. Due to an overhead of function calls, the function `power` is slower than an expression that calculates $x * x * \cdots * x$.

Figure 2.7 shows a program that generates a power function specialized to n and executes it. The function `spower` is a staged function of `power`, which takes `n` of type `int` and `x` of type `int code`, and generates code of type `int code`, where `n` is a static parameter and `x` is a dynamic parameter. If `n` equals 0, `spower` generates code of `1`, otherwise, the result of the recursive call is spliced into code. The `pow3_code` builds code that calculates $x * x * x$ without recursive calls. Running this code using the primitive `run` yields a function `pow3` of type `int -> int`. The execution of `pow3 2` calculates $2 * 2 * 2 * 1$ internally, therefore the performance problem is solved.

For the sake of readability, we use the multi-stage constructors without dots such as $< e >$ and $\sim e$, but MetaOCaml in the real world uses constructors with dots such as $. < e > .$ and $. \sim e$.

```
let rec power n x =
    if n = 0 then 1
    else x * (power (n-1) x)
(* val power : int -> int -> int = <fun> *)
```

Figure 2.6: Power Function

```
let rec spower n x =
    if n = 0 then <1>
    else <~x * ~(spower (n-1) x)>
(* val spower : int -> int code -> int code = <fun> *)

let pow3_code = <fun x -> ~(spower 3 <x>)>
(* val pow3_code : (int -> int) code =
    <fun x_1 -> x_1 * (x_1 * (x_1 * 1))>  *)

let pow3 = Runcode.run pow3_code
(* val pow3 : int -> int = <fun> *)

let result = pow3 2
(* val result : int = 8 *)
```

Figure 2.7: Generating Power Function Specialized to $n = 3$

Chapter 3

Module Generation

In this chapter, we state motivation for module generation and previous work. We also illustrate problems in the previous work that we will address.

3.1 Motivation

Traditional MSP languages such as MetaOCaml do not support module generation. We argue that module generation in MSP is an important issue for the following reasons. First, modules obtained by applying functors involve indirections through a run-time representation. This indirection is a run-time overhead, which is especially serious in real-world applications. For example, the unikernel that runs the MirageOS website contains a functor application depth of up to 10 [11]. We can remove such indirections via generating modules inlined. Second, in MSP, programmers are responsible for optimizations such as inlining. As inlining does not always improve performance, it is important that programmers can control optimizations on modules. Also, programmers can use non-trivial domain knowledge to generate aggressively optimized modules. We can also easily specialize ordinary functions into multiple parameters via modules. Third, generating code of first-class modules allows us to choose which code of a module is given to a functor depending on run-time parameters.

3.2 Previous Work

Generating modules beyond terms in MSP has been started by Inoue et al. [15]. They proposed an extension of core MetaOCaml that allows module generation and demonstrated that the extension may eliminate overhead of functor applications. In addition, they proposed a way to represent the extension in plain MetaOCaml by representing a module as a polymorphic record. Unfortunately, they have not formulated the extension nor shown its implementation.

Watanabe et al. [16] gave the language $\lambda^{<M>}$, an extension of core MetaOCaml, that allows generating and manipulating code of first-class modules. The language was implemented by a translation to plain MetaOCaml. We explain their proposal and problems

```
let makeSet (eq: (module EQ) code) =
  <(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 3.1: MakeSet Functor in $\lambda^{<M>}$

using program examples.

Figure 3.1 shows a $\lambda^{<M>}$ program implementing the `MakeSet` and the `IntSet` shown in Figure 2.3 and 2.4. The functor[1] `makeSet` receives code of a module and returns a new code of a module. $\lambda^{<M>}$ provides three new constructors `$`, `%` and `run_module` to manipulate a code of modules. The constructor `$` extracts code of a component from code of a module. In this example, `$eq.eq` means that extract the code of the function `eq` from the code of the module `eq`. Thus, $\sim$`($eq.eq)` is replaced with the function `(=)`, and the abstraction overhead is eliminated. The constructor `%` is CSP for types, if `$eq.t` has a type of `int code`, then `%($eq.t)` become a type of `int`. Although MetaOCaml implicitly performs CSP, their language uses explicit notation. Finally, `run_module` executes code of a module and returns its module. In this example, `run_module` receives a module of type `(module SET) code` and returns a module of type `(module SET)`. Since $\lambda^{<M>}$ uses first-class modules, packaging `(module ...)` and unpackaging `(val ...)` are used to interconvert between first- and second-class modules.

The main role of the translation from $\lambda^{<M>}$ to MetaOCaml is to eliminate outer brackets of modules. Their idea is to turn code of a module into a module containing code. Since a module has a record-like structure, they regarded that these two are isomorphic. Figure 3.2 shows the `makeSet` and the `IntSet` translated from Figure 3.1. The type of `makeSet` is translated from `(module EQ) code -> (module SET) code` to `((module EQ') -> (module SET')) code`, where the signature `EQ'` is the result of translation from `EQ` and `SET'` is from `SET`. By the translation, brackets outside modules move to the body of value components, while type components are kept intact. The constructors added to $\lambda^{<M>}$ are eliminated by the translation, and we can execute the program shown in Figure 3.2 on MetaOCaml.

Watanabe et al. measured performance against microbenchmarks, and demonstrated that performance can be improved by generating code for modules.

[1]Since Watanabe et al. uses first-class modules, functors are represented by normal functions

```
module type SET' =
  sig
    type elt_t
    type set_t
    val member: (elt_t -> set_t -> bool) code
  end

module type EQ' =
  sig
    type t
    val eq: (t -> t -> bool) code
  end

let makeSet (eq: (module EQ')) =
  (struct
    module Eq = (val eq)
    type elt_t = Eq.t
    type set_t = elt_t list
    let member =
      <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 = <(=)>
      end: EQ')))
    type elt_t = S.elt_t
    type set_t = S.set_t
    let member = Runcode.run S.member
  end: EQ))
```

Figure 3.2: MakeSet Functor Translated from $\lambda^{<M>}$ to MetaOCaml

```
module type S =
  sig
    val x1: int
    val x2: int
    val x3: int
  end

let m =
  <(module struct
    let x1 = 1 + 2
    let x2 = x1 + x1
    let x3 = x1 * x1
  end : S)>
```

```
module type S' =
  sig
    val x1: int code
    val x2: int code
    val x3: int code
  end

let m =
  (module struct
    let x1 = <1 + 2>
    let x2 = <let x1 = 1 + 2 in
              x1+x1>
    let x3 = <let x1 = 1 + 2 in
              let x2 = x1+x1 in
              x1*x1>
  end : S')
```

Figure 3.3: $\lambda^{<M>}$ Program With Dependencies (Left) and Its Translated Program (Right)

```
let f (m : (module S) code) =
  <(module struct
    let y =
      ~($m.x3) + ~($m.x3)
  end : T)>
```

```
let f (m : (module S')) =
  (module struct
    module M = (val m)
    let y =
      <~(M.x3) + ~(M.x3)>
  end : T')
```

Figure 3.4: $\lambda^{<M>}$ Program of Functor (Left) and Its Translated Program (Right)

3.2.1 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 Figure 3.3. The program on the left-hand side is translated to the right-hand side. The problem here is that the let-binding `x2` is defined in the value component `x3` 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 number of value components.

Furthermore, the situation becomes worse when applying a functor to the module `m`. Let us consider defining the functor `f` as shown in Figure 3.4 and applying `f` to `m`. The component `y` in the functor has two references to the component `x3` in the module `m` passed as an argument. `T` and `T'` are appropriate signatures. Figure 3.5 shows code generated by the functor application `f m`. The problem in this code is that the expression `(let x1 = ... in x1 * x1)` is defined twice. An ideal code would define a let-binding for its expression locally and dereference the let-bound variable twice. One might think that `m.x3` should be shared in the functor `f` using a let-binding, but its implementation is not natural in a case that the part of `m.x3` is a function like the function `eq` in the `MakeSet` example.

```
# let module M = (val (f m)) in M.y;;
- : int code = <
  (let x1 = 1 + 2 in let x2 = x1 + x1 in x1 * x1) +
  (let x1 = 1 + 2 in let x2 = x1 + x1 in x1 * x1)>
```

Figure 3.5: Duplicated Code by Functor Application

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

```
let x = 10 + 20 in
  (module struct
    let y = <1>
  end : T')
```

Figure 3.6: Violating Semantics in Staging (Left Is Translated to Right)

In the worst case, the code size increases exponentially in the number of functor applications. In the example of Figure 3.5, as the component `m.x3` was referenced twice, the size of the generated code is (approximately) doubled. If the component `y` is referenced twice in another functor, the size of the code would be (approximately) four times as large as the original one. 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. Since the code is generated before compilation, compiler optimization is not useful. 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.

The next chapter will describe our translations that solve this problem.

3.2.2 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 example shown in Figure 3.6. The left-hand side is translated to the right-hand side. `T` and `T'` is appropriate signatures. The subexpression `10+20` is executed at the future stage on the left, while it is executed at the present stage on the right, violating the distinction of stages.

Second, their language did not allow abstract types and nested modules. These are essential features for modular programming. As we mentioned in the previous chapter,

abstract types conceal implementations and improve the maintainability of modules. On the other hand, nested modules are needed to develop applications by assembling modules.

Third, since their language is based on first-class modules, the semantics of functors is generative. As explained in Section 2.1.4, not only generative functors, but applicative functors are also useful. A motivation for generating first-class modules with generative functors and second-class modules with applicative functors will be described in the next chapter.

Chapter 4

Our Proposal

We introduce a refined translation for languages with module generation to solve the code-explosion problem described in the previous chapter. Our translation performs dynamic let-insertion, which allows code fragments to be shared among different components. Also, we propose two languages implemented by our translation: $\lambda^{<M_A>}$ and $\lambda^{<M_G>}$. The language $\lambda^{<M_A>}$ has functors with applicative semantics and allows generating code of second-class modules. On the other hand, the language $\lambda^{<M_G>}$ has generative functors and allows generating code of first-class modules. In this chapter, we explain how our translation work using examples, and describe designs and concrete examples of the languages. The formal definitions of the languages and their translations will be given in the next chapter and beyond.

4.1 Sharing Code Fragments Using let-Insertion

Let-insertion is a well-known technique for code sharing in program transformation (partial evaluation, in particular) [19]. It can be implemented in various ways, and here we review the two most relevant approaches for let-insertion.

4.1.1 Static let-Insertion by shift and reset

The first approach uses the delimited-control operators *shift* and *reset* [20], 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 [21]. 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 runtime. 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 [22]. 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 written in $\lambda^{<M>}$:

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

where S is a suitable signature. Watanabe et al.'s translation removes 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 (function) `f` to the `mcod0`:

```
let f (mcod: (module S')) =
  (module struct
    module M = (val mcod)
    let y = <~(M.x1) + ~(M.x2)>
  end : T')
```

where `T'` is an appropriate signature, and note that the component `y` refers to two components `x1` and `x2` (as opposed to the example in Figure 3.4). The result of a functor application `f mcod0` is the following module:

```
(module struct
  let y = <(10 + 20) + (let t = 10 + 20 in t + t)>
end : T')
```

For the sake of simplicity, nested module `M` is omitted. 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 y = <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.

4.1.2 Dynamic let-Insertion by genlet

The second way uses the genlet primitive [23] in MetaOCaml[1] 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 and 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')
```

where `genlet` is used twice. 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 component `x1`, we get `<let t1 = 10 + 20 in t1>` as its value. The result of the execution of the component `x2` 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 the `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. [2] For the

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

[2] Our explanation here is essentially due to Kiselyov's explanation for genlet, available from the BER MetaOCaml repository on GitHub, `https://github.com/metaocaml/ber-metaocaml/blob/ber-n111/ber-metaocaml-111/patches/trx.ml`.

`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 or compiled), 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 expression `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 thesis 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 expression `f mcod1`, which simulates a functor application using first-class modules. When we evaluate the expression, again the dynamic let-insertion triggered by genlet in `mcod1` is delayed until the result of the whole expression 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 y = <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 thesis.

4.2 $\lambda^{<M_A>}$: Extension to Applicative Functors and Second-Class Modules

We propose a language $\lambda^{<M_A>}$ for generating and manipulating code of second-class modules in applicative semantics. The language $\lambda^{<M_A>}$ is implemented by a translation using genlet presented in the previous section. Programs written in $\lambda^{<M_A>}$ are translated to plain MetaOCaml and are executed on MetaOCaml. This section introduces the language $\lambda^{<M_A>}$ through an example of `MakeSet`. Then, we motivate the combination of applicative functors and module generation.

To manipulate and generate code of modules, $\lambda^{<M_A>}$ newly 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 written in $\lambda^{<M_A>}$ (Figure 4.1). Following Watanabe et al.'s language, our $\lambda^{<M_A>}$ provides a constructor \$ to extract code of a component from code of a module and a constructor **Runmod** to *run* code of a module. On the other hand, CSP for types (%) is implicitly performed. This choice is based on the design that our language does not treat type generation. The key point in $\lambda^{<M_A>}$ is to distinguish code of modules from code of core expressions in order

```
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 mcod)
```

Figure 4.1: MakeSet Functor in $\lambda^{<M_A>}$

```
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 4.2: MakeSet Functor Translated from $\lambda^{<M_A>}$ to MetaOCaml

to avoid expressions that cannot be translated as described in Section 3.2.2. To do so, we introduce a type **mcod**, brackets $\langle\!\langle\rangle\!\rangle$, and an escape $\approx$, 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 4.1, `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 4.2 shows a program translated from Figure 4.1, where the definitions of `SET'` and `EQ'` are omitted because they are the same as in Figure 3.2. Through the translation, constructors for module generation are eliminated and genlet is inserted into the body of the component `member`.

Next, we explain the usefulness of module generation with applicative functors. We borrow an example from Leroy's paper [24]. The example is a functor `MakeDict` implementing dictionaries without code generation:

```
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)
```

where a type of keys of a dictionary is parameterized. Then consider extending this functor with the operation `domain` that returns a 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 abstraction overhead of functor applications, we can rewrite the above program in the language $\lambda^{<M_A>}$.

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

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 =
        (* return a set of elements of the queue *)
      ...
    end: PRIOQUEUE) ⟫
```

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

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

`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 a 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 is no above problem. Therefore, we argue that applicative functors are useful for module generation.

Besides the above merit, since applicative functors and second-class modules are common in OCaml programming, existing OCaml programs can be staged in $\lambda^{<M_A>}$ with little cost.

However, module generation in $\lambda^{<M_A>}$ has some demerits compared with $\lambda^{<M_G>}$ introduced in the next section. Because dependencies between second-class modules are statically solved, we cannot choose at runtime which modules to be specialized. Hence, there is trade-off between the two languages.

4.3 $\lambda^{<M_G>}$: Extension to Generative Functors and First-Class Modules

We also propose a language $\lambda^{<M_G>}$ for generating and manipulating code of first-class modules in generative semantics, which is a refined version of Watanabe et al's $\lambda^{<M>}$. The most useful aspect of $\lambda^{<M_G>}$ is that a program can choose code of modules. Figure 4.3 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 now, we may be able to generate the main application inlined for all possible combinations at compile time, such as `consoleLogApp` and `fileLogApp`.

However, in generally, applications have many runtime parameters such as command-line arguments. Due to combinatorial explosions, it is difficult to generate them at compile time for all possible combinations. Hence, generating code of modules at runtime is useful for specializing applications with many parameters.

Another aspect is that abstraction overhead can be eliminated from programs suitable for generative functors such as the example of `SymbolTable` described in Section 2.1.4. Functors represented as ordinary functions return modules with fresh abstract types.

$\lambda^{<M_G>}$ 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 $\langle\!\langle\rangle\!\rangle$ 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, at least in the Watanabe et al.'s translation (and ours), code of ordinary expressions and code of modules have different semantics, so they need to be distinguished.

Figure 4.4 shows `MakeSet` program written in $\lambda^{<M_G>}$. This program is the same as the program in Figure 3.1 except for the code type and brackets for modules. Our translation rewrites this program to the program shown in Figure 4.5. The translated program is the same as the one in Figure 3.2 except for genlet.

```
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) "Starting 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 4.3: Choosing Logger Module 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 4.4: MakeSet Functor in $\lambda^{<M_G>}$

```
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 = Runcode.run S.member
  end: EQ))
```

Figure 4.5: MakeSet Functor Translated from $\lambda^{<M_G>}$ to MetaOCaml

Chapter 5

Proposed Langugae: $\lambda^{<M_A>}$

Our first language $\lambda^{<M_A>}$ is a two-stage programming language for generating and manipulating code of second-class modules with applicative functors, which is an extension of core MetaOCaml. It includes simply typed lambda calculus with let expressions, second-class modules, and multi-stage constructors for code generation. The design of the language is based on Leroy's applicative-functor calculus [24], the classic type system $\lambda\circ$ [25], and Watanabe et al.'s calculus [16]. We confine ourselves to a minimal language to express our results. The language $\lambda^{<M_A>}$ is implementad by a translation to MetaOCaml.

In this chapter, we first define the syntax and type system of $\lambda^{<M_A>}$. Then, we define the translation from $\lambda^{<M_A>}$ to MetaOCaml and prove type preservation of the translation.

5.1 Syntax

Figure 5.1 defines the syntax for terms. 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 value, type, and module, respectively), and x_i, t_i, and X_i are identifiers (for value, type, and module, respectively). All identifiers (e.g. x_i) have a name part (x) and a stamp part (i). The stamp exists to distinguish identifiers with the same name, which can be replaced by α-conversion. In contrast, α-conversion does not change the name because components of modules are referenced by names from outside the modules, as in $X_i.x$. Duplicate component names are prohibited by typing rules (see the next section). In this language, base types and primitives are unspecified, but it is easy to introduce them. Complete programs P are sequence of structure components. For simplicity, we sometimes omit **prog** and **end** in program examples.

The syntax for terms is mostly standard except the following. We introduce brackets $\langle\!\langle\rangle\!\rangle$, an escape $\approx$, and **Runmod**, for module expressions. Because the module expressions are *second-class* and exist on a different layer than the core expressions, the multi-stage constructors should be distinguished. In contrast, $<>$, $\sim$, and **run** are the standard multi-stage constructors for the core expressions. Following Watanabe et al.'s calculus, we introduce the \$ constructor to extract a component contained in code of a module as code. For example,

$$
\begin{array}{rl}
\text{Module expressions :} & m ::= X_i \mid p.X \mid \textbf{struct } s \textbf{ end} \mid (m : M) \mid m\ (p) \\
& \quad \mid \textbf{functor } (X_i : M) \rightarrow m \\
& \quad \mid \langle\!\langle m \rangle\!\rangle \mid \approx m \ \mid \textbf{Runmod } (m : M) \\
& \quad \mid \$p.X \\
\text{Structures :} & s ::= \epsilon \mid c\ s \\
\text{Structure components :} & c ::= \textbf{let } x_i : \tau \ = \ e \mid \textbf{type } t_i \ = \ \tau \mid \textbf{module } X_i = m \\
\text{Access paths :} & p ::= X_i \mid p.X \mid p_1(p_2) \mid \$p.X \\
\text{Core expressions :} & e ::= x_i \mid p.x \\
& \quad \mid \textbf{fun } x_i \ \rightarrow \ e \mid e\ e \mid \textbf{let } x_i \ = \ e \textbf{ in } e \\
& \quad \mid < e > \mid \sim e \mid \textbf{run } e \\
& \quad \mid \$p.x \\
\text{Program :} & P ::= \textbf{prog } s \textbf{ end}
\end{array}
$$

Figure 5.1: Syntax for terms

$$
\begin{array}{rl}
\text{Module types :} & M ::= \textbf{sig } S \textbf{ end} \mid \textbf{functor } (X_i : M_1) \rightarrow M_2 \\
& \quad \mid M \textbf{ mcod} \\
\text{Signatures :} & S ::= \epsilon \mid C\ S \\
\text{Signature components :} & C ::= \textbf{val } x_i : \tau \mid \textbf{type } t_i \mid \textbf{type } t_i \ = \ \tau \mid \textbf{module } X_i : M \\
\text{Core types :} & \tau ::= t_i \mid p.t \\
& \quad \mid \tau \ \rightarrow \ \tau \\
& \quad \mid \tau \textbf{ code} \\
& \quad \mid \$p.t
\end{array}
$$

Figure 5.2: Syntax for types

$$
\begin{array}{l}
E ::= \epsilon \mid C\ ^{l}, E \\
\Delta ::= \epsilon \mid C\ ^{l}, \Delta
\end{array}
$$

Figure 5.3: Typing environments

$\$p.x$ 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 $\$(p).x$. To simplify the syntax, a functor definition and a restriction by a signature, are defined by **functor** $(X_i : M) \to m$ and $(m : M)$, respectively. They are unfamiliar syntax for OCaml users, but OCaml (and MetaOCaml) supports them. The key to applicative functors is that the access paths include a path application $p_1(p_2)$, which 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. For more details, see Section 5.2.5.

Figure 5.2 defines the syntax for types. We use metavariables M for module types, S for a sequence of signature components, C for signature components, and τ 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_i) and a manifest type component (**type** $t_i \ = \ \tau$). The $\$p.t$ refers to the type component t within code of a module specified with the path p. Our language does not provide an explicit syntax for CSP (Cross-Stage Persistence) for types in contrast to Watanabe et al's language. This choice follows our principle of not generating types.

5.2 Type System

5.2.1 Typing Environments

Figure 5.3 shows typing environments of the language $\lambda^{<M_A>}$. The typing environments E and Δ are both a sequence of signature components that are annotated with a level (stage) l. Since $\lambda^{<M_A>}$ is a two-stage language, l is either 0 or 1. Δ is a subset of E, which contains bindings for level-1 value components and level-1 module components. Δ is used in translation, not for typing in the language.

5.2.2 Typing Judgements

Typing judgments are also annotated by the level l. For example, $E; \Delta \vdash^l$ means a typing judgment at the level l under the environment E and Δ. At the level l, only the elements annotated with l in the environment E may be dereferenced.

5.2.3 Well-Typedness

We define that a program P is well-typed if $\vdash \ P$ **wt** is derivable. WT-Prog is a rule for well-typedness of programs. The stage level starts from 0. Both typing environments are empty.

$\boxed{\vdash P \ \textbf{wt}}$

$$\frac{\phi; \phi \vdash^0 \ \textbf{struct}\ s\ \textbf{end}\ :\ \textbf{sig}\ S\ \textbf{end}}{\vdash\ \textbf{prog}\ s\ \textbf{end wt}}\ (\text{WT-Prog})$$

5.2.4 Well-Formedness

Our language $\lambda^{<M_A>}$ requires that all component names within each module are mutually distinct. We define types that satisfy such property as *well-formed* (**wf**). For instance, a signature **sig type** t **type** t **end** contains components with the same name, so it is not well-formed. In this section, we define the well-formedness rules.

$E; \Delta \vdash^l M$ **wf** means that the module type M is well-formed at level l under the environments E and Δ. WF-Sig is a rule for signatures, which defines a signature as well-formed if all signature components are well-formed. WF-Functor and WF-ModCod are rules for functors and code of modules, respectively.

$\boxed{E; \Delta \vdash^l M \textbf{ wf}}$

$$\frac{E; \Delta \vdash^l S \textbf{ wf}}{E; \Delta \vdash^l \textbf{sig } S \textbf{ end wf}} \text{ (WF-Sig)}$$

$$\frac{E; \Delta \vdash^0 M_1 \textbf{ wf} \qquad E,\ (\textbf{module } X_i : M_1)^0; \Delta \vdash^0 M_2 \textbf{ wf}}{E; \Delta \vdash^0 \textbf{functor } (X_i : M_1) \to M_2 \textbf{ wf}} \text{ (WF-Functor)}$$

$$\frac{E; \Delta \vdash^0 M \textbf{ wf}}{E; \Delta \vdash^0 M \textbf{ mcod wf}} \text{ (WF-ModCod)}$$

Rules for signature components are shown below. WF-SigComponents requires all signature components are well-formed. WF-Val is a rule for value components, WF-TypeAbs is for abstract type components, WF-Type is for manifest type components, and WF-Mod is for module components. We write $\textbf{Dom}(E)$ for the set of identifiers bound in the environment E. The conditions in the form of $x_i^l \notin \textbf{Dom}(E)$ are used to guarantee that each component name is distinct. We may omit rules with the abstract type component when we can infer from rules with the manifest type components.

$\boxed{E; \Delta \vdash^l S \textbf{ wf}}$

$$\frac{}{E; \Delta \vdash^l \epsilon \textbf{ wf}} \text{ (WF-Empty)}$$

$$\frac{E; \Delta \vdash^l C \textbf{ wf} \qquad E,\ (C)^l; \Delta \vdash^l S \textbf{ wf}}{E; \Delta \vdash^l C\ S \textbf{ wf}} \text{ (WF-SigComponents)}$$

$\boxed{E; \Delta \vdash^l C \textbf{ wf}}$

$$\frac{E; \Delta \vdash^l \tau \textbf{ wf} \qquad {x_i}^l \notin \textbf{Dom}(E)}{E; \Delta \vdash^l \textbf{val } x_i : \tau \textbf{ wf}} \text{ (WF-Val)}$$

$$\frac{{t_i}^l \notin \textbf{Dom}(E)}{E; \Delta \vdash^l \textbf{type } t_i \textbf{ wf}} \text{ (WF-TypeAbs)}$$

$$\frac{{t_i}^l \notin \mathbf{Dom}(E)}{E; \Delta \vdash^l \ \mathbf{type}\ t_i \ = \ \tau\ \mathbf{wf}}\ (\text{WF-Type})$$

$$\frac{E; \Delta \vdash^l \ M\ \mathbf{wf} \qquad {X_i}^l \notin \mathbf{Dom}(E)}{E; \Delta \vdash^l \ \mathbf{module}\ X_i : M\ \mathbf{wf}}\ (\text{WF-Mod})$$

Rules for the core type τ are shown below. T-Var for variables and T-Arr for arrow types are straightforward. T-Dot is a rule for dereferencing a component in a module, while T-DotCode is for a component in code of a module. T-VarAbs, T-DotAbs, and T-DotCodeAbs are rules for abstract type components. A rule for code types is T-Code. Since $\lambda^{<M_A>}$ is a two-stage multistage language, code types can only appear at level 0. CSPing for types is performed implicitly using T-Csp.

$\boxed{E; \Delta \vdash^l \ \tau\ \mathbf{wf}}$

$$\frac{(\mathbf{type}\ t_i \ = \ \tau)^l \in E}{E; \Delta \vdash^l \ t_i\ \mathbf{wf}}\ (\text{T-Var})$$

$$\frac{(\mathbf{type}\ t_i)^l \in E}{E; \Delta \vdash^l \ t_i\ \mathbf{wf}}\ (\text{T-VarAbs})$$

$$\frac{E; \Delta \vdash^l \ p \ : \ \mathbf{sig}\ S_1\ (\mathbf{type}\ t_i \ = \ \tau)\ S_2\ \mathbf{end}}{E; \Delta \vdash^l \ p.t\ \mathbf{wf}}\ (\text{T-Dot})$$

$$\frac{E; \Delta \vdash^l \ p \ : \ \mathbf{sig}\ S_1\ (\mathbf{type}\ t_i)\ S_2\ \mathbf{end}}{E; \Delta \vdash^l \ p.t\ \mathbf{wf}}\ (\text{T-DotAbs})$$

$$\frac{E; \Delta \vdash^0 \ p \ : \ (\mathbf{sig}\ S_1\ (\mathbf{type}\ t_i \ = \ \tau)\ S_2\ \mathbf{end})\ \mathbf{mcod}}{E; \Delta \vdash^0 \ \$p.t\ \mathbf{wf}}\ (\text{T-DotCode})$$

$$\frac{E; \Delta \vdash^0 \ p \ : \ (\mathbf{sig}\ S_1\ (\mathbf{type}\ t_i)\ S_2\ \mathbf{end})\ \mathbf{mcod}}{E; \Delta \vdash^0 \ \$p.t\ \mathbf{wf}}\ (\text{T-DotCodeAbs})$$

$$\frac{E; \Delta \vdash^l \ \tau_1\ \mathbf{wf} \qquad E; \Delta \vdash^l \ \tau_2\ \mathbf{wf}}{E; \Delta \vdash^l \ \tau_1 \ \rightarrow \ \tau_2\ \mathbf{wf}}\ (\text{T-Arr})$$

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

$$\frac{E; \Delta \vdash^0 \ \tau\ \mathbf{wf}}{E; \Delta \vdash^1 \ \tau\ \mathbf{wf}}\ (\text{T-Csp})$$

5.2.5 Typing Rules

This section defines typing rules of $\lambda^{<M_A>}$. First, we describe the typing rules for structure components. As with the well-formedness rules, these rules require component names to be unique at each level. The environment Δ is updated in S-Let1 and S-Mod1. Because our translation moves brackets from outside of a module to inside components, level-1 value components and level-1 module components are exposed at level 0 after the translation. Hence, we accumulate those components to Δ and use it in translation time.

$\boxed{E; \Delta \vdash^l \ s \ : \ S}$

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

$$\frac{E; \Delta \vdash^0 \ e \ : \ \tau \qquad {x_i}^0 \notin \mathbf{Dom}(E) \qquad E, \ (\mathbf{val} \ x_i : \tau)^0; \Delta \vdash^0 \ s \ : \ S}{E; \Delta \vdash^0 \ (\mathbf{let} \ x_i : \tau \ = \ e) \ s \ : \ (\mathbf{val} \ x_i : \tau) \ S} \text{ (S-Let0)}$$

$$\frac{E; \Delta \vdash^1 \ e \ : \ \tau \qquad {x_i}^1 \notin \mathbf{Dom}(E) \qquad E, (\mathbf{val} \ x_i : \tau)^1; \Delta, (\mathbf{val} \ x_i : \tau)^1 \vdash^1 \ s \ : \ S}{E; \Delta \vdash^1 \ (\mathbf{let} \ x_i : \tau \ = \ e) \ s \ : \ (\mathbf{val} \ x_i : \tau) \ S} \text{ (S-Let1)}$$

$$\frac{{t_i}^l \notin \mathbf{Dom}(E) \qquad E, \ (\mathbf{type} \ t_i \ = \ \tau)^l; \Delta \vdash^l \ s \ : \ S}{E; \Delta \vdash^l \ (\mathbf{type} \ t_i \ = \ \tau) \ s \ : \ (\mathbf{type} \ t_i \ = \ \tau) \ S} \text{ (S-Type)}$$

$$\frac{\begin{array}{c} E; \Delta \vdash^0 \ m \ : \ M \qquad E; \Delta \vdash^0 \ M \ \mathbf{wf} \qquad {X_i}^0 \notin \mathbf{Dom}(E) \\ E, \ (\mathbf{module} \ X_i : M)^0; \Delta \vdash^0 \ s \ : \ S \end{array}}{E; \Delta \vdash^0 \ (\mathbf{module} \ X_i = m) \ s \ : \ (\mathbf{module} \ X_i : M) \ S} \text{ (S-Mod0)}$$

$$\frac{\begin{array}{c} E; \Delta \vdash^1 \ m \ : \ M \qquad E; \Delta \vdash^1 \ M \ \mathbf{wf} \qquad {X_i}^1 \notin \mathbf{Dom}(E) \\ E, (\mathbf{module} \ X_i : M)^1; \Delta, (\mathbf{module} \ X_i : M)^1 \vdash^1 \ s \ : \ S \end{array}}{E; \Delta \vdash^1 \ (\mathbf{module} \ X_i = m) \ s \ : \ (\mathbf{module} \ X_i : M) \ S} \text{ (S-Mod1)}$$

Second, we describe the typing rules for module expressions. M-Var is a rule for module variables. M-Dot is a rule for module components in a module, and M-DotCode is for module components in code of a module. In these two rules, the substitution is performed in order to avoid free occurrences of type variables. We call it substitution for type variables, which is defined in Section 5.2.6. We write $\mathbf{Dom}(S)$ for the set of identifiers defined in the signature S. M-Str is for structure expressions and M-Sig is for a restriction by a signature. To avoid code of functors, M-Functor for functor definitions and M-App for functor applications are defined at level 0 only. M-App requires a path in the functor argument. In the semantics of applicative, the result type of a functor application may include path applications such as `sig type t = F(X).t end`, but the syntax does not allow the `X` to be a module expression. Subtyping for modules is defined by M-Subtyping, and its rules are defined later. We introduce multi-stage constructors for modules M-Cod, M-Esc, and M-Runmod. M-Cod is a rule for code of modules, for a module expression m of type M, $\langle\!\langle m \rangle\!\rangle$ has a type of M **mcod**. An escape for code of a module is typed by M-Esc, and **Runmod** is typed by M-Runmod. M-Strengthening is an important rule for module type equivalences, which enriches the module type `M` with the path `p`.

$\boxed{E; \Delta \vdash^l \ m \ : \ M}$

$$\frac{(\textbf{module}\ X_i : M)^l \in E}{E; \Delta \vdash^l \ X_i \ : \ M} \ \text{(M-Var)}$$

$$\frac{E; \Delta \vdash^l \ p \ : \ \textbf{sig}\ S_1\ (\textbf{module}\ X_i : M)\ S_2\ \textbf{end}}{E; \Delta \vdash^l \ p.X \ : \ M[z_j \leftarrow p.z \mid z_j \in \textbf{Dom}(S_1)]} \ \text{(M-Dot)}$$

$$\frac{E; \Delta \vdash^0 \ p \ : \ (\textbf{sig}\ S_1\ (\textbf{module}\ X_i : M)\ S_2\ \textbf{end})\ \textbf{mcod}}{E; \Delta \vdash^0 \ \$p.X \ : \ M[z_j \leftarrow \$p.z \mid z_j \in \textbf{Dom}(S_1)]\ \textbf{mcod}} \ \text{(M-DotCode)}$$

$$\frac{E; \Delta \vdash^l \ s \ : \ S}{E; \Delta \vdash^l \ \textbf{struct}\ s\ \textbf{end} \ : \ \textbf{sig}\ S\ \textbf{end}} \ \text{(M-Str)}$$

$$\frac{E; \Delta \vdash^l \ M\ \textbf{wf} \qquad E; \Delta \vdash^l \ m \ : \ M}{E; \Delta \vdash^l \ (m : M) \ : \ M} \ \text{(M-Sig)}$$

$$\frac{{X_i}^0 \notin \textbf{Dom}(E) \qquad E,\ (\textbf{module}\ X_i : M_2)^0; \Delta \vdash^0 \ m \ : \ M_1}{E; \Delta \vdash^0 \ \textbf{functor}\ (X_i : M_2) \to m \ : \ \textbf{functor}\ (X_i : M_2) \to M_1} \ \text{(M-Functor)}$$

$$\frac{E; \Delta \vdash^0 \ m \ : \ \textbf{functor}\ (X_i : M_2) \to M_1 \qquad E; \Delta \vdash^0 \ p \ : \ M_2}{E; \Delta \vdash^0 \ m\ (p) \ : \ M_1[X_i \leftarrow p]} \ \text{(M-App)}$$

$$\frac{E; \Delta \vdash^l \ m \ : \ M_1 \qquad E; \Delta \vdash^l \ M_1 \ <: \ M_2}{E; \Delta \vdash^l \ m \ : \ M_2} \ \text{(M-Subtyping)}$$

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

$$\frac{E; \Delta \vdash^0 \ m \ : \ M\ \textbf{mcod}}{E; \Delta \vdash^1 \ \approx m \ : \ M} \ \text{(M-Esc)}$$

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

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

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 [24], 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**. Its precise definition is given below.

$$
\begin{aligned}
(\textbf{sig}\ S\ \textbf{end})/p^l &= \textbf{sig}\ S/p^l\ \textbf{end} \\
(\textbf{functor}\ (X_i : M_1) \to M_2)/p^0 &= \textbf{functor}\ (X_i : M_1) \to M_2/(p(X_i))^0 \\
(M\ \textbf{mcod})/p^0 &= M/p^1\ \textbf{mcod} \\
\epsilon/p^l &= \epsilon \\
((\textbf{val}\ x_i : \tau)\ S)/p^l &= (\textbf{val}\ x_i : \tau)\ S/p^l \\
((\textbf{type}\ t_i)\ S)/p^0 &= (\textbf{type}\ t_i\ =\ p.t)\ S/p^0 \\
((\textbf{type}\ t_i)\ S)/p^1 &= (\textbf{type}\ t_i\ =\ \$p.t)\ S/p^1 \\
((\textbf{type}\ t_i\ =\ \tau)\ S)/p^l &= (\textbf{type}\ t_i\ =\ \tau)\ S/p^l \\
((\textbf{module}\ X_i : M)\ S)/p^0 &= (\textbf{module}\ X_i : M/(p.X)^0)\ S/p^0 \\
((\textbf{module}\ X_i : M)\ S)/p^1 &= (\textbf{module}\ X_i : M/(\$p.X)^1)\ S/p^1
\end{aligned}
$$

Then, we describe the typing rules for core expressions. E-VAR is a rule for value variables. E-DOT and E-DOTCODE are rules for dereferencing value components within a module and code of a module, respectively. The substitution for type variables in these two rules is defined in Section 5.2.6. We briefly explain why the substitution is necessary. Considering a signature `sig type t val x: t -> t end` is bound to a path `p`, an expression `p.x` should have the type `p.t -> p.t` because the `t` is free outside the module. In E-DOTCODE, if `p` has a module type `sig type t val x: t -> t end mcod`, then `$p.x` has `($p.t -> $p.t) code`. E-FUN, E-APP, and E-LET rules are the same as the usual rules for function abstractions, function applications, and let expressions, respectively, except for the annotation of the level. E-CODE, E-ESC, and E-RUN are rules for multi-stage constructors for core expressions. E-SUBSUMPTION is a subsumption rule for core expressions.

$\boxed{E; \Delta \vdash^l\ e\ :\ \tau}$

$$\frac{(\textbf{val}\ x_i : \tau)^l \in E}{E; \Delta \vdash^l\ x_i\ :\ \tau}\ \text{(E-VAR)}$$

$$\frac{E; \Delta \vdash^l\ p\ :\ \textbf{sig}\ S_1\ (\textbf{val}\ x_i : \tau)\ S_2\ \textbf{end}}{E; \Delta \vdash^l\ p.x\ :\ \tau[z_j \leftarrow p.z \mid z_j \in \textbf{Dom}(S_1)]}\ \text{(E-DOT)}$$

$$\frac{E; \Delta \vdash^0\ p\ :\ (\textbf{sig}\ S_1\ (\textbf{val}\ x_i : \tau)\ S_2\ \textbf{end})\ \textbf{mcod}}{E; \Delta \vdash^0\ \$p.x\ :\ \tau[z_j \leftarrow \$p.z \mid z_j \in \textbf{Dom}(S_1)]\ \textbf{code}}\ \text{(E-DOTCODE)}$$

$$\frac{E; \Delta \vdash^l\ \tau_1\ \textbf{wf} \qquad E,\ (\textbf{val}\ x_i : \tau_1)^l; \Delta \vdash^l\ e\ :\ \tau_2}{E; \Delta \vdash^l\ \textbf{fun}\ x_i\ \to\ e\ :\ \tau_1\ \to\ \tau_2}\ \text{(E-FUN)}$$

$$\frac{E; \Delta \vdash^l\ e_1\ :\ \tau_1\ \to\ \tau_2 \qquad E; \Delta \vdash^l\ e_2\ :\ \tau_1}{E; \Delta \vdash^l\ e_1\ e_2\ :\ \tau_2}\ \text{(E-APP)}$$

$$\frac{E;\Delta \vdash^l \ e_1 \ : \ \tau_1 \qquad E,\ (\mathbf{val}\ x_i : \tau_1)^l; \Delta \vdash^l \ e_2 \ : \ \tau_2}{E;\Delta \vdash^l \ \mathbf{let}\ x_i \ = \ e_1\ \mathbf{in}\ e_2 \ : \ \tau_2}\ \text{(E-Let)}$$

$$\frac{E;\Delta \vdash^1 \ e \ : \ \tau}{E;\Delta \vdash^0 \ < e > \ : \ \tau\ \mathbf{code}}\ \text{(E-Code)}$$

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

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

$$\frac{E;\Delta \vdash^l \ e \ : \ \tau_1 \qquad E;\Delta \vdash^l \ \tau_1 \ \equiv \ \tau_2}{E;\Delta \vdash^l \ e \ : \ \tau_2}\ \text{(E-Subsumption)}$$

The subtyping rules for modules are the same as Leroy's calculus, except for the following. First, our typing judgment is annotated with a level. Second, subtyping for functors is defined only at level 0. Finally, the subtyping rule Sub-Mcod is added for code of modules. The definitions are shown below.

$\boxed{E;\Delta \vdash^l \ M_1 \ <: \ M_2}$

$$\frac{\sigma : \{1...m\} \to \{1...n\} \qquad E;\ C_1^l \cdots C_n^l \vdash^l \ C_{\sigma(i)} \ <: \ C_i' \ \text{ for } i = 1 \cdots m}{E;\Delta \vdash^l \ \mathbf{sig}\ C_1 \cdots C_n\ \mathbf{end} \ <: \ \mathbf{sig}\ C_1' \cdots C_m'\ \mathbf{end}}\ \text{(Sub-Sig)}$$

$$\frac{E;\Delta \vdash^0 \ M_2 \ <: \ M_1 \qquad E,\ (\mathbf{module}\ Y_j : M_2)^0; \Delta \vdash^0 \ M_1'[X_i \leftarrow Y_j] \ <: \ M_2'}{E;\Delta \vdash^0 \ \mathbf{functor}\ (X_i : M_1) \to M_1' \ <: \ \mathbf{functor}\ (Y_j : M_2) \to M_2'}\ \text{(Sub-Functor)}$$

$$\frac{E;\Delta \vdash^1 \ M_1 \ <: \ M_2}{E;\Delta \vdash^0 \ M_1\ \mathbf{mcod} \ <: \ M_2\ \mathbf{mcod}}\ \text{(Sub-Mcod)}$$

$$\frac{E;\Delta \vdash^l \ \tau_1 \ \equiv \ \tau_2}{E;\Delta \vdash^l \ \mathbf{val}\ x_i : \tau_1 \ <: \ \mathbf{val}\ x_i : \tau_2}\ \text{(Sub-Val)}$$

$$\frac{E;\Delta \vdash^l \ M_1 \ <: \ M_2}{E;\Delta \vdash^l \ \mathbf{module}\ X_i : M_1 \ <: \ \mathbf{module}\ X_i : M_2}\ \text{(Sub-Mod)}$$

$$\frac{}{E;\Delta \vdash^l \ \mathbf{type}\ t_i \ <: \ \mathbf{type}\ t_i}\ \text{(Sub-AbstractAbstract)}$$

$$\frac{}{E;\Delta \vdash^l \ \mathbf{type}\ t_i \ = \ \tau \ <: \ \mathbf{type}\ t_i}\ \text{(Sub-ManifestAbstract)}$$

$$\frac{E;\Delta \vdash^l \ \tau_1 \ \equiv \ \tau_2}{E;\Delta \vdash^l \ \mathbf{type}\ t_i \ = \ \tau_1 \ <: \ \mathbf{type}\ t_i \ = \ \tau_2}\ \text{(Sub-ManifestManifest)}$$

$$\frac{E; \Delta \vdash^l \ t_i \ \equiv \ \tau}{E; \Delta \vdash^l \ \textbf{type} \ t_i \ <: \ \textbf{type} \ t_i \ = \ \tau} \ (\textsc{Sub-AbstractManifest})$$

$E; \Delta \vdash^l \ \tau_1 \ \equiv \ \tau_2$ means that the type τ_1 and the type τ_2 are equivalent at the level l. Type equivalence for ordinary type variables is defined by EQ-Type. On the other hand, type equivalence for type components is defined by EQ-Dot and EQ-DotCode. The substitution for type variables in EQ-Dot and EQ-DotCode is defined in Section 5.2.6. Usual rules for congruence, reflexivity, symmetry, and transitivity are omitted.

$\boxed{E; \Delta \vdash^l \ \tau_1 \ \equiv \ \tau_2}$

$$\frac{}{E_1, \ \textbf{type} \ t_i \ = \ \tau, \ E_2; \Delta \vdash^l \ t_i \ \equiv \ \tau} \ (\textsc{Eq-Type})$$

$$\frac{E; \Delta \vdash^l \ p \ : \ \textbf{sig} \ S_1 \ (\textbf{type} \ t_i \ = \ \tau) \ S_2 \ \textbf{end}}{E; \Delta \vdash^l \ p.t \ \equiv \ \tau[z_j \leftarrow p.z \mid z_j \in \textbf{Dom}(S_1)]} \ (\textsc{Eq-Dot})$$

$$\frac{E; \Delta \vdash^0 \ p \ : \ (\textbf{sig} \ S_1 \ (\textbf{type} \ t_i \ = \ \tau) \ S_2 \ \textbf{end}) \ \textbf{mcod}}{E; \Delta \vdash^0 \ \$p.t \ \equiv \ \tau[z_j \leftarrow \$p.z \mid z_j \in \textbf{Dom}(S_1)] \ \textbf{code}} \ (\textsc{Eq-DotCode})$$

5.2.6 Substitution

This section defines the substitution for type variables in E-Dot, E-DotCode, M-Dot, M-DotCode, EQ-Dot and EQ-DotCode. We first define $\mathit{subst}^{\bullet}(\tau, \ p, \ S)$ for $\tau\,[z_j \leftarrow p.z \mid z_j \in \textbf{Dom}(S)]$. For example, assuming that a signature S includes a type component t_i, the result of $\mathit{subst}^{\bullet}(t_i \ \rightarrow \ t_i, \ p, \ S)$ is $p.t \ \rightarrow \ p.t$. In the same way, we define $\mathit{subst}^{\$}(\tau, \ p, \ S)$ for $\tau\,[z_j \leftarrow \$p.z \mid z_j \in \textbf{Dom}(S)]$. The difference from the first definition is that components defined in S are accessed with a dollar instead of a dot. In other words, $\mathit{subst}^{\bullet}(\tau, \ p, \ S)$ is used when p refers a module, while $\mathit{subst}^{\$}(\tau, \ p, \ S)$ is used when p refers code of a module. The definitions are shown below, where α is either $\bullet$ or \$.

$\boxed{\mathit{subst}^{\alpha}(\tau, \ p, \ S)}$

$$\begin{aligned}
\mathit{subst}^{\bullet}(t_i, \ p, \ S) \ &= \ \begin{cases} p.t & (t_i \in \mathrm{Dom}(S)) \\ t_i & (\mathit{otherwise}) \end{cases} \\
\mathit{subst}^{\$}(t_i, \ p, \ S) \ &= \ \begin{cases} \$p.t & (t_i \in \mathrm{Dom}(S)) \\ t_i & (\mathit{otherwise}) \end{cases} \\
\mathit{subst}^{\bullet}(p'.t, \ p, \ S) \ &= \ \begin{cases} p.p'.t & (\mathrm{head}(p') \in \mathrm{Dom}(S)) \\ p'.t & (\mathit{otherwise}) \end{cases} \\
\mathit{subst}^{\bullet}(\$p'.t, \ p, \ S) \ &= \ \begin{cases} \$(p.p').t & (\mathrm{head}(p') \in \mathrm{Dom}(S)) \\ \$p'.t & (\mathit{otherwise}) \end{cases} \\
\mathit{subst}^{\alpha}(\tau_1 \ \rightarrow \ \tau_2, \ p, \ S) \ &= \ \mathit{subst}^{\alpha}(\tau_1, \ p, \ S) \ \rightarrow \ \mathit{subst}^{\alpha}(\tau_2, \ p, \ S) \\
\mathit{subst}^{\alpha}(\tau \ \textbf{code}, \ p, \ S) \ &= \ \mathit{subst}^{\alpha}(\tau, \ p, \ S) \ \textbf{code}
\end{aligned}$$

We use the auxiliary function head(p), which means the head of the path p. The definition is as follows:

$$\begin{aligned}\mathrm{head}(X_i) &= X_i \\ \mathrm{head}(p.X) &= \mathrm{head}(p) \\ \mathrm{head}(\$p.X) &= \mathrm{head}(p)\end{aligned}$$

We give an additional explanation for $subst^{\bullet}\,(\$p'.t,\ p,\ S)$. This substitution is used when a module `A` has the following signature:

```
module A: sig
  module B: sig
    type t
  end mcod
  val x: $B.t
end
```

If we have an expression `A.x` outside the module `A`, its type is derived as follows:

$$\frac{E;\Delta \vdash^{l}\ A\ :\ \mathbf{sig}\ S_1\ (\mathbf{val}\ x : \$B.t)\ S_2\ \mathbf{end}}{E;\Delta \vdash^{l}\ A.x\ :\ (\$B.t)[z_j \leftarrow A.z \mid z_j \in \mathbf{Dom}(S_1)]}\ (\text{E-Dot})$$

where $(\$B.t)[z_j \leftarrow A.z \mid z_j \in \mathbf{Dom}(S_1)]$ is represented as $subst^{\bullet}\,(\$B.t,\ A,\ S_1)$. Note that $\mathbf{Dom}(S)$ for a signature S does not only include the names of type components, but also the names of module components. Namely, $\mathbf{Dom}(S_1)$ includes B. Therefore, the expression `A.x` has type `$(A.B).t` rather than `$B.t`,

Then, we define $subst^{\bullet}\,(M,\ p,\ S)$ for $M\,[z_j \leftarrow p.z \mid z_j \in \mathbf{Dom}(S)]$, and $subst^{\$}\,(M,\ p,\ S)$ for $M\,[z_j \leftarrow \$p.z \mid z_j \in \mathbf{Dom}(S)]$. These are used in M-Dot and M-DotCode. To propagate the substitution for type variables to components, we also define $subst^{\alpha}\,(S',\ p,\ S)$ and $subst^{\alpha}\,(C,\ p,\ S)$. $subst^{\$}\,(\mathbf{module}\ X_i : M,\ p,\ S)$ is used for modules within code of a module.

$\boxed{subst^{\alpha}\,(M,\ p,\ S)}$

$$\begin{aligned}subst^{\alpha}\,(\mathbf{sig}\ S'\ \mathbf{end},\ p,\ S) &= \mathbf{sig}\ subst^{\alpha}\,(S',\ p,\ S)\ \mathbf{end} \\ subst^{\alpha}\,(\mathbf{functor}\ (X_i : M_1) \to M_2,\ p,\ S) &= \mathbf{functor}\ (X_i : subst^{\alpha}\,(M_1,\ p,\ S)) \to subst^{\alpha}\,(M_2,\ p,\ S) \\ subst^{\alpha}\,(M\ \mathbf{mcod},\ p,\ S) &= subst^{\alpha}\,(M,\ p,\ S)\ \mathbf{mcod}\end{aligned}$$

$\boxed{subst^{\alpha}\,(S',\ p,\ S)}$

$$\begin{aligned}subst^{\alpha}\,(\epsilon,\ p,\ S) &= \epsilon \\ subst^{\alpha}\,(C\ S',\ p,\ S) &= subst^{\alpha}\,(C,\ p,\ S)\ subst^{\alpha}\,(S',\ p,\ S)\end{aligned}$$

$\boxed{subst^{\alpha}\,(C,\ p,\ S)}$

$$\begin{aligned}subst^{\alpha}\,(\mathbf{val}\ x_i : \tau,\ p,\ S) &= \mathbf{val}\ x_i : subst^{\alpha}\,(\tau,\ p,\ S) \\ subst^{\alpha}\,(\mathbf{type}\ t_i,\ p,\ S) &= \mathbf{type}\ t_i \\ subst^{\alpha}\,(\mathbf{type}\ t_i\ =\ \tau,\ p,\ S) &= \mathbf{type}\ t_i\ =\ subst^{\alpha}\,(\tau,\ p,\ S) \\ subst^{\bullet}\,(\mathbf{module}\ X_i : M,\ p,\ S) &= \mathbf{module}\ X_i : subst^{\bullet}\,(M,\ p.X,\ S) \\ subst^{\$}\,(\mathbf{module}\ X_i : M,\ p,\ S) &= \mathbf{module}\ X_i : subst^{\$}\,(M,\ \$p.X,\ S)\end{aligned}$$

5.3 Translation to plain MetaOCaml

We introduce a translation from $\lambda^{<M_A>}$ to plain MetaOCaml. Following Watanabe et al., the translation turns code of a module into a module containing code. The major difference from their translation is the performance improvement of translated code.

We use the notation $[\![\cdot]\!]^l_\Delta$ for the translation, which is parameterized by the level l (for $l = 0, 1$) and the environment Δ which may be referenced in translations for components. For example, $[\![e]\!]^0_\Delta$ is the result of the level-0 translation for the expression e on the Δ, and similarly $[\![e]\!]^1_\Delta$ is the result of the level-1 translation for the expression e on the Δ. $[\![< e >]\!]^0_\Delta$ is translated to $< [\![e]\!]^1_\Delta >$.

We first describe the translation $[\![m]\!]^l_\Delta$ for a module expression m. The key to the rules is to remove the multi-stage constructors for modules. The translation for code of a module is given by Rule 5.7, which eliminates brackets surrounding a module and translates the module at level 1. An escape for code of a module is translated by Rule 5.8. Rule 5.10 is for dereferencing a component from code of a module, which removes \$ simply. The translation for the expression which runs code of a module is complicated (Rule 5.9 and 5.11-5.14), which depend on the signature of the target module since we need to construct a new module. The new module contains a nested module with a fresh name, and components. The expression $S /\!\!/ X_i$, defined below, applies the MetaOCaml's **run**-primitive to each value component in the module X_i of the signature S. The **run**-primitive is propagated to nested modules. In Rule 5.13, the type τ does not appear on the right side, where $X_i.t$ is equal to τ. Other module expressions are kept intact and the environment Δ is not important for now.

$[\![m]\!]^l_\Delta$

$$[\![X_i]\!]^l_\Delta = X_i \tag{5.1}$$

$$[\![p.X]\!]^l_\Delta = [\![p]\!]^l.X \tag{5.2}$$

$$[\![\textbf{struct } s \textbf{ end}]\!]^l_\Delta = \textbf{struct } [\![s]\!]^l_\Delta \textbf{ end} \tag{5.3}$$

$$[\![(m : M)]\!]^l_\Delta = ([\![m]\!]^l_\Delta : [\![M]\!]^l) \tag{5.4}$$

$$[\![m\ (p)]\!]^0_\Delta = [\![m]\!]^0_\Delta\ ([\![p]\!]^0) \tag{5.5}$$

$$[\![\textbf{functor } (X_i : M) \to m]\!]^0_\Delta = \textbf{functor } (X_i : [\![M]\!]^0) \to [\![m]\!]^0_\Delta \tag{5.6}$$

$$[\![\langle\!\langle m \rangle\!\rangle]\!]^0_\Delta = [\![m]\!]^1_\Delta \tag{5.7}$$

$$[\![\approx m]\!]^1_\Delta = [\![m]\!]^0_\Delta \tag{5.8}$$

$$[\![\textbf{Runmod } (m : \textbf{sig } S \textbf{ end mcod})]\!]^0_\Delta = \textbf{struct} \tag{5.9}$$

$$\quad \textbf{module } X_i = [\![m]\!]^0_\Delta$$

$$\quad S /\!\!/ X_i$$

$$\textbf{end}$$

where X_i is a fresh identifier

$$[\![\$p.X]\!]^0_\Delta = [\![p]\!]^0.X \tag{5.10}$$

$S /\!\!/ X_i$

$$\epsilon /\!\!/ X_i = \epsilon \tag{5.11}$$

$$((\textbf{val}\ x_i : \tau)\ S) /\!\!/ X_i = (\textbf{let}\ x_i : \tau\ =\ \textbf{run}\ X_i.x)\ S /\!\!/ X_i \tag{5.12}$$

$$((\textbf{type}\ t_i\ =\ \tau)\ S) /\!\!/ X_i = (\textbf{type}\ t_i\ =\ X_i.t)\ S /\!\!/ X_i \tag{5.13}$$

$$((\textbf{module}\ X_i' : M)\ S) /\!\!/ X_i = (\textbf{module}\ X_i' = [\![\ \textbf{Runmod}\ (X_i.X' : M\ \textbf{mcod})\ ]\!]^0_\epsilon)\ S /\!\!/ X_i \tag{5.14}$$

We show an example of the translation for **Runmod**. Consider the following program written in $\lambda^{<M_A>}$.

```
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 mcod)
```

`A` is code of a module, and `A'` is a module obtained by applying **Runmod** to `A`. The result of the translation is below.

```
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
```

The translation for structure components is defined below. In Rule 5.17, Δ' is the environment updated in the typing rules S-Let1 and S-Mod1. That is, the result of a translation $[\![\ (\textbf{let}\ x_i : \tau\ =\ e)\ s\ ]\!]^1_\Delta$ is $[\![\ \textbf{let}\ x_i : \tau\ =\ e\ ]\!]^1_\Delta\ [\![\ s\ ]\!]^1_{\Delta,\ (\textbf{val}\ x_i:\tau)^1}$, and the result of a translation $[\![\ (\textbf{module}\ X_i = m)\ s\ ]\!]^1_\Delta$ is $[\![\ \textbf{module}\ X_i = m\ ]\!]^1_\Delta\ [\![\ s\ ]\!]^1_{\Delta,\ (\textbf{module}\ X_i:M)^1}$.

$\boxed{[\![\ s\ ]\!]^l_\Delta}$

$$[\![\ \epsilon\ ]\!]^l_\Delta = \epsilon \tag{5.15}$$
$$[\![\ c\ s\ ]\!]^0_\Delta = [\![\ c\ ]\!]^0_\Delta\ [\![\ s\ ]\!]^0_\Delta \tag{5.16}$$
$$[\![\ c\ s\ ]\!]^1_\Delta = [\![\ c\ ]\!]^1_\Delta\ [\![\ s\ ]\!]^1_\Delta, \tag{5.17}$$

Structure components are translated by Rule 5.18-5.21. As we described in Chapter 4, we use the genlet primitive to avoid code duplication. By Rule 5.19, a level-1 value component (within code of a module) is translated into code of the value and genlet is inserted in front of it. Others are kept intact.

$\boxed{[\![\ c\ ]\!]^l_\Delta}$

$$[\![\ \textbf{let}\ x_i : \tau\ =\ e\ ]\!]^0_\Delta = \textbf{let}\ x_i : [\![\ \tau\ ]\!]^0\ =\ [\![\ e\ ]\!]^0_\Delta \tag{5.18}$$
$$[\![\ \textbf{let}\ x_i : \tau\ =\ e\ ]\!]^1_\Delta = \textbf{let}\ x_i : [\![\ \tau\ ]\!]^1\ \textbf{code}\ =\ \textbf{genlet}\ <[\![\ e\ ]\!]^1_\Delta> \tag{5.19}$$
$$[\![\ \textbf{type}\ t_i\ =\ \tau\ ]\!]^l_\Delta = \textbf{type}\ t_i\ =\ [\![\ \tau\ ]\!]^l \tag{5.20}$$
$$[\![\ \textbf{module}\ X_i = m\ ]\!]^l_\Delta = \textbf{module}\ X_i = [\![\ m\ ]\!]^l_\Delta \tag{5.21}$$

Core expressions are translated by Rule 5.22-5.32. Because variables (component names) in Δ have level 1 before the translation, and they are bound to level-0 expressions, we need to splice them. Hence, the rule for variables at level 1 (Rule 5.23) has two cases, where $\mathrm{Dom}(\Delta)$ is the set of variables in the domain of Δ. In Rule 5.25, $\mathrm{head}(p)$ is used to reference the component of a locally defined module within code of a module. The definition of $\mathrm{head}(p)$ is given in Section 5.2.6. In Rule 5.29-5.31, the multi-stage constructors for core expressions are *not* eliminated because they are MetaOCaml's constructors. In contrast, the constructor \$ is eliminated through the translation (Rule 5.32).

$\boxed{[\![\ e\ ]\!]^l_\Delta}$

$$[\![\ x_i\ ]\!]^0_\Delta = x_i \tag{5.22}$$
$$[\![\ x_i\ ]\!]^1_\Delta = \begin{cases} \sim x_i & (x_i \in \mathrm{Dom}(\Delta)) \\ x_i & (otherwise) \end{cases} \tag{5.23}$$
$$[\![\ p.x\ ]\!]^0_\Delta = [\![\ p\ ]\!]^0.x \tag{5.24}$$
$$[\![\ p.x\ ]\!]^1_\Delta = \begin{cases} \sim ([\![\ p\ ]\!]^1.x) & (\mathrm{head}(p) \in \mathrm{Dom}(\Delta)) \\ [\![\ p\ ]\!]^1.x & (otherwise) \end{cases} \tag{5.25}$$
$$[\![\ \textbf{fun}\ x_i\ \rightarrow\ e\ ]\!]^l_\Delta = \textbf{fun}\ x_i\ \rightarrow\ [\![\ e\ ]\!]^l_\Delta \tag{5.26}$$
$$[\![\ e_1\ e_2\ ]\!]^l_\Delta = [\![\ e_1\ ]\!]^l_\Delta\ [\![\ e_2\ ]\!]^l_\Delta \tag{5.27}$$
$$[\![\ \textbf{let}\ x_i\ =\ e_1\ \textbf{in}\ e_2\ ]\!]^l_\Delta = \textbf{let}\ x_i\ =\ [\![\ e_1\ ]\!]^l_\Delta\ \textbf{in}\ [\![\ e_2\ ]\!]^l_\Delta \tag{5.28}$$
$$[\![\ <e>\ ]\!]^0_\Delta = <[\![\ e\ ]\!]^1_\Delta> \tag{5.29}$$
$$[\![\ \sim e\ ]\!]^1_\Delta = \sim [\![\ e\ ]\!]^0_\Delta \tag{5.30}$$
$$[\![\ \textbf{run}\ e\ ]\!]^0_\Delta = \textbf{run}\ [\![\ e\ ]\!]^0_\Delta \tag{5.31}$$
$$[\![\ \$p.x\ ]\!]^0_\Delta = [\![\ p\ ]\!]^0.x \tag{5.32}$$

Next, we define the translation rules for types. We use the notation $[\![\cdot]\!]^l$ to define the translation for types. The environment Δ is no longer needed. The translation rules for module types are straightforward as follows:

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

$$[\![\textbf{sig } S \textbf{ end}]\!]^l = \textbf{sig } [\![S]\!]^l \textbf{ end} \tag{5.33}$$
$$[\![\textbf{functor } (X_i : M_1) \to M_2]\!]^0 = \textbf{functor } (X_i : [\![M_1]\!]^0) \to [\![M_2]\!]^0 \tag{5.34}$$
$$[\![M \textbf{ mcod}]\!]^0 = [\![M]\!]^1 \tag{5.35}$$

The translation for a sequence of signature components is defined as:

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

$$[\![\epsilon]\!]^l = \epsilon \tag{5.36}$$
$$[\![C\ S]\!]^l = [\![C]\!]^l\ [\![S]\!]^l \tag{5.37}$$

The translation rules for signature components are defined by Rule 5.38-5.42. A type of a value component at level 1 is translated to a **code** type by Rule 5.39.

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

$$[\![\textbf{val } x_i : \tau]\!]^0 = \textbf{val } x_i : [\![\tau]\!]^0 \tag{5.38}$$
$$[\![\textbf{val } x_i : \tau]\!]^1 = \textbf{val } x_i : [\![\tau]\!]^1 \textbf{ code} \tag{5.39}$$
$$[\![\textbf{type } t_i]\!]^l = \textbf{type } t_i \tag{5.40}$$
$$[\![\textbf{type } t_i \ = \ \tau]\!]^l = \textbf{type } t_i \ = \ [\![\tau]\!]^l \tag{5.41}$$
$$[\![\textbf{module } X_i : M]\!]^l = \textbf{module } X_i : [\![M]\!]^l \tag{5.42}$$

The translation for core types simply eliminates the symbol \$. As the language $\lambda^{<M_A>}$ implicitly performs CSP for types, Rule 5.47 is defined at both levels 0 and 1.

$\boxed{[\![\tau]\!]^l}$

$$[\![t_i]\!]^l = t_i \tag{5.43}$$
$$[\![p.t]\!]^l = [\![p]\!]^l.t \tag{5.44}$$
$$[\![\tau_1 \ \to \ \tau_2]\!]^l = [\![\tau_1]\!]^l \ \to \ [\![\tau_2]\!]^l \tag{5.45}$$
$$[\![\tau \textbf{ code}]\!]^0 = [\![\tau]\!]^0 \textbf{ code} \tag{5.46}$$
$$[\![\$p.t]\!]^l = [\![p]\!]^l.t \tag{5.47}$$

Paths are kept intact, except for eliminating \$. The rules for paths are defined as follows:

$[\![\ p\ ]\!]^l$

$$[\![\ X_i\ ]\!]^l = X_i \tag{5.48}$$
$$[\![\ p.X\ ]\!]^l = [\![\ p\ ]\!]^l.X \tag{5.49}$$
$$[\![\ \$p.X\ ]\!]^l = [\![\ p\ ]\!]^l.X \tag{5.50}$$
$$[\![\ p_1(p_2)\ ]\!]^l = [\![\ p_1\ ]\!]^l([\![\ p_2\ ]\!]^l) \tag{5.51}$$

The translation rule for programs removes **prog** and **end**, and translates the sequence of structure components s.

$[\![\ P\ ]\!]^0_\Delta$

$$[\![\ \textbf{prog}\ s\ \textbf{end}\ ]\!]^0 = [\![\ s\ ]\!]^0_\Delta \tag{5.52}$$

Finally, we define the translation for environments. An environment is translated element-wise. Since each element of an environment carries its level, we do not need to annotate the translation with a level. After the translation, value components and module components in code of a module are defined at level 0. Hence, those components in an environment should be changed to level 0 and a type of code. Rule 5.56 and 5.59 perform it. Note that bindings inside ordinary brackets remain at level 1. Other components are kept intact.

$[\![\ E\ ]\!]_\Delta$

$$[\![\ \epsilon\ ]\!]_\Delta = \epsilon \tag{5.53}$$
$$[\![\ C^l;\ E\ ]\!]_\Delta = [\![\ C^l\ ]\!]_\Delta;\ [\![\ E\ ]\!]_\Delta \tag{5.54}$$

$[\![\ C^l\ ]\!]_\Delta$

$$[\![\ (\textbf{val}\ x_i : \tau)^0\ ]\!]_\Delta = (\textbf{val}\ x_i : [\![\ \tau\ ]\!]^0)^0 \tag{5.55}$$
$$[\![\ (\textbf{val}\ x_i : \tau)^1\ ]\!]_\Delta = \begin{cases} (\textbf{val}\ x_i : [\![\ \tau\ ]\!]^1\ \textbf{code})^0 & (x_i \in \mathrm{Dom}(\Delta)) \\ (\textbf{val}\ x_i : [\![\ \tau\ ]\!]^1)^1 & (\mathit{otherwise}) \end{cases} \tag{5.56}$$
$$[\![\ (\textbf{type}\ t_i\ =\ \tau)^l\ ]\!]_\Delta = (\textbf{type}\ t_i\ =\ [\![\ \tau\ ]\!]^l)^l \tag{5.57}$$
$$[\![\ (\textbf{module}\ X_i : M)^0\ ]\!]_\Delta = (\textbf{module}\ X_i : [\![\ M\ ]\!]^0)^0 \tag{5.58}$$
$$[\![\ (\textbf{module}\ X_i : M)^1\ ]\!]_\Delta = (\textbf{module}\ X_i : [\![\ M\ ]\!]^1)^0 \tag{5.59}$$

5.4 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 Δ, which is obtained by simply removing Δ from our type system. Second, the target type system has a typing rule for genlet. The typing rule for genlet is defined below.

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

In this section, we first give some lemmas to prove preservation, then give the proof of preservation.

Lemma 1 If $[\![\tau]\!]^l$, $[\![p]\!]^l$, and $[\![S]\!]^l$ are defined, then $\mathit{subst}^\bullet([\![\tau]\!]^l, \ [\![p]\!]^l, \ [\![S]\!]^l) = [\![\mathit{subst}^\bullet(\tau, \ p, \ S)]\!]^l$.

By the translation $[\![\tau]\!]^l$, dollars (\$) that appear in the τ are simply translated to dots (.). Also, $[\![p]\!]^l$ simply removes dollars. In addition, $\mathbf{Dom}(S)$ and $\mathbf{Dom}([\![S]\!]^l)$ contain the same component names. Hence, we obtain $\mathit{subst}^\bullet([\![\tau]\!]^l, \ [\![p]\!]^l, \ [\![S]\!]^l) = [\![\mathit{subst}^\bullet(\tau, \ p, \ S)]\!]^l$

We can obtain the following Lemma 2 by extending Lemma 1.

Lemma 2 If $[\![\tau]\!]^l$, $[\![p]\!]^l$, and $[\![S]\!]^l$ are defined, then $\mathit{subst}^\bullet([\![\tau]\!]^l, \ [\![p]\!]^l, \ [\![S]\!]^l) = [\![\mathit{subst}^\$(\tau, \ p, \ S)]\!]^0$.

If $[\![\tau]\!]^l$ is defined on any level l, the results of $[\![\tau]\!]^0$ and $[\![\tau]\!]^1$ are the same. The difference between $\mathit{subst}^\bullet(\tau, \ p, \ S)$ and $\mathit{subst}^\$(\tau, \ p, \ S)$ is only whether dots are used or dollars are used to access the path p. From the above and Lemma 1, we can prove this lemma.

Lemma 3 If $[\![M]\!]^l$, $[\![p]\!]^l$, and $[\![S]\!]^l$ are defined, then $\mathit{subst}^\bullet([\![M]\!]^l, \ [\![p]\!]^l, \ [\![S]\!]^l) = [\![\mathit{subst}^\bullet(M, \ p, \ S)]\!]^l$.

We can prove this lemma in the same way as Lemma 1.

Lemma 4 If $[\![M]\!]^l$, $[\![p]\!]^l$, and $[\![S]\!]^l$ are defined, then $\mathit{subst}^\bullet([\![M]\!]^l, \ [\![p]\!]^l, \ [\![S]\!]^l) = [\![\mathit{subst}^\$(M, \ p, \ S)]\!]^l$.

We can prove this lemma in the same way to Lemma 2. Note that the level l in the translation $[\![\mathit{subst}^\$(M, \ p, \ S)]\!]^l$ is not 0 in contrast to Lemma 2. The result of $[\![M]\!]^1$ may include code of value components, but the result of $[\![M]\!]^0$ does not. Therefore, M should be translated at the same level l.

Lemma 5 For a module type M, an identifier X_i, and a path p, $[\![M[X_i \leftarrow p]]\!]^0 = [\![M]\!]^0[X_i \leftarrow [\![p]\!]^0]$.

The proof is straightforward.

Lemma 6 For a signature S and an identifier X_i, $(\textbf{module}\ X_i : \textbf{sig}\ [\![\ S\ ]\!]^1\ \textbf{end})^0 \vdash^0\ \textbf{struct}\ S /\!\!/ X_i\ \textbf{end}\ :\ \textbf{sig}\ [\![\ S\ ]\!]^0\ \textbf{end}$ is derivable.

The proof goes by induction on the translation for $S /\!\!/ X_i$.

- Case: Rule 5.11.
  The proof is straightforward.

- Case: Rule 5.12.
  The type of $X_i.x$ is $\tau\ \textbf{code}$, and the type of $\textbf{run}\ X_i.x$ is τ. By the induction hypothesis, we obtain $(\textbf{module}\ X_i : \textbf{sig}\ [\![\ S\ ]\!]^1\ \textbf{end})^0 \vdash^0\ \textbf{struct}\ (\textbf{let}\ x_i : \tau\ =\ \textbf{run}\ X_i.x)\ S /\!\!/ X_i\ \textbf{end}\ :\ \textbf{sig}\ (\textbf{val}\ x_i : \tau)\ [\![\ S\ ]\!]^0\ \textbf{end}$.

- Case: Rule 5.13.
  The type of $X_i.t$ is τ. By the induction hypothesis, we obtain $(\textbf{module}\ X_i : \textbf{sig}\ [\![\ S\ ]\!]^1\ \textbf{end})^0 \vdash^0\ \textbf{struct}\ (\textbf{type}\ t_i =\ X_i.t)\ S /\!\!/ X_i\ \textbf{end}\ :\ \textbf{sig}\ (\textbf{type}\ t_i\ =\ \tau)\ [\![\ S\ ]\!]^0\ \textbf{end}$.

- Case: Rule 5.14.
  The result of the translation $[\![\ \textbf{Runmod}\ (X_i.X_i' : M\ \textbf{mcod})\ ]\!]^0_\epsilon$ is $\textbf{struct}\ (\textbf{module}\ Y_i = X_i.X')\ S' /\!\!/ Y_i\ \textbf{end}$, where $M = \textbf{sig}\ S'\ \textbf{end}$ and Y_i is a fresh identifier. By the induction hypothesis, we obtain $(\textbf{module}\ X_i : \textbf{sig}\ [\![\ S\ ]\!]^1\ \textbf{end})^0 \vdash^0\ \textbf{struct}\ (\textbf{module}\ X_i' = \textbf{struct}\ (\textbf{module}\ Y_i\ =\ X_i.X')\ S' /\!\!/ Y_i\ \textbf{end})\ S /\!\!/ X_i\ \textbf{end}\ :\ \textbf{sig}\ (\textbf{module}\ X_i' : \textbf{sig}\ [\![\ S'\ ]\!]^0\ \textbf{end})\ [\![\ S\ ]\!]^0\ \textbf{end}$.

Lemma 7 If $E; \Delta \vdash^l\ M\ \textbf{\textit{wf}}$, then $[\![\ E\ ]\!]_\Delta \vdash^l\ [\![\ M\ ]\!]^l\ \textbf{\textit{wf}}$

The proof goes by induction on the well-formedness derivation.

Lemma 8 If $E; \Delta \vdash^l\ M_1\ <:\ M_2$, then $[\![\ E\ ]\!]_\Delta \vdash^l\ [\![\ M_1\ ]\!]^l\ <:\ [\![\ M_2\ ]\!]^l$

The proof goes by induction on the subtyping derivation.

Theorem 1 If $E; \Delta \vdash^l\ e_0\ :\ \tau_0$, then $[\![\ E\ ]\!]_\Delta \vdash^l\ [\![\ e_0\ ]\!]^l_\Delta\ :\ [\![\ \tau_0\ ]\!]^l$

Proof 1 The proof goes by induction on the type derivation of e_0. First, we suppose $l = 0$.

- Case: E-Code.
  We have the conclusion $E; \Delta \vdash^0\ <\ e\ >\ :\ \tau\ \textbf{code}$ and the subderivation $E; \Delta \vdash^1\ e\ :\ \tau$. By the induction hypothesis, we have $[\![\ E\ ]\!]_\Delta \vdash^1\ [\![\ e\ ]\!]^1_\Delta\ :\ [\![\ \tau\ ]\!]^1$. By Rule 5.29, $[\![\ <e>\ ]\!]^0_\Delta = <[\![\ e\ ]\!]^1_\Delta>$. By Rule 5.46, $[\![\ \tau\ \textbf{code}\ ]\!]^0 = [\![\ \tau\ ]\!]^0\ \textbf{code}$. By the definition of $[\![\ \tau\ ]\!]^l$, we have $[\![\ \tau\ ]\!]^0 = [\![\ \tau\ ]\!]^1$. Hence, we obtain $[\![\ E\ ]\!]_\Delta \vdash^0\ [\![\ <e>\ ]\!]^0_\Delta\ :\ [\![\ \tau\ \textbf{code}\ ]\!]^0$.

- Case: E-Esc.
  This rule does not exist for $l = 0$.

- Case: E-Run.
  We have the conclusion $E; \Delta \vdash^0\ \textbf{run}\ e\ :\ \tau$ and the subderivation $E; \Delta \vdash^0\ e\ :\ \tau\ \textbf{code}$.

By the induction hypothesis, we have $[\![E]\!]_\Delta \vdash^0 [\![e]\!]^0_\Delta : [\![\tau \ \mathbf{code}]\!]^0$. By Rule 5.31, $[\![\mathbf{run}\ e]\!]^0_\Delta = \mathbf{run}\ [\![e]\!]^0_\Delta$. By Rule 5.46, $[\![\tau \ \mathbf{code}]\!]^0 = [\![\tau]\!]^0\ \mathbf{code}$. Hence, we obtain $[\![E]\!]_\Delta \vdash^0 [\![\mathbf{run}\ e]\!]^0_\Delta : [\![\tau]\!]^0$.

- Case: E-Fun, E-App, E-Let, and E-Substitution.
  The proofs are straightforward.

- Case: E-Var.
  We have the conclusion $E; \Delta \vdash^0 x_i : \tau$ and the subderivation $(\mathbf{val}\ x_i : \tau)^0 \in E$. By Rule 5.22, $[\![x_i]\!]^0_\Delta = x_i$. By Rule 5.55, $[\![(\mathbf{val}\ x_i : \tau)^0]\!]_\Delta = (\mathbf{val}\ x_i : [\![\tau]\!]^0)^0$. Thus, we have $(\mathbf{val}\ x_i : [\![\tau]\!]^0)^0 \in [\![E]\!]_\Delta$. Hence, we obtain $[\![E]\!]_\Delta \vdash^0 [\![x_i]\!]^0_\Delta : [\![\tau]\!]^0$.

- Case: E-Dot.
  We have the conclusion $E; \Delta \vdash^0 p.x : \tau[z_j \leftarrow p.z \mid z_j \in \mathbf{Dom}(S_1)]$ and the subderivation $E; \Delta \vdash^0 p : \mathbf{sig}\ S_1\ (\mathbf{val}\ x_i : \tau)\ S_2\ \mathbf{end}$. By the induction hypothesis for module expressions, we have $[\![E]\!]_\Delta \vdash^0 [\![p]\!]^0 : [\![\mathbf{sig}\ S_1\ (\mathbf{val}\ x_i : \tau)\ S_2\ \mathbf{end}]\!]^0$. By Rule 5.33 and 5.38, $[\![\mathbf{sig}\ S_1\ (\mathbf{val}\ x_i : \tau)\ S_2\ \mathbf{end}]\!]^0 = \mathbf{sig}\ [\![S_1]\!]^0\ (\mathbf{val}\ x_i : [\![\tau]\!]^0)\ [\![S_2]\!]^0\ \mathbf{end}$. By Lemma 1, $\mathit{subst}^\bullet([\![\tau]\!]^0, [\![p]\!]^0, [\![S_1]\!]^0) = [\![\mathit{subst}^\bullet(\tau, p, S_1)]\!]^0$. By Rule 5.24, $[\![p.x]\!]^0_\Delta = [\![p]\!]^0.x$. Hence, we obtain $[\![E]\!]_\Delta \vdash^0 [\![p.x]\!]^0_\Delta : [\![\tau[z_j \leftarrow p.z \mid z_j \in \mathbf{Dom}(S_1)]]\!]^0$.

- Case: E-DotCode.
  We have the conclusion $E; \Delta \vdash^0 \$p.x : \tau[z_j \leftarrow \$p.z \mid z_j \in \mathbf{Dom}(S_1)]\ \mathbf{code}$ and the subderivation $E; \Delta \vdash^0 p : (\mathbf{sig}\ S_1\ (\mathbf{val}\ x_i : \tau)\ S_2\ \mathbf{end})\ \mathbf{mcod}$. By the induction hypothesis for module expressions, we have $[\![E]\!]_\Delta \vdash^0 [\![p]\!]^0 : [\![(\mathbf{sig}\ S_1\ (\mathbf{val}\ x_i : \tau)\ S_2\ \mathbf{end})\ \mathbf{mcod}]\!]^0$. By Rule 5.32, $[\![\$p.x]\!]^0_\Delta = [\![p]\!]^0.x$. By Rule 5.35 5.33, and 5.39, $[\![(\mathbf{sig}\ S_1\ (\mathbf{val}\ x_i : \tau)\ S_2\ \mathbf{end})\ \mathbf{mcod}]\!]^0 = \mathbf{sig}\ [\![S_1]\!]^1\ (\mathbf{val}\ x_i : [\![\tau]\!]^1\ \mathbf{code})\ [\![S_2]\!]^1\ \mathbf{end}$. Thus, the following typing can be derived in the target language.

$$\frac{\vdots}{[\![E]\!]_\Delta \vdash^0 [\![p]\!]^0 : \mathbf{sig}\ [\![S_1]\!]^1\ (\mathbf{val}\ x_i : [\![\tau]\!]^1\ \mathbf{code})\ [\![S_2]\!]^1\ \mathbf{end}}{[\![E]\!]_\Delta \vdash^0 [\![p]\!]^0.x : ([\![\tau]\!]^1\ \mathbf{code})[z_j \leftarrow [\![p]\!]^0.z \mid z_j \in \mathbf{Dom}([\![S_1]\!]^1)]}$$

  The type of the conclusion in the above derivation is $\mathit{subst}^\bullet([\![\tau]\!]^1, [\![p]\!]^0, [\![S_1]\!]^1)\ \mathbf{code}$. By the definition, we have $[\![p]\!]^0 = [\![p]\!]^1$. We can apply Lemma 2 to them, then we obtain $\mathit{subst}^\bullet([\![\tau]\!]^1, [\![p]\!]^1, [\![S_1]\!]^1) = [\![\mathit{subst}^\$(\tau, p, S_1)]\!]^0$. Hence, we obtain $[\![E]\!]_\Delta \vdash^0 [\![\$p.x]\!]^0_\Delta : [\![\tau[z_j \leftarrow \$p.z \mid z_j \in \mathbf{Dom}(S_1)]\ \mathbf{code}]\!]^0$.

Then, we suppose $l = 1$.

- Case: E-Var.
  We have the conclusion $E; \Delta \vdash^1 x_i : \tau$ and the subderivation $(\mathbf{val}\ x_i : \tau)^1 \in E$. We divide the proof into two cases.

– Case: $x_i \in \mathbf{Dom}(\Delta)$. By Rule 5.23, $[\![\ x_i\ ]\!]^1_\Delta = \sim x_i$. By Rule 5.56, $(\mathbf{val}\ x_i : [\![\ \tau\ ]\!]^1\ \mathbf{code})^0 \in [\![\ E\ ]\!]_\Delta$. Thus, we obtain $[\![\ E\ ]\!]_\Delta \vdash^0 x_i : [\![\ \tau\ ]\!]^1\ \mathbf{code}$, then we obtain $[\![\ E\ ]\!]_\Delta \vdash^1 \sim x_i : [\![\ \tau\ ]\!]^1$ by E-Esc.

– Case: $x_i \notin \mathbf{Dom}(\Delta)$. By Rule 5.23, $[\![\ x_i\ ]\!]^1_\Delta = x_i$. By Rule 5.56, $(\mathbf{val}\ x_i : [\![\ \tau\ ]\!]^1)^1 \in [\![\ E\ ]\!]_\Delta$. Thus, we obtain $[\![\ E\ ]\!]_\Delta \vdash^1 x_i : [\![\ \tau\ ]\!]^1$.

Hence, in either cases, we obtain $[\![\ E\ ]\!]_\Delta \vdash^1 [\![\ x_i\ ]\!]^1_\Delta : [\![\ \tau\ ]\!]^1$.

- Case: E-Esc.
We have the conclusion $E; \Delta \vdash^1 \sim e : \tau$ and the subderivation $E; \Delta \vdash^0 e : \tau\ \mathbf{code}$. By the induction hypothesis, we have $[\![\ E\ ]\!]_\Delta \vdash^0 [\![\ e\ ]\!]^0_\Delta : [\![\ \tau\ \mathbf{code}\ ]\!]^0$. By Rule 5.30, $[\![\ \sim e\ ]\!]^1_\Delta = \sim [\![\ e\ ]\!]^0_\Delta$. By Rule 5.46, $[\![\ \tau\ \mathbf{code}\ ]\!]^0 = [\![\ \tau\ ]\!]^0\ \mathbf{code}$. By the definition of $[\![\ \tau\ ]\!]^l$, we have $[\![\ \tau\ ]\!]^0 = [\![\ \tau\ ]\!]^1$. Hence, we obtain $[\![\ E\ ]\!]_\Delta \vdash^1 [\![\ \sim e\ ]\!]^1_\Delta : [\![\ \tau\ ]\!]^1$.

- Case: E-Dot.
We have the conclusion $E; \Delta \vdash^1 p.x : \tau\,[z_j \leftarrow p.z \mid z_j \in \mathbf{Dom}(S_1)]$ and the subderivation $E; \Delta \vdash^1 p : \mathbf{sig}\ S_1\ (\mathbf{val}\ x_i : \tau)\ S_2\ \mathbf{end}$. By the induction hypothesis for module expressions, we have $[\![\ E\ ]\!]_\Delta \vdash^0 [\![\ p\ ]\!]^1 : [\![\ \mathbf{sig}\ S_1\ (\mathbf{val}\ x_i : \tau)\ S_2\ \mathbf{end}\ ]\!]^1$. We divide the proof into two cases.

  – Case: $\mathrm{head}(p) \in \mathbf{Dom}(\Delta)$. By Rule 5.25, $[\![\ p.x\ ]\!]^1_\Delta = \sim ([\![\ p\ ]\!]^1.x)$. By Rule 5.33 and 5.39, $[\![\ \mathbf{sig}\ S_1\ (\mathbf{val}\ x_i : \tau)\ S_2\ \mathbf{end}\ ]\!]^1 = \mathbf{sig}\ [\![\ S_1\ ]\!]^1\ (\mathbf{val}\ x_i : [\![\ \tau\ ]\!]^1\ \mathbf{code})\ [\![\ S_2\ ]\!]^1\ \mathbf{end}$. Therefore, we can construct the following derivation:

$$\dfrac{\dfrac{[\![\ E\ ]\!]_\Delta \vdash^0 [\![\ p\ ]\!]^1 : \mathbf{sig}\ [\![\ S_1\ ]\!]^1\ (\mathbf{val}\ x_i : [\![\ \tau\ ]\!]^1\ \mathbf{code})\ [\![\ S_2\ ]\!]^1\ \mathbf{end}}{[\![\ E\ ]\!]_\Delta \vdash^0 [\![\ p\ ]\!]^1.x : [\![\ \tau\ ]\!]^1[z_j \leftarrow [\![\ p\ ]\!]^1.z \mid z_j \in \mathbf{Dom}([\![\ S_1\ ]\!]^1)]\ \mathbf{code}}}{[\![\ E\ ]\!]_\Delta \vdash^1 \sim([\![\ p\ ]\!]^1.x) : [\![\ \tau\ ]\!]^1[z_j \leftarrow [\![\ p\ ]\!]^1.z \mid z_j \in \mathbf{Dom}([\![\ S_1\ ]\!]^1)]}$$

  By Lemma 1, $\mathit{subst}^\bullet([\![\ \tau\ ]\!]^1, [\![\ p\ ]\!]^1, [\![\ S_1\ ]\!]^1) = [\![\ \mathit{subst}^\bullet(\tau,\ p,\ S_1)\ ]\!]^1$.

  – Case: $\mathrm{head}(p) \notin \mathbf{Dom}(\Delta)$. By Rule 5.25, $[\![\ p.x\ ]\!]^1_\Delta = [\![\ p\ ]\!]^1.x$. By Rule 5.33 and 5.39, $[\![\ \mathbf{sig}\ S_1\ (\mathbf{val}\ x_i : \tau)\ S_2\ \mathbf{end}\ ]\!]^1 = \mathbf{sig}\ [\![\ S_1\ ]\!]^1\ (\mathbf{val}\ x_i : [\![\ \tau\ ]\!]^1)\ [\![\ S_2\ ]\!]^1\ \mathbf{end}$. By Lemma 1, $\mathit{subst}^\bullet([\![\ \tau\ ]\!]^1, [\![\ p\ ]\!]^1, [\![\ S_1\ ]\!]^1) = [\![\ \mathit{subst}^\bullet(\tau,\ p,\ S_1)\ ]\!]^1$.

  Hence, in either cases, we obtain $[\![\ E\ ]\!]_\Delta \vdash^1 [\![\ p.x\ ]\!]^1_\Delta : [\![\ \tau\,[z_j \leftarrow p.z \mid z_j \in \mathbf{Dom}(S_1)]\ ]\!]^1$.

- Case: E-DotCode, E-Code, and E-Run.
This rule does not exist for $l = 1$.

- Case: E-Fun, E-App, E-Let, and E-Substitution.
The proofs are straightforward.

Now we can prove the type preservation for module expressions. If $E; \Delta \vdash^l m : M$, then $[\![\ E\ ]\!]_\Delta \vdash^0 [\![\ m\ ]\!]^l_\Delta : [\![\ M\ ]\!]^l$. Note that translated expressions are typed at level 0 instead of l. The target language (MetaOCaml) allows module expressions to be typed at level 0 only.

The proof goes by induction on the type derivation of m. First, we suppose $l = 0$.

- Case: M-Cod.
  We have the conclusion $E; \Delta \vdash^0 \langle\!\langle m \rangle\!\rangle \ : \ M$ **mcod** and the subderivation $E; \Delta \vdash^1 \ m \ : \ M$. By the induction hypothesis, we have $[\![\ E \ ]\!]_\Delta \vdash^0 \ [\![\ m \ ]\!]^1_\Delta \ : \ [\![\ M \ ]\!]^1$. By Rule 5.7, $[\![\ \langle\!\langle m \rangle\!\rangle \ ]\!]^0_\Delta = [\![\ m \ ]\!]^1_\Delta$. By Rule 5.35, $[\![\ M$ **mcod** $]\!]^0 = [\![\ M \ ]\!]^1$. we obtain $[\![\ E \ ]\!]_\Delta \vdash^0 \ [\![\ \langle\!\langle m \rangle\!\rangle \ ]\!]^0_\Delta \ : \ [\![\ M$ **mcod** $]\!]^0$.

- Case: M-Esc.
  This rule does not exist for $l = 0$.

- Case: M-Runmod.
  We have the conclusion $E; \Delta \vdash^0$ **Runmod** $(m : M$ **mcod**$) \ : \ M$ and the subderivation $E; \Delta \vdash^0 \ m \ : \ M$ **mcod**. By the induction hypothesis, we have $[\![\ E \ ]\!]_\Delta \vdash^0 \ [\![\ m \ ]\!]^0_\Delta \ : \ [\![\ M$ **mcod** $]\!]^0$. We have a signature S for **sig** S **end** $= M$, $[\![$ **Runmod** $(m : M$ **mcod**$) \ ]\!]^0_\Delta$ is translated to a structure that satisfies S by Lemma 6. We have $[\![$ **sig** S **end** $\ ]\!]^0 = [\![\ M \ ]\!]^0$. Hence, we obtain $[\![\ E \ ]\!]_\Delta \vdash^0 \ [\![$ **Runmod** $(m : M$ **mcod**$) \ ]\!]^0_\Delta \ : \ [\![\ M \ ]\!]^0$.

- Case: M-Var.
  We have the conclusion $E; \Delta \vdash^0 \ X_i \ : \ M$ and the subderivation (**module** $X_i : M)^0 \in E$. By Rule 5.1, $[\![\ X_i \ ]\!]^0_\Delta = X_i$. By Rule 5.55, $[\![\ ($**module** $X_i : M)^0 \ ]\!]^0 = ($**module** $X_i : [\![\ M \ ]\!]^0)^0$. Thus, we have (**module** $X_i : [\![\ M \ ]\!]^0)^0 \in [\![\ E \ ]\!]_\Delta$. Hence, we obtain $[\![\ E \ ]\!]_\Delta \vdash^0 \ [\![\ X_i \ ]\!]^0_\Delta \ : \ [\![\ M \ ]\!]^0$.

- Case: M-Dot.
  We have the conclusion $E; \Delta \vdash^0 \ p.X \ : \ M \, [z_j \leftarrow p.z \mid z_j \in \mathbf{Dom}(S_1)]$ and the subderivation $E; \Delta \vdash^0 \ p \ :$ **sig** S_1 (**module** $X_i : M$) S_2 **end**. By Rule 5.2, $[\![\ p.X \ ]\!]^0_\Delta = [\![\ p \ ]\!]^0.X$. By the induction hypothesis, we have $[\![\ E \ ]\!]_\Delta \vdash^0 \ [\![\ p \ ]\!]^0 \ : \ [\![$ **sig** S_1 (**module** $X_i : M$) S_2 **end** $\ ]\!]^0$. By Rule 5.33 and 5.42, $[\![$ **sig** S_1 (**module** $X_i : M$) S_2 **end** $\ ]\!]^0 =$ **sig** $[\![\ S_1 \ ]\!]^0$ (**module** $X_i : [\![\ M \ ]\!]^0)$ $[\![\ S_2 \ ]\!]^0$ **end**. By Lemma 3, $[\![\ subst^\bullet \, (M, \ p, \ S_1) \ ]\!]^0 = subst^\bullet \, ([\![\ M \ ]\!]^0, \ [\![\ p \ ]\!]^0, \ [\![\ S_1 \ ]\!]^0)$. Hence, we obtain $[\![\ E \ ]\!]_\Delta \vdash^0 \ [\![\ p.X \ ]\!]^0_\Delta \ : \ [\![\ M \, [z_j \leftarrow p.z \mid z_j \in \mathbf{Dom}(S_1)] \ ]\!]^0$.

- Case: M-DotCode.
  We have the conclusion $E; \Delta \vdash^0 \ \$p.X \ : \ M \, [z_j \leftarrow \$p.z \mid z_j \in \mathbf{Dom}(S_1)]$ **mcod** and the subderivation $E; \Delta \vdash^0 \ p \ :$ (**sig** S_1 (**module** $X_i : M$) S_2 **end**) **mcod**. By the induction hypothesis, we have $[\![\ E \ ]\!]_\Delta \vdash^0 \ [\![\ p \ ]\!]^0 \ : \ [\![\ ($**sig** S_1 (**module** $X_i : M$) S_2 **end**$)$ **mcod** $\ ]\!]^0$. By Rule 5.10, $[\![\ \$p.X \ ]\!]^0_\Delta = [\![\ p \ ]\!]^0.X$. By Rule 5.35, $[\![\ ($**sig** S_1 (**module** $X_i : M$) S_2 **end**$)$ **mcod** $\ ]\!]^0 = [\![$ **sig** S_1 (**module** $X_i : M$) S_2 **end** $\ ]\!]^1$. Moreover, this is translated to **sig** $[\![\ S_1 \ ]\!]^1$ (**module** $X_i : [\![\ M \ ]\!]^1)$ $[\![\ S_2 \ ]\!]^1$ **end**. By the definition, we have $[\![\ p \ ]\!]^0 = [\![\ p \ ]\!]^1$. By Lemma 4, $subst^\bullet \, ([\![\ M \ ]\!]^1, \ [\![\ p \ ]\!]^1, \ [\![\ S_1 \ ]\!]^1) = [\![\ subst^\$ \, (M, \ p, \ S_1) \ ]\!]^1$. Hence, we obtain $[\![\ E \ ]\!]_\Delta \vdash^0 \ [\![\ \$p.X \ ]\!]^0_\Delta \ : \ [\![\ M \, [z_j \leftarrow \$p.z \mid z_j \in \mathbf{Dom}(S_1)]$ **mcod** $\ ]\!]^0$.

- Case: M-Functor.
  We have the conclusion $E; \Delta \vdash^0$ **functor** $(X_i : M_2) \to m \ :$ **functor** $(X_i : M_2) \to M_1$, $X_i^0 \notin \mathbf{Dom}(E)$, and the derivation for E, (**module** $X_i : M_2)^0; \Delta \vdash^0 \ m \ : \ M_1$. By the induction hypothesis, we have $[\![\ E, ($**module** $X_i : M_2)^0 \ ]\!]_\Delta \vdash^0 \ [\![\ m \ ]\!]^0_\Delta \ : \ [\![\ M_1 \ ]\!]^0$.

By Rule 5.58, $[\![\ E, (\mathbf{module}\ X_i : M_2)^0\ ]\!]_\Delta = [\![\ E\ ]\!]_\Delta, (\mathbf{module}\ X_i : [\![\ M_2\ ]\!]^0)^0$. In addition, by Rule 5.6 and Rule 5.34, we obtain $[\![\ E\ ]\!]_\Delta \vdash^0 [\![\ \mathbf{functor}\ (X_i : M_2) \to m\ ]\!]^0_\Delta : [\![\ \mathbf{functor}\ (X_i : M_2) \to M_1\ ]\!]^0$.

- Case: M-App.
  We have the conclusion $E; \Delta \vdash^0 m\,(p) : M_1\,[X_i \leftarrow p]$, and the subderivations $E; \Delta \vdash^0 p : M_2$ and $E; \Delta \vdash^0 m : \mathbf{functor}\ (X_i : M_2) \to M_1$. By the induction hypothesis, we have $[\![\ E\ ]\!]_\Delta \vdash^0 [\![\ m\ ]\!]^0_\Delta : [\![\ \mathbf{functor}\ (X_i : M_2) \to M_1\ ]\!]^0$ and $[\![\ E\ ]\!]_\Delta \vdash^0 [\![\ p\ ]\!]^0 : [\![\ M_2\ ]\!]^0$. By Rule 5.34, $[\![\ \mathbf{functor}\ (X_i : M_2) \to M_1\ ]\!]^0 = \mathbf{functor}\ (X_i : [\![\ M_2\ ]\!]^0) \to [\![\ M_1\ ]\!]^0$. By Rule 5.5, $[\![\ m\,(p)\ ]\!]^0_\Delta = [\![\ m\ ]\!]^0_\Delta([\![\ p\ ]\!]^0)$. By Lemma 5, $[\![\ M_1\,[X_i \leftarrow p]\ ]\!]^0 = [\![\ M_1\ ]\!]^0\,[X_i \leftarrow [\![\ p\ ]\!]^0]$. Hence, we obtain $[\![\ E\ ]\!]_\Delta \vdash^0 [\![\ m\,(p)\ ]\!]^0_\Delta : [\![\ M_1\,[X_i \leftarrow p]\ ]\!]^0$.

- Case: M-Subtyping.
  We have the conclusion $E; \Delta \vdash^0 m : M_2$ and the subderivations $E; \Delta \vdash^0 m : M_1$ and $E; \Delta \vdash^0 M_1 <: M_2$. By Lemma 8 and the induction hypothesis, we obtain $[\![\ E\ ]\!]_\Delta \vdash^0 [\![\ m\ ]\!]^0_\Delta : [\![\ M_2\ ]\!]^0$.

- Case: M-Strengthening.
  We have the conclusion $E; \Delta \vdash^0 p : M/p^0$ and the subderivation $E; \Delta \vdash^0 p : M$. By the induction hypothesis, we have $[\![\ E\ ]\!]_\Delta \vdash^0 [\![\ p\ ]\!]^0 : [\![\ M\ ]\!]^0$. By the definition of the strengthening operation, $[\![\ M/p^0\ ]\!]^0 = [\![\ M\ ]\!]^0/([\![\ p\ ]\!]^0)^0$. Hence, we obtain $[\![\ E\ ]\!]_\Delta \vdash^0 [\![\ p\ ]\!]^0_\Delta : [\![\ M/p^0\ ]\!]^0$.

- Case: M-Str.
  We have the conclusion $E; \Delta \vdash^0 \mathbf{struct}\ s\ \mathbf{end} : \mathbf{sig}\ S\ \mathbf{end}$ and the subderivation $E; \Delta \vdash^0 s : S$. By the induction hypothesis, we have $[\![\ E\ ]\!]_\Delta \vdash^0 [\![\ s\ ]\!]^0_\Delta : [\![\ S\ ]\!]^0$. By Rule 5.3 and Rule 5.33, we obtain $[\![\ E\ ]\!]_\Delta \vdash^0 [\![\ \mathbf{struct}\ s\ \mathbf{end}\ ]\!]^0_\Delta : [\![\ \mathbf{sig}\ S\ \mathbf{end}\ ]\!]^0$.

- Case: M-Sig.
  We have the conclusion $E; \Delta \vdash^0 (m : M) : M$ and the subderivations $E; \Delta \vdash^0 M\ \mathbf{wf}$ and $E; \Delta \vdash^0 m : M$. By the induction hypothesis, we have $[\![\ E\ ]\!]_\Delta \vdash^0 [\![\ m\ ]\!]^0_\Delta : [\![\ M\ ]\!]^0$. By Lemma 7, $[\![\ E\ ]\!]_\Delta \vdash^0 [\![\ M\ ]\!]^0\ \mathbf{wf}$. Hence, we obtain $[\![\ E\ ]\!]_\Delta \vdash^0 [\![\ m\ ]\!]^0_\Delta : [\![\ M\ ]\!]^0$.

Then, we suppose $l = 1$.

- Case: M-Esc.
  We have the conclusion $E; \Delta \vdash^1 \approx m : M$ and the subderivation $E; \Delta \vdash^0 m : M\ \mathbf{mcod}$. By the induction hypothesis, we have $[\![\ E\ ]\!]_\Delta \vdash^0 [\![\ m\ ]\!]^0_\Delta : [\![\ M\ \mathbf{mcod}\ ]\!]^0$. By Rule 5.8, $[\![\ \approx m\ ]\!]^1_\Delta = [\![\ m\ ]\!]^0_\Delta$. By Rule 5.35, $[\![\ M\ \mathbf{mcod}\ ]\!]^0 = [\![\ M\ ]\!]^1$. we obtain $[\![\ E\ ]\!]_\Delta \vdash^0 [\![\ \approx m\ ]\!]^1_\Delta : [\![\ M\ ]\!]^1$.

- Case: M-Var.
  We have the conclusion $E; \Delta \vdash^1 X_i : M$ and the subderivation $(\mathbf{module}\ X_i : M)^1 \in E$. By Rule 5.1, $[\![\ X_i\ ]\!]^1_\Delta = X_i$. By Rule 5.59, $[\![\ (\mathbf{module}\ X_i : M)^1\ ]\!]_\Delta = (\mathbf{module}\ X_i : [\![\ M\ ]\!]^1)^0$. Hence, we obtain the subderivation $(\mathbf{module}\ X_i : [\![\ M\ ]\!]^1)^0 \in [\![\ E\ ]\!]_\Delta$, then $[\![\ E\ ]\!]_\Delta \vdash^0 [\![\ X_i\ ]\!]^1_\Delta : [\![\ M\ ]\!]^1$.

- Case: M-Dot.
  We have the conclusion $E; \Delta \vdash^1 \ p.X \ : \ M\,[z_j \leftarrow p.z \mid z_j \in \mathbf{Dom}(S_1)]$ and the subderivation $E; \Delta \vdash^1 \ p \ : \ \mathbf{sig}\ S_1\ (\mathbf{module}\ X_i : M)\ S_2\ \mathbf{end}$. By Rule 5.2, $[\![\ p.X\ ]\!]^1_\Delta = [\![\ p\ ]\!]^1.X$. By the induction hypothesis, we have $[\![\ E\ ]\!]_\Delta \vdash^0 \ [\![\ p\ ]\!]^1 \ : \ [\![\ \mathbf{sig}\ S_1\ (\mathbf{module}\ X_i : M)\ S_2\ \mathbf{end}\ ]\!]^1$. By Rule 5.33 and 5.42, $[\![\ \mathbf{sig}\ S_1\ (\mathbf{module}\ X_i : M)\ S_2\ \mathbf{end}\ ]\!]^1 = \mathbf{sig}\ [\![\ S_1\ ]\!]^1\ (\mathbf{module}\ X_i : [\![\ M\ ]\!]^1)\ [\![\ S_2\ ]\!]^1\ \mathbf{end}$. By Lemma 3, $[\![\ \mathit{subst}^\bullet\,(M,\ p,\ S_1)\ ]\!]^1 = \mathit{subst}^\bullet\,([\![\ M\ ]\!]^1,\ [\![\ p\ ]\!]^1,\ [\![\ S_1\ ]\!]^1)$. Hence, we obtain $[\![\ E\ ]\!]_\Delta \vdash^0 \ [\![\ p.X\ ]\!]^1_\Delta \ : \ [\![\ M\,[z_j \leftarrow p.z \mid z_j \in \mathbf{Dom}(S_1)]\ ]\!]^1$.

- Case: M-DotCode, M-Functor, M-App, M-Strengthening, M-Cod, and M-Runmod.
  This rule does not exist for $l = 1$.

- Case: M-Str, M-Sig, M-Subtyping.
  These cases are the same as $l = 0$ and omitted.

Finally, we prove that if $E; \Delta \vdash^l \ s \ : \ S$ then $[\![\ E\ ]\!]_\Delta \vdash^0 \ [\![\ s\ ]\!]^l_\Delta \ : \ [\![\ S\ ]\!]^l$. Note that the typing judgment in the target language is annotated with level 0 only.

- Case: S-Empty.
  The proof is straightforward.

- Case: S-Let0.
  We have the conclusion $E; \Delta \vdash^0 \ (\mathbf{let}\ x_i : \tau \ = \ e)\ s \ : \ (\mathbf{val}\ x_i : \tau)\ S$ and the subderivations $E; \Delta \vdash^0 \ e \ : \ \tau$, ${x_i}^0 \notin \mathbf{Dom}(E)$, and $E,\ (\mathbf{val}\ x_i : \tau)^0; \Delta \vdash^0 \ s \ : \ S$. By the induction hypothesis, we have $[\![\ E\ ]\!]_\Delta \vdash^0 \ [\![\ e\ ]\!]^0_\Delta \ : \ [\![\ \tau\ ]\!]^0$ and $[\![\ E, (\mathbf{val}\ x_i : \tau)^0\ ]\!]_\Delta \vdash^0 \ [\![\ s\ ]\!]^0_\Delta \ : \ [\![\ S\ ]\!]^0$. We have ${x_i}^0 \notin \mathbf{Dom}([\![\ E\ ]\!]_\Delta)$. By Rule 5.16 and Rule 5.18, $[\![\ (\mathbf{let}\ x_i : \tau \ = \ e)\ s\ ]\!]^0_\Delta = (\mathbf{let}\ x_i : [\![\ \tau\ ]\!]^0 \ = \ [\![\ e\ ]\!]^0_\Delta)\ [\![\ s\ ]\!]^0_\Delta$. By Rule 5.37 and Rule 5.38, $[\![\ (\mathbf{val}\ x_i : \tau)\ S\ ]\!]^0 = (\mathbf{val}\ x_i \ : \ [\![\ \tau\ ]\!]^0)\ [\![\ S\ ]\!]^0$. By Rule 5.54 and Rule 5.55, $[\![\ E, (\mathbf{val}\ x_i : \tau)^0\ ]\!]_\Delta = [\![\ E\ ]\!]_\Delta, (\mathbf{val}\ x_i \ : \ [\![\ \tau\ ]\!]^0)^0$. Hence, we obtain $[\![\ E\ ]\!]_\Delta \vdash^0 \ [\![\ (\mathbf{let}\ x_i : \tau \ = \ e)\ s\ ]\!]^0_\Delta \ : \ [\![\ (\mathbf{val}\ x_i : \tau)\ S\ ]\!]^0$.

- Case: S-Let1.
  We have the conclusion $E; \Delta \vdash^1 \ (\mathbf{let}\ x_i : \tau \ = \ e)\ s \ : \ (\mathbf{val}\ x_i : \tau)\ S$, and the subderivations $E; \Delta \vdash^1 \ e \ : \ \tau$, ${x_i}^1 \notin \mathbf{Dom}(E)$, and $E, (\mathbf{val}\ x_i : \tau)^1; \Delta, (\mathbf{val}\ x_i : \tau)^1 \vdash^1 \ s \ : \ S$. Let Δ' be $\Delta, (\mathbf{val}\ x_i : \tau)^1$. By the induction hypothesis, we have $[\![\ E\ ]\!]_\Delta \vdash^1 \ [\![\ e\ ]\!]^1_\Delta \ : \ [\![\ \tau\ ]\!]^1$ and $[\![\ E, (\mathbf{val}\ x_i : \tau)^1\ ]\!]_{\Delta'} \vdash^0 \ [\![\ s\ ]\!]^1_{\Delta'} \ : \ [\![\ S\ ]\!]^1$. We have ${x_i}^1 \notin \mathbf{Dom}([\![\ E\ ]\!]_\Delta)$. By Rule 5.17 and Rule 5.19, $[\![\ (\mathbf{let}\ x_i : \tau \ = \ e)\ s\ ]\!]^1_\Delta = (\mathbf{let}\ x_i \ : \ [\![\ \tau\ ]\!]^1\ \mathbf{code} \ = \ \mathbf{genlet}\ < [\![\ e\ ]\!]^1_\Delta >)\ [\![\ s\ ]\!]^1_{\Delta'}$. By Rule 5.37 and Rule 5.39, $[\![\ (\mathbf{val}\ x_i : \tau)\ S\ ]\!]^1 = (\mathbf{val}\ x_i \ : \ [\![\ \tau\ ]\!]^1\ \mathbf{code})\ [\![\ S\ ]\!]^1$. By Rule 5.54 and Rule 5.56, $[\![\ E, (\mathbf{val}\ x_i : \tau)^1\ ]\!]_{\Delta'} = [\![\ E\ ]\!]_{\Delta'}, (\mathbf{val}\ x_i \ : \ [\![\ \tau\ ]\!]^1\ \mathbf{code})^0$. Hence, we obtain $[\![\ E\ ]\!]_\Delta \vdash^0 \ [\![\ (\mathbf{let}\ x_i : \tau \ = \ e)\ s\ ]\!]^1_\Delta \ : \ [\![\ (\mathbf{val}\ x_i : \tau)\ S\ ]\!]^1$.

- Case: S-Type.
  Suppose $l = 0$ (for $l = 1$, we can prove it in the same way). We have the conclusion $E; \Delta \vdash^0 \ (\mathbf{type}\ t_i \ = \ \tau)\ s \ : \ (\mathbf{type}\ t_i \ = \ \tau)\ S$ and the subderivations ${t_i}^0 \notin \mathbf{Dom}(E)$

and E, $(\textbf{type}\ t_i\ =\ \tau)^0; \Delta \vdash^0\ s\ :\ S$. By the induction hypothesis, we have $[\![\ E, (\textbf{type}\ t_i\ =\ \tau)^0\ ]\!]_\Delta \vdash^0\ [\![\ s\ ]\!]^0_\Delta\ :\ [\![\ S\ ]\!]^0$. We have ${t_i}^0 \notin \textbf{Dom}([\![\ E\ ]\!]_\Delta)$. By Rule 5.54 and Rule 5.57, $[\![\ E, (\textbf{type}\ t_i\ =\ \tau)^0\ ]\!]_\Delta = [\![\ E\ ]\!]_\Delta, (\textbf{type}\ t_i\ =\ [\![\ \tau\ ]\!]^0)^0$. By Rule 5.16 and Rule 5.20, $[\![\ (\textbf{type}\ t_i\ =\ \tau)\ s\ ]\!]^0_\Delta = (\textbf{type}\ t_i\ =\ [\![\ \tau\ ]\!]^0)\ [\![\ s\ ]\!]^0_\Delta$. By Rule 5.37 and Rule 5.41, $[\![\ (\textbf{type}\ t_i\ =\ \tau)\ S\ ]\!]^0 = (\textbf{type}\ t_i\ =\ [\![\ \tau\ ]\!]^0)\ [\![\ S\ ]\!]^0$. Hence, we obtain $[\![\ E\ ]\!]_\Delta \vdash^0\ [\![\ (\textbf{type}\ t_i\ =\ \tau)\ s\ ]\!]^0_\Delta\ :\ [\![\ (\textbf{type}\ t_i\ =\ \tau)\ S\ ]\!]^0$.

- Case: S-Mod0.
  We have the conclusion $E; \Delta \vdash^0\ (\textbf{module}\ X_i = m)\ s\ :\ (\textbf{module}\ X_i : M)\ S$, and the subderivations $E; \Delta \vdash^0\ m\ :\ M$, $E; \Delta \vdash^0\ M\ \textbf{wf}$, ${X_i}^0 \notin \textbf{Dom}(E)$, and $E, (\textbf{module}\ X_i : M)^0; \Delta \vdash^0\ s\ :\ S$. By the induction hypothesis, we have $[\![\ E\ ]\!]_\Delta \vdash^0\ [\![\ m\ ]\!]^0_\Delta\ :\ [\![\ M\ ]\!]^0$ and $[\![\ E, (\textbf{module}\ X_i : M)^0\ ]\!]_\Delta \vdash^0\ [\![\ s\ ]\!]^0_\Delta\ :\ [\![\ S\ ]\!]^0$. By Lemma 7, we have $[\![\ E\ ]\!]_\Delta \vdash^0\ [\![\ M\ ]\!]^0\ \textbf{wf}$. We have ${X_i}^0 \notin \textbf{Dom}([\![\ E\ ]\!]_\Delta)$. By Rule 5.16 and Rule 5.21, $[\![\ (\textbf{module}\ X_i = m)\ s\ ]\!]^0_\Delta = (\textbf{module}\ X_i = [\![\ m\ ]\!]^0_\Delta)\ [\![\ s\ ]\!]^0_\Delta$. By Rule 5.37 and Rule 5.42, $[\![\ (\textbf{module}\ X_i : M)\ S\ ]\!]^0 = (\textbf{module}\ X_i : [\![\ M\ ]\!]^0)\ [\![\ S\ ]\!]^0$. By Rule 5.54 and Rule 5.58, $[\![\ E, (\textbf{module}\ X_i : M)^0\ ]\!]_\Delta = [\![\ E\ ]\!]_\Delta, (\textbf{module}\ X_i : [\![\ M\ ]\!]^0)^0$. Hence, we obtain $[\![\ E\ ]\!]_\Delta \vdash^0\ [\![\ (\textbf{module}\ X_i = m)\ s\ ]\!]^0_\Delta\ :\ [\![\ (\textbf{module}\ X_i : M)\ S\ ]\!]^0$.

- Case: S-Mod1.
  We have the conclusion $E; \Delta \vdash^1\ (\textbf{module}\ X_i = m)\ s\ :\ (\textbf{module}\ X_i : M)\ S$, and the subderivations $E; \Delta \vdash^1\ m\ :\ M$, $E; \Delta \vdash^1\ M\ \textbf{wf}$, ${X_i}^1 \notin \textbf{Dom}(E)$, and $E, (\textbf{module}\ X_i : M)^1; \Delta, (\textbf{module}\ X_i : M)^1 \vdash^1\ s\ :\ S$. Let Δ' be $\Delta, (\textbf{module}\ X_i : M)^1$. By the induction hypothesis, we have $[\![\ E\ ]\!]_\Delta \vdash^0\ [\![\ m\ ]\!]^1_\Delta\ :\ [\![\ M\ ]\!]^1$ and $[\![\ E, (\textbf{module}\ X_i : M)^1\ ]\!]_{\Delta'} \vdash^0\ [\![\ s\ ]\!]^1_{\Delta'}\ :\ [\![\ S\ ]\!]^1$. We have ${X_i}^1 \notin \textbf{Dom}([\![\ E\ ]\!]_\Delta)$. By Rule 5.17 and Rule 5.21, $[\![\ (\textbf{module}\ X_i = m)\ s\ ]\!]^1_\Delta = (\textbf{module}\ X_i = [\![\ m\ ]\!]^1_\Delta)\ [\![\ s\ ]\!]^1_{\Delta'}$. By Rule 5.37 and Rule 5.42, $[\![\ (\textbf{module}\ X_i : M)\ S\ ]\!]^1 = (\textbf{module}\ X_i : [\![\ M\ ]\!]^1)\ [\![\ S\ ]\!]^1$. By Rule 5.54 and Rule 5.59, $[\![\ E, (\textbf{module}\ X_i : M)^1\ ]\!]_{\Delta'} = [\![\ E\ ]\!]_{\Delta'}, (\textbf{module}\ X_i : [\![\ M\ ]\!]^1)^0$. Hence, we obtain $[\![\ E\ ]\!]_\Delta \vdash^0\ [\![\ (\textbf{module}\ X_i = m)\ s\ ]\!]^1_\Delta\ :\ [\![\ (\textbf{module}\ X_i : M)\ S\ ]\!]^1$.

Q.E.D.

We have proved that the translation preserves typing. Therefore, we can implement $\lambda^{<M_A>}$ by translating to MetaOCaml. Our implementation and experiments will be described in Chapter 7.

Chapter 6

Proposed Langugae: $\lambda^{<M_G>}$

This section defines the language $\lambda^{<M_G>}$ for generative functors and first-class modules. We first define the syntax and type system of $\lambda^{<M_G>}$, then define a translation to MetaOCaml.

The language $\lambda^{<M_G>}$ is a two-stage programming language and an extension of core MetaOCaml, which consists of simply typed lambda calculus with let expressions, first-class modules, and multi-stage constructors. The design of the language is based on Leroy's module calculus [26], the classic type system $\lambda\circ$ [25], and Watanabe et al.'s calculus [16]. The language in this thesis contains some changes from our previous work [27].

6.1 Syntax

Figure 6.1 shows the syntax for terms. We use metavariables m^l for level-l module expressions, s^l for a sequence of level-l structure components, c^l for level-l structure components, and e^l for level-l core expressions. The level l is either 0 (present stage) or 1 (future stage) because of two-stage language. We also use metavariables x (resp. t, X) for value names (resp. type, module). \$ and **run_module** are constructors introduced by Watanabe et al. If x is bound to code of a first-class module, $\$x.y$ refers to the code of the component y in the module. The **run_module** executes code of a module. $<>$, $\sim$, and **run** are the standard multi-stage constructors in MetaOCaml, while we take a different syntax for code of a module $\langle\!\langle(\textbf{module}\ m^1 : M)\rangle\!\rangle$. We do not allow the escape constructor $\sim$ to be applied to the code of a module. In addition, as the level-1 core expressions e^1 do not include modules, there is no other way to make the code of modules, and our syntax rejects expressions that cannot be translated as described in Section 3.2.2. For the same reason, the level-1 module expressions m^1 do not include an unpacking expression (**val** e^0). Programs P are defined as a sequence of the level-0 core expressions.

Figure 5.2 defines the syntax for types. We use metavariables M for types of module expressions, S for a sequence of types of structure components, C for types of structure components, and σ for types of core expressions. A type of a first-class module is (**module** M). In this work we introduced the type (**module** M) **mcod** to distinguish the code type for modules than that for core expressions (τ **code**) in order to disallow code of functors.

$$
\begin{array}{rl}
\text{Module expressions :} & m^0 ::= X \mid \textbf{struct } s^0 \textbf{ end} \mid (\textbf{val } e^0) \\
& m^1 ::= X \mid \textbf{struct } s^1 \textbf{ end} \\
\text{Structures :} & s^l ::= \epsilon \mid c^l \ s^l \\
\text{Structure components :} & c^l ::= \textbf{let } x : \sigma \ = \ e^l \mid \textbf{type } t \ = \ \sigma \mid \textbf{module } X = m^l \\
\text{Core expressions :} & e^0 ::= x \mid X.x \\
& \quad \mid \textbf{fun } x \ \rightarrow \ e^0 \mid e^0 \ e^0 \mid \textbf{let } x \ = \ e^0 \textbf{ in } e^0 \\
& \quad \mid (\textbf{module } m^0 : M) \\
& \quad \mid < e^1 > \mid \textbf{run } e^0 \\
& \quad \mid \ \langle\!\langle (\textbf{module } m^1 : M) \rangle\!\rangle \ \mid (\textbf{run\_module } e^0 : M) \\
& \quad \mid \$x.x \\
& e^1 ::= x \mid X.x \\
& \quad \mid \textbf{fun } x \ \rightarrow \ e^1 \mid e^1 \ e^1 \mid \textbf{let } x \ = \ e^1 \textbf{ in } e^1 \\
& \quad \mid \sim e^0 \\
\text{Programs :} & P ::= \epsilon \mid e^0 \ P
\end{array}
$$

Figure 6.1: Syntax for terms

$$
\begin{array}{rl}
\text{Module types :} & M ::= \textbf{sig } S \textbf{ end} \\
\text{Signatures :} & S ::= \epsilon \mid C \ S \\
\text{Signature components :} & C ::= \textbf{val } x : \sigma \mid \textbf{type } t \mid \textbf{type } t \ = \ \sigma \mid \textbf{module } X : M \\
\text{Core types :} & \sigma ::= t \mid X.t \\
& \quad \mid \sigma \ \rightarrow \ \sigma \mid (\textbf{module } M) \\
& \quad \mid \tau \textbf{ code} \\
& \quad \mid (\textbf{module } M) \textbf{ mcod} \mid \$x.t \\
& \quad \text{where } \tau \text{ is } \textbf{module}\text{-free}
\end{array}
$$

Figure 6.2: Syntax for types

$$
\begin{array}{l}
E ::= \epsilon \mid C^l, E \\
\Delta ::= \epsilon \mid C^l, \Delta
\end{array}
$$

Figure 6.3: Typing environments

6.2 Type System

6.2.1 Typing Environments

Figure 6.3 shows typing environments of $\lambda^{<M_G>}$. The typing environments E and Δ are both a sequence of signature components that are annotated with a level l, where l is either 0 or 1. Δ is used in translation rather than typing.

6.2.2 Typing Judgements

Typing judgments are also annotated by the level l. For example, $E; \Delta \vdash^l$ means a typing judgment at the level l under the environment E and Δ. At the level l, only the elements annotated with l in the environment E may be dereferenced. Δ is used in translation.

6.2.3 Well-Typedness

We define that a program P is well-typed if $\vdash\ P$ **wt** is derivable. WT-Prog is a rule for well-typedness of programs. The stage level starts from 0.

$\boxed{\vdash\ P\ \mathbf{wt}}$

$$\frac{\phi; \phi \vdash^0\ e_1^0\ :\ \sigma \qquad \cdots \qquad \phi; \phi \vdash^0\ e_n^0\ :\ \sigma}{\vdash\ e_1^0\ \cdots\ e_n^0\ \mathbf{wt}}\ \text{(WT-Prog)}$$

6.2.4 Well-Formedness

The language $\lambda^{<M_G>}$ defines well-formed (**wf**) types in which component names are unique. Most of rules for well-formedness are the same as $\lambda^{<M_A>}$ for applicative. The well-formedness rules are defined as follows. T-Mod is a rule for the type of first-class modules and T-ModCode is a rule for the type of code of first-class modules. T-Code requires the type τ. T-CSP implicitly performs CSP for types. We write $\mathbf{Dom}(E)$ for variables bound in the environment E.

$\boxed{E; \Delta \vdash^l\ M\ \mathbf{wf}}$

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

$\boxed{E; \Delta \vdash^l\ S\ \mathbf{wf}}$

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

$$\frac{E; \Delta \vdash^l\ C\ \mathbf{wf} \qquad E,\ (C)^l; \Delta \vdash^l\ S\ \mathbf{wf}}{E; \Delta \vdash^l\ C\ S\ \mathbf{wf}}\ \text{(WF-SigComponents)}$$

$\boxed{E; \Delta \vdash^l \ C \ \mathbf{wf}}$

$$\frac{E; \Delta \vdash^l \ \sigma \ \mathbf{wf} \qquad x^l \notin \mathbf{Dom}(E)}{E; \Delta \vdash^l \ \mathbf{val} \ x^l : \sigma \ \mathbf{wf}} \ (\text{WF-VAL})$$

$$\frac{t^l \notin \mathbf{Dom}(E)}{E; \Delta \vdash^l \ \mathbf{type} \ t \ \mathbf{wf}} \ (\text{WF-TYPEABS})$$

$$\frac{t^l \notin \mathbf{Dom}(E)}{E; \Delta \vdash^l \ \mathbf{type} \ t \ = \ \sigma \ \mathbf{wf}} \ (\text{WF-TYPE})$$

$$\frac{E; \Delta \vdash^l \ M \ \mathbf{wf} \qquad X^l \notin \mathbf{Dom}(E)}{E; \Delta \vdash^l \ \mathbf{module} \ X : M \ \mathbf{wf}} \ (\text{WF-MOD})$$

$\boxed{E; \Delta \vdash^l \ \sigma \ \mathbf{wf}}$

$$\frac{(\mathbf{type} \ t \ = \ \sigma)^l \in E}{E; \Delta \vdash^l \ t \ \mathbf{wf}} \ (\text{T-VAR})$$

$$\frac{(\mathbf{type} \ t)^l \in E}{E; \Delta \vdash^l \ t \ \mathbf{wf}} \ (\text{T-VARABS})$$

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

$$\frac{(\mathbf{module} \ X : \mathbf{sig} \ S \ \mathbf{end})^l \in E \qquad (\mathbf{type} \ t) \in S}{E; \Delta \vdash^l \ X.t \ \mathbf{wf}} \ (\text{T-DOTABS})$$

$$\frac{(\mathbf{val} \ x : (\mathbf{module \ sig} \ S \ \mathbf{end}) \ \mathbf{mcod})^0 \in E \qquad (\mathbf{type} \ t \ = \ \tau) \in S}{E; \Delta \vdash^0 \ \$x.t \ \mathbf{wf}} \ (\text{T-DOTCODE})$$

$$\frac{(\mathbf{val} \ x : (\mathbf{module \ sig} \ S \ \mathbf{end}) \ \mathbf{mcod})^0 \in E \qquad (\mathbf{type} \ t) \in S}{E; \Delta \vdash^0 \ \$x.t \ \mathbf{wf}} \ (\text{T-DOTCODEABS})$$

$$\frac{E; \Delta \vdash^l \ \sigma_1 \ \mathbf{wf} \qquad E; \Delta \vdash^l \ \sigma_2 \ \mathbf{wf}}{E; \Delta \vdash^l \ \sigma_1 \ \rightarrow \ \sigma_2 \ \mathbf{wf}} \ (\text{T-ARR})$$

$$\frac{E; \Delta \vdash^0 \ \tau \ \mathbf{wf}}{E; \Delta \vdash^0 \ \tau \ \mathbf{code \ wf}} \ (\text{T-CODE})$$

$$\frac{E; \Delta \vdash^0 \ \sigma \ \mathbf{wf}}{E; \Delta \vdash^1 \ \sigma \ \mathbf{wf}} \ (\text{T-CSP})$$

$$\frac{E; \Delta \vdash^l \ M \ \mathbf{wf}}{E; \Delta \vdash^l \ (\mathbf{module} \ M) \ \mathbf{wf}} \ (\text{T-MOD})$$

$$\frac{E; \Delta \vdash^0 \ (\mathbf{module} \ M) \ \mathbf{wf}}{E; \Delta \vdash^0 \ (\mathbf{module} \ M) \ \mathbf{mcod \ wf}} \ (\text{T-MODCODE})$$

6.2.5 Typing Rules

This section defines typing rules of the language $\lambda^{<M_G>}$. M-VAL is a typing rule for unpacking first-class modules, which is restricted at level 0 due to avoiding the problem shown in Section 3.2.2.

$\boxed{E; \Delta \vdash^l \ m \ : \ M}$

$$\frac{(\textbf{module}\ X : M)^l \in E}{E; \Delta \vdash^l \ X \ : \ M}\ (\text{M-VAR})$$

$$\frac{E; \Delta \vdash^l \ s^l \ : \ S}{E; \Delta \vdash^l \ \textbf{struct}\ s^l\ \textbf{end} \ : \ \textbf{sig}\ S\ \textbf{end}}\ (\text{M-STR})$$

$$\frac{E; \Delta \vdash^0 \ e^0 \ : \ (\textbf{module}\ M)}{E; \Delta \vdash^0 \ (\textbf{val}\ e^0) \ : \ M}\ (\text{M-VAL})$$

The typing rules for structure components are defined below. In S-LET1 and S-MOD1, the environment Δ is updated to prevent dangling references.

$\boxed{E; \Delta \vdash^l \ s \ : \ S}$

$$\frac{}{E; \Delta \vdash^l \ \epsilon \ : \ \epsilon}\ (\text{S-EMPTY})$$

$$\frac{E; \Delta \vdash^0 \ e^0 \ : \ \sigma \qquad x^0 \notin \textbf{Dom}(E) \qquad E,\ (\textbf{val}\ x : \sigma)^0; \Delta \vdash^0 \ s^0 \ : \ S}{E; \Delta \vdash^0 \ (\textbf{let}\ x : \sigma \ = \ e^0)\ s^0 \ : \ (\textbf{val}\ x : \sigma)\ S}\ (\text{S-LET0})$$

$$\frac{E; \Delta \vdash^1 \ e^1 \ : \ \sigma \qquad x^1 \notin \textbf{Dom}(E) \qquad E, (\textbf{val}\ x : \sigma)^1; \Delta, (\textbf{val}\ x : \sigma)^1 \vdash^1 \ s^1 \ : \ S}{E; \Delta \vdash^1 \ (\textbf{let}\ x : \sigma \ = \ e^1)\ s^1 \ : \ (\textbf{val}\ x : \sigma)\ S}\ (\text{S-LET1})$$

$$\frac{t^l \notin \textbf{Dom}(E) \qquad E,\ (\textbf{type}\ t \ = \ \sigma)^l; \Delta \vdash^l \ s^l \ : \ S}{E; \Delta \vdash^l \ (\textbf{type}\ t \ = \ \sigma)\ s^l \ : \ (\textbf{type}\ t \ = \ \sigma)\ S}\ (\text{S-TYPE})$$

$$\frac{\begin{array}{c} E; \Delta \vdash^0 \ m^0 \ : \ M \qquad E; \Delta \vdash^0 \ M\ \textbf{wf} \qquad X^0 \notin \textbf{Dom}(E) \\ E,\ (\textbf{module}\ X : M)^0; \Delta \vdash^0 \ s^0 \ : \ S \end{array}}{E; \Delta \vdash^0 \ (\textbf{module}\ X = m^0)\ s^0 \ : \ (\textbf{module}\ X : M)\ S}\ (\text{S-MOD0})$$

$$\frac{\begin{array}{c} E; \Delta \vdash^1 \ m^1 \ : \ M \qquad E; \Delta \vdash^1 \ M\ \textbf{wf} \qquad X^1 \notin \textbf{Dom}(E) \\ E, (\textbf{module}\ X : M)^1; \Delta, (\textbf{module}\ X : M)^1 \vdash^1 \ s^1 \ : \ S \end{array}}{E; \Delta \vdash^1 \ (\textbf{module}\ X = m^1)\ s^1 \ : \ (\textbf{module}\ X : M)\ S}\ (\text{S-MOD1})$$

The typing rules are carefully designed to distinguish between code of modules and ordinary code such as $< e^1 >$. To prevent a module enclosed with ordinary brackets, the typing rules E-Code, E-Esc, and E-Run require that expressions have the type τ, which is a module-free type. For example, (**module** M) is not the type τ. Only the expression $\langle\!\langle$(**module** m^1 : M)$\rangle\!\rangle$ is allowed to construct code of modules, which has the type (**module** M) **mcod**. For generative semantics, each abstract types has a fresh type. The substitutions in E-Dot and E-DotCode avoid free occurrences of type variables. For example, let x be $\langle\!\langle$(**module struct type** t **let** $y : t \ = \ 1$ **end** : S)$\rangle\!\rangle$, the type of $\$x.y$ becomes $\$x.t$ instead of t.

$\boxed{E; \Delta \vdash^l \ e \ : \ \sigma}$

$$\frac{(\textbf{val}\ x : \sigma)^l \in E}{E; \Delta \vdash^l \ x \ : \ \sigma}\ \text{(E-Var)}$$

$$\frac{(\textbf{module}\ X : \textbf{sig}\ S_1\ (\textbf{val}\ x : \sigma)\ S_2\ \textbf{end})^l \in E}{E; \Delta \vdash^l \ X.x \ : \ \sigma[z \leftarrow X.z \mid z \in \textbf{Dom}(S_1)]}\ \text{(E-Dot)}$$

$$\frac{(\textbf{val}\ x : (\textbf{module}\ (\textbf{sig}\ S_1\ (\textbf{val}\ y : \sigma)\ S_2\ \textbf{end}))\ \textbf{mcod})^0 \in E}{E; \Delta \vdash^0 \ \$x.y \ : \ \sigma[z \leftarrow \$x.z \mid z \in \textbf{Dom}(S_1)]\ \textbf{code}}\ \text{(E-DotCode)}$$

$$\frac{E; \Delta \vdash^l \ \sigma_1\ \textbf{wf} \qquad E,\ (\textbf{val}\ x : \sigma_1)^l; \Delta \vdash^l \ e^l \ : \ \sigma_2}{E; \Delta \vdash^l \ \textbf{fun}\ x \ \rightarrow \ e^l \ : \ \sigma_1 \ \rightarrow \ \sigma_2}\ \text{(E-Fun)}$$

$$\frac{E; \Delta \vdash^l \ e_1^l \ : \ \sigma_1 \ \rightarrow \ \sigma_2 \qquad E; \Delta \vdash^l \ e_2^l \ : \ \sigma_1}{E; \Delta \vdash^l \ e_1^l\ e_2^l \ : \ \sigma_2}\ \text{(E-App)}$$

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

$$\frac{E; \Delta \vdash^0 \ M\ \textbf{wf} \qquad E; \Delta \vdash^0 \ m^0 \ : \ M}{E; \Delta \vdash^0 \ (\textbf{module}\ m^0 : M) \ : \ (\textbf{module}\ M)}\ \text{(E-Mod)}$$

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

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

$$\frac{E; \Delta \vdash^0 \ e^0 \ : \ \tau\ \textbf{code}}{E; \Delta \vdash^0 \ \textbf{run}\ e^0 \ : \ \tau}\ \text{(E-Run)}$$

$$\frac{E; \Delta \vdash^1 \ M\ \textbf{wf} \qquad E; \Delta \vdash^1 \ m^1 \ : \ M}{E; \Delta \vdash^0 \ \langle\!\langle(\textbf{module}\ m^1 : M)\rangle\!\rangle \ : \ (\textbf{module}\ M)\ \textbf{mcod}}\ \text{(E-ModCode)}$$

$$\frac{E; \Delta \vdash^0 \ e^0 \ : \ (\textbf{module}\ M)\ \textbf{mcod}}{E; \Delta \vdash^0 \ (\textbf{run\_module}\ e^0 : M) \ : \ (\textbf{module}\ M)}\ \text{(E-Runmod)}$$

6.3 Translation to plain MetaOCaml

In this section, we define a translation from $\lambda^{<M_G>}$ to MetaOCaml and explain the key part of the translation. As with the language $\lambda^{<M_A>}$ for applicative, we write the translation as $[\![\cdot]\!]^l_\Delta$, where l is the level and Δ is the environment. For convenience, we slightly extend the unpacking constructor in MetaOCaml so that $(\mathbf{val}\ e^0).x$ is a valid syntax, but it can be easily resolved by introducing the declaration $\mathbf{module}\ X = (\mathbf{val}\ e^0)$, and relacing it to $X.x$ for a fresh name X.

We define the translation rules for module expressions by Rule 6.1-6.3. The translation for a sequence of structure components is defined by Rule 6.4-6.6, where Δ' is the environment updated by the typing rules S-LET1 and S-MOD1. The translation for structure components is defined by Rule 6.7-6.10. To avoid duplicate code, Rule 6.8 inserts genlet in front of the code.

$\boxed{[\![\ m\ ]\!]^l_\Delta}$

$$[\![\ X\ ]\!]^l_\Delta = X \tag{6.1}$$
$$[\![\ \mathbf{struct}\ s^l\ \mathbf{end}\ ]\!]^l_\Delta = \mathbf{struct}\ [\![\ s^l\ ]\!]^l_\Delta\ \mathbf{end} \tag{6.2}$$
$$[\![\ (\mathbf{val}\ e^0)\ ]\!]^0_\Delta = (\mathbf{val}\ [\![\ e^0\ ]\!]^0_\Delta) \tag{6.3}$$

$\boxed{[\![\ s\ ]\!]^l_\Delta}$

$$[\![\ \epsilon\ ]\!]^l_\Delta = \epsilon \tag{6.4}$$
$$[\![\ c^0\ s^0\ ]\!]^0_\Delta = [\![\ c^0\ ]\!]^0_\Delta\ [\![\ s^0\ ]\!]^0_\Delta \tag{6.5}$$
$$[\![\ c^1\ s^1\ ]\!]^1_\Delta = [\![\ c^1\ ]\!]^1_\Delta\ [\![\ s^1\ ]\!]^1_{\Delta'} \tag{6.6}$$

$\boxed{[\![\ c\ ]\!]^l_\Delta}$

$$[\![\ \mathbf{let}\ x : \sigma\ =\ e^0\ ]\!]^0_\Delta = \mathbf{let}\ x : [\![\ \sigma\ ]\!]^0\ =\ [\![\ e^0\ ]\!]^0_\Delta \tag{6.7}$$
$$[\![\ \mathbf{let}\ x : \sigma\ =\ e^1\ ]\!]^1_\Delta = \mathbf{let}\ x : [\![\ \sigma\ ]\!]^1\ \mathbf{code}\ =\ \mathbf{genlet}\ <[\![\ e^1\ ]\!]^1_\Delta> \tag{6.8}$$
$$[\![\ \mathbf{type}\ t\ =\ \sigma\ ]\!]^l_\Delta = \mathbf{type}\ t\ =\ [\![\ \sigma\ ]\!]^l \tag{6.9}$$
$$[\![\ \mathbf{module}\ X = m^l\ ]\!]^l_\Delta = \mathbf{module}\ X = [\![\ m^l\ ]\!]^l_\Delta \tag{6.10}$$

The translation for core expressions is defined by Rule 6.11-6.28. The environment Δ is used in Rule 6.12 and Rule 6.14 to adjust a level of variables. By Rule 6.23, brackets outside a module are removed, and the module is translated at level 1. The translation for **run_module** is defined by Rule 6.24-6.28, which depend on the signature of the target module. The expression $S /\!\!/ X$ applies the **run**-primitive to each value component in the module X of the signature S.

$\boxed{[\![\ e\ ]\!]^l_\Delta}$

$$[\![\ x\ ]\!]^0_\Delta = x \tag{6.11}$$

$$[\![\ x\ ]\!]^1_\Delta = \begin{cases} \sim x & (x \in \mathrm{Dom}(\Delta)) \\ x & (otherwise) \end{cases} \tag{6.12}$$

$$[\![\ X.x\ ]\!]^0_\Delta = X.x \tag{6.13}$$

$$[\![\ X.x\ ]\!]^1_\Delta = \begin{cases} \sim (X.x) & (X \in \mathrm{Dom}(\Delta)) \\ X.x & (otherwise) \end{cases} \tag{6.14}$$

$$[\![\ \mathbf{fun}\ x\ \rightarrow\ e^l\ ]\!]^l_\Delta = \mathbf{fun}\ x\ \rightarrow\ [\![\ e^l\ ]\!]^l_\Delta \tag{6.15}$$

$$[\![\ e^l_1\ e^l_2\ ]\!]^l_\Delta = [\![\ e^l_1\ ]\!]^l_\Delta\ [\![\ e^l_2\ ]\!]^l_\Delta \tag{6.16}$$

$$[\![\ \mathbf{let}\ x\ =\ e^l_1\ \mathbf{in}\ e^l_2\ ]\!]^l_\Delta = \mathbf{let}\ x\ =\ [\![\ e^l_1\ ]\!]^l_\Delta\ \mathbf{in}\ [\![\ e^l_2\ ]\!]^l_\Delta \tag{6.17}$$

$$[\![\ < e^1 >\ ]\!]^0_\Delta = < [\![\ e^1\ ]\!]^1_\Delta > \tag{6.18}$$

$$[\![\ \sim e^0\ ]\!]^1_\Delta = \sim [\![\ e^0\ ]\!]^0_\Delta \tag{6.19}$$

$$[\![\ \mathbf{run}\ e^0\ ]\!]^0_\Delta = \mathbf{run}\ [\![\ e^0\ ]\!]^0_\Delta \tag{6.20}$$

$$[\![\ \$x_1.x_2\ ]\!]^0_\Delta = (\mathbf{val}\ x_1)\,.\,x_2 \tag{6.21}$$

$$[\![\ (\mathbf{module}\ m^0 : M)\ ]\!]^0_\Delta = (\mathbf{module}\ [\![\ m^0\ ]\!]^0_\Delta : [\![\ M\ ]\!]^0) \tag{6.22}$$

$$[\![\ \langle\!\langle(\mathbf{module}\ m^1 : M)\rangle\!\rangle\ ]\!]^0_\Delta = (\mathbf{module}\ [\![\ m^1\ ]\!]^1_\Delta : [\![\ M\ ]\!]^1) \tag{6.23}$$

$$\begin{aligned} [\![\ (\mathbf{run\_module}\ e^0 : \mathbf{sig}\ S\ \mathbf{end})\ ]\!]^0_\Delta = (&\mathbf{module\ struct} \\ &\mathbf{module}\ X = (\mathbf{val}\ [\![\ e^0\ ]\!]^0_\Delta) \\ &S /\!\!/ X \\ &\mathbf{end} : \mathbf{sig}\ [\![\ S\ ]\!]^0\ \mathbf{end}) \end{aligned} \tag{6.24}$$

$\boxed{S /\!\!/ X}$

$$\epsilon /\!\!/ X = \epsilon \tag{6.25}$$

$$((\mathbf{val}\ x : \sigma)\ S) /\!\!/ X = (\mathbf{let}\ x : \sigma\ =\ \mathbf{run}\ X.x)\ S /\!\!/ X \tag{6.26}$$

$$((\mathbf{type}\ t\ =\ \sigma)\ S) /\!\!/ X = (\mathbf{type}\ t\ =\ X.t)\ S /\!\!/ X \tag{6.27}$$

$$\begin{aligned} ((\mathbf{module}\ X' : M)\ S) /\!\!/ X = (&\mathbf{module}\ X' = \\ &(\mathbf{val}\ [\![\ (\mathbf{run\_module}\ (\mathbf{module}\ X.X' : M) : M)\ ]\!]^0_\epsilon)) \\ &S /\!\!/ X \end{aligned} \tag{6.28}$$

The translation for programs removes **prog** and **end**, and the sequence of core expressions is translated at level 0.

$\boxed{[\![\ P\ ]\!]^0_\Delta}$

$$[\![\ \mathbf{prog}\ e^0_1 \cdots e^0_n\ \mathbf{end}\ ]\!]^0 = [\![\ e^0_1\ ]\!]^0_\Delta \cdots [\![\ e^0_n\ ]\!]^0_\Delta \tag{6.29}$$

Then, we define the translation for module types by Rule 6.30, the translation for a sequence of signature components by Rule 6.31-6.32, the translation for signature components by Rule 6.33-6.37, and the translation for core types by Rule 6.38-6.44. We write $[\![\cdot]\!]^l$ for these translations without the environment Δ. Our extensions are eliminated by the translation.

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

$$[\![\textbf{sig } S \textbf{ end}]\!]^l = \textbf{sig } [\![S]\!]^l \textbf{ end} \tag{6.30}$$

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

$$[\![\epsilon]\!]^l = \epsilon \tag{6.31}$$
$$[\![C\ S]\!]^l = [\![C]\!]^l\ [\![S]\!]^l \tag{6.32}$$

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

$$[\![\textbf{val } x : \sigma]\!]^0 = \textbf{val } x : [\![\sigma]\!]^0 \tag{6.33}$$
$$[\![\textbf{val } x : \sigma]\!]^1 = \textbf{val } x : [\![\sigma]\!]^1 \textbf{ code} \tag{6.34}$$
$$[\![\textbf{type } t]\!]^l = \textbf{type } t \tag{6.35}$$
$$[\![\textbf{type } t\ =\ \sigma]\!]^l = \textbf{type } t\ =\ [\![\sigma]\!]^l \tag{6.36}$$
$$[\![\textbf{module } X : M]\!]^l = \textbf{module } X : [\![M]\!]^l \tag{6.37}$$

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

$$[\![t]\!]^l = t \tag{6.38}$$
$$[\![X.t]\!]^l = X.t \tag{6.39}$$
$$[\![\$x.t]\!]^l = (\textbf{val } x)\,.\,t \tag{6.40}$$
$$[\![\sigma_1\ \rightarrow\ \sigma_2]\!]^l = [\![\sigma_1]\!]^l\ \rightarrow\ [\![\sigma_2]\!]^l \tag{6.41}$$
$$[\![\tau \textbf{ code}]\!]^0 = [\![\tau]\!]^0 \textbf{ code} \tag{6.42}$$
$$[\![(\textbf{module } M)]\!]^l = (\textbf{module } [\![M]\!]^l) \tag{6.43}$$
$$[\![(\textbf{module } M) \textbf{ mcod}]\!]^0 = (\textbf{module } [\![M]\!]^1) \tag{6.44}$$

The translation for the typing environment E is defined by Rule 6.45-6.51, where we use the notation $[\![\cdot]\!]_\Delta$. Since each element of an environment carries its level, we do not need to annotate the translation with a level. The translation changes the level of bindings to level-1 value components and level-1 module components to level 0. Note that bindings inside ordinary brackets remain at level 1.

$\boxed{[\![E]\!]_\Delta}$

$$[\![\epsilon]\!]_\Delta = \epsilon \tag{6.45}$$

$$[\![C^l;\ E]\!]_\Delta = [\![C^l]\!]_\Delta;\ [\![E]\!]_\Delta \tag{6.46}$$

$\boxed{[\![C^j]\!]_\Delta}$

$$[\![(\mathbf{val}\ x : \sigma)^0]\!]_\Delta = (\mathbf{val}\ x : [\![\sigma]\!]^0)^0 \tag{6.47}$$

$$[\![(\mathbf{val}\ x : \sigma)^1]\!]_\Delta = \begin{cases} (\mathbf{val}\ x : [\![\sigma]\!]^1\ \mathbf{code})^0 & (x \in \mathrm{Dom}(\Delta)) \\ (\mathbf{val}\ x : [\![\sigma]\!]^1)^1 & (otherwise) \end{cases} \tag{6.48}$$

$$[\![(\mathbf{type}\ t\ =\ \sigma)^l]\!]_\Delta = (\mathbf{type}\ t\ =\ [\![\sigma]\!]^l)^l \tag{6.49}$$

$$[\![(\mathbf{module}\ X : M)^0]\!]_\Delta = (\mathbf{module}\ X : [\![M]\!]^0)^0 \tag{6.50}$$

$$[\![(\mathbf{module}\ X : M)^1]\!]_\Delta = (\mathbf{module}\ X : [\![M]\!]^1)^0 \tag{6.51}$$

It can be expected that the following form of theorem holds for the translation defined in this section; if $E; \Delta \vdash^l e^l\ :\ \sigma$, then $[\![E]\!]_\Delta \vdash^l [\![e^l]\!]^l_\Delta\ :\ [\![\sigma]\!]^l$. The proof of this theorem is left for future work.

Chapter 7

Performance Evaluation

We have implemented our languages through the translations, 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. perform a domain-specific optimization for arithmetic expressions such as $0 + n \rightarrow n$ using the tagless-final embedding [14]. Figure 7.1 shows the core part of the benchmark written in $\lambda^{<M_A>}$, where the language includes extensions such as the base type `int`, conditional expressions, and primitives for arithmetic operations. The tagless-final style uses module types to embed syntax and typing rules of the object language. The module type `S` specifies the 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 functor `SuppressAddMulZero` is a program translator in such an object language. It is given code of a module with signature `S` and returns code of a module after performing the optimization. By applying it repeatedly, a fully optimized module can be obtained. In this chapter, the depth refers to the number of repeated functor applications. For the complete implementation, see our repositories [28, 29].

The code-explosion problem shows up if we use Watanabe et al.'s translation for the above program. The functor `SuppressAddMulZero`, given a module `M`, splices the components of `M` into (the code of) a new module. For example, in the `int` component, the `M.int` code is spliced twice. Thus, as the depth of functor calls increases, the size of the generated code increases exponentially.

We use the following programs for experiments.

- A MetaOCaml program translated from the benchmark program written in $\lambda^{<M_A>}$.
- A MetaOCaml program translated from the benchmark program written in $\lambda^{<M_G>}$.
- A MetaOCaml program translated from the benchmark program written in $\lambda^{<M>}$.
- A naive OCaml program that expresses the benchmark without code generation.

```
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

module SuppressAddMulZero = functor(M : S mcod) ->
  .< (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) >.
```

Figure 7.1: Core Part of the Benchmark Written in $\lambda^{<M_A>}$

For these programs, we measure the time for code generation and compilation, the execution time of generated code, the size of generated code, and the memory usage during program execution. The measurement result is the average of 10 trials. We conduct these experiments on Ubuntu 18.04 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. The size of code of a generated module is defined as the sum of string lengths for each code of component in the module.

Figure 7.2 shows the time for code generation and compilation, where the horizontal axis is the number of functor applications (depth) and the vertical axis is the time for code generation on a logarithmic scale. The time of Watanabe et al.'s program increases exponentially, and the experiment was only performed up to depth 15. On the other hand, ours have a gentle slope. In this benchmark, our two programs actually generate the same code

for value components because the value component contains only values and not modules.

Figure 7.3 shows the size of generated code where the vertical axis is on a logarithmic scale. As with the time for code generation, the size of Watanabe et al.'s program increases exponentially, while ours do linearly.

Figure 7.4 shows the execution time (excluding the time for code generation). The result shows that the generated modules run faster than the naive one. Our programs are about 30% faster than Watanabe et al.'s (Figure 7.5). Our non-duplicating code reduces the number of steps in program execution and encourages optimization by the compiler.

Figure 7.6 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. On the other hand, the memory usage of ours has a gentle slope. 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 create at most 100 modules.

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 [11]. At depth 10, the execution time is 0.14 seconds for the naive program (without code generation), while it is only 0.016 seconds for our programs. Even if the time for code generation is added, our program is more efficient. 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.

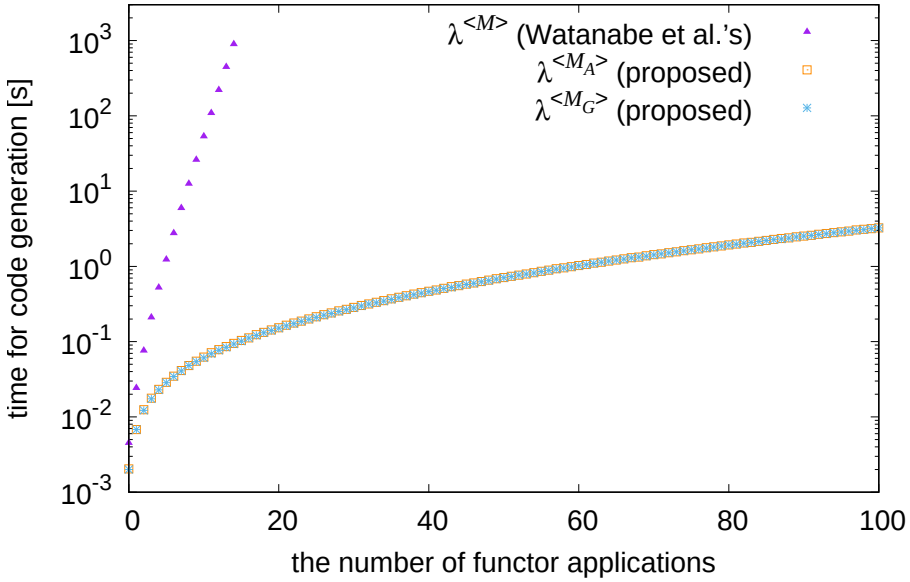

Figure 7.2: Time for Code Generation

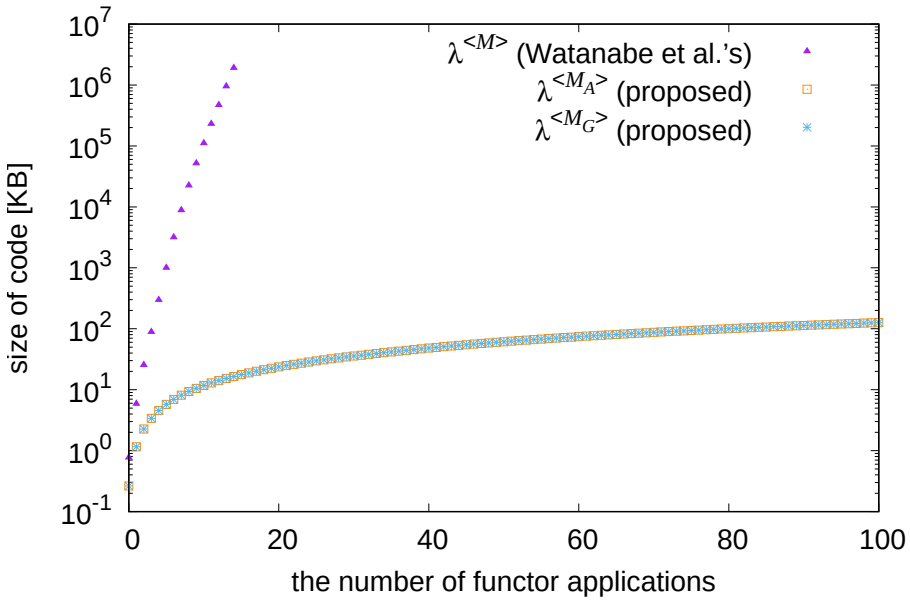

Figure 7.3: Size of Generated Code

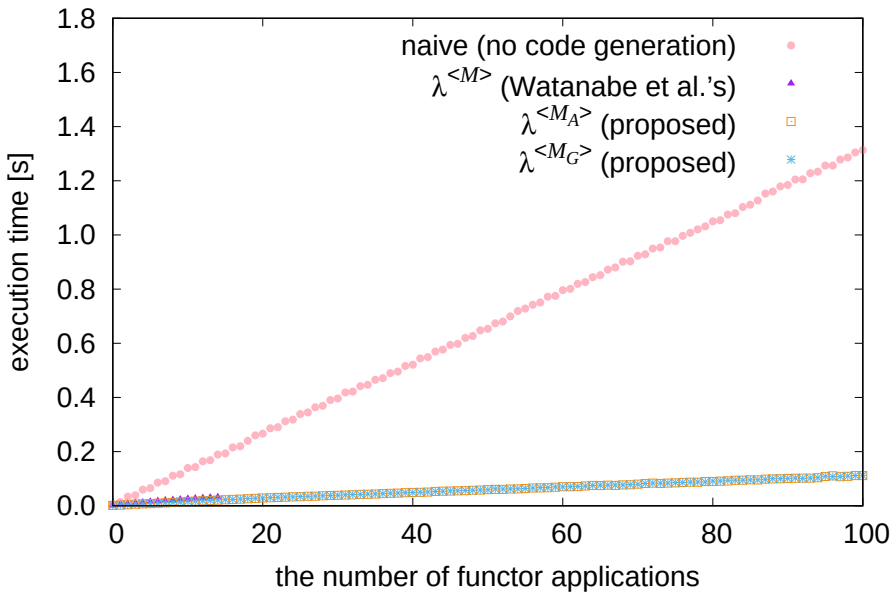

Figure 7.4: Execution Time for Generated Code

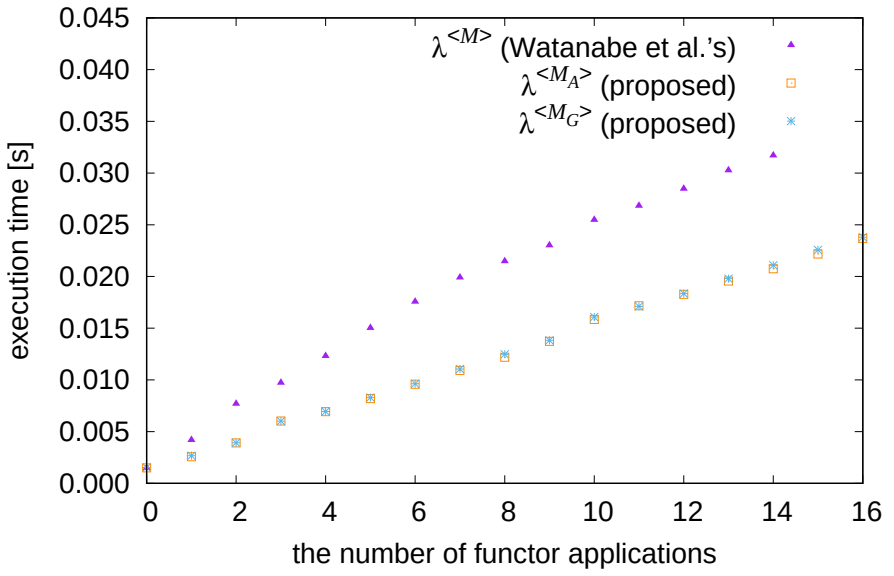

Figure 7.5: Execution Time for Generated Code (zoomed)

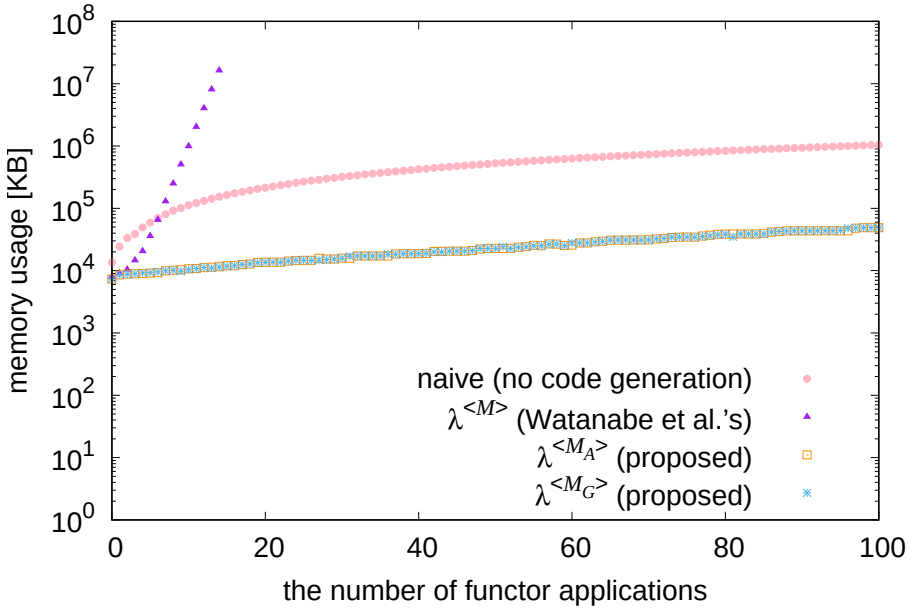

Figure 7.6: Memory Usage

Chapter 8

Discussion

8.1 Code of Functors

Our languages allow code of modules, but disallow code of functors. This restriction (slightly) complicated the syntaxes of our languages. However, 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 the language $\lambda^{<M_G>}$. 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 [30]. 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 thesis, 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 in the sense that we cannot manipulate code inside brackets, hence the code of functors may not be further optimized.

8.2 Remaining Duplicated Code

Although our translations eliminate most code duplication, they still allow duplicated code to be generated at the top level. Namely, each component has no duplicated code, but duplicated code may exist at the top-level components. We think that this is not a serious problem, as such duplication appears only at the top level.

8.3 Possible Scope Extrusion of Local Module References

Parreaux and Shaikhha [31] cite our paper [27] and are concerned that genlet may extrude local module references. Figure 8.1 is an illustrative example that we inferred based on their

```
module type S = sig
  val v: int code
end

module X: S = struct
  module Y = struct
    let f = fun x -> x
  end
  let v = genlet < Y.f 1 >
end

let u = < ~(X.v) > ;;
(* val u : int code =
    <let lv_1 = (* CSP Y.f *) 1 in lv_1> *)
Runcode.run u ;;
(* - : int = 1 *)
```

Figure 8.1: Extruding Local Module References

claims. In this example, the nested module `Y` is concealed by sealing with the signature `S`, while a reference to `Y` is embedded into code of the value component `v` and genlet is applied to its code. Since the component `v` is exposed by the signature `S`, it can be accessed from outside the module `X`, as in the code bound to the variable `u`. The problem here is that the code of `u` contains the reference to the module `Y` that is out of scope. This seems to be the scope extrusion problem. However, it is due to the combination of modules and CSP rather than genlet. The generated code can be executed, but it is difficult for us to argue that MetaOCaml is sound. To avoid this problem, our languages do not allow CSP for arbitrary values. In addition, since all value components in code of a module are translated to ones of type code, we do not encounter the pattern in Figure 8.1.

Let us hint at two possible solutions for it. One is to limit the upper bound for the scope of genlet; if code contains local module references, then genlet should not perform let-insertion beyond the scope of its module. Although our translation relies on genlet crossing module boundaries, this limitation does not affect ours. Because our translation inlines code of value components into another code, no module reference is left in the result code. The other is to limit CSP to basic values or external module references only. Namely, it does not allow CSP for local module references. Addressing this problem is future work.

8.4 Preserving Semantics of Termination

The genlet primitive changes the semantics of program termination. An illustrative example is:

```
# <if true then 0 else ~(genlet <1+2>)>
- : int code = <let lv_1 = 1 + 2 in if true then 0 else
   lv_1>
```

When the first line is evaluated, the second line is generated in which the expression `1+2` is

inserted into the top of the `if` statement. If we replace `1+2` by a non-terminating expression, the semantics of the code is changed by inserting genlet in the original code. Thus, inserting genlet is problematic in general.

Our translations use genlet, but we argue that this problem does not occur. The first reason is that genlet is not a primitive of our source languages. The second reason is that OCaml evaluates the right-hand side of each component when evaluating a module. That is, if a module has components that do not terminate, the computation of the entire module does not terminate, either. Therefore, in our translations, the semantics of termination is preserved even if genlet let-inserts components that do not terminate, as in the above example.

Chapter 9

Related Work

In this chapter, we mention 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 class generation and research on using genlet.

Macros [32] 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 [33] 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 thesis might be eliminated by Flambda, too. Also, MLton [34] 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 a programmer is given full control as to how and what code is generated.

Functoria [11] 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.

Squid [4] is a multi-stage programming framework for Scala, and guarantees that generated code is well-typed and well-scoped. The latest work in Squid is generating classes, which is presented by Parreaux and Shaikhha [31]. They 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.

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, Yallop and Kiselyov [35] use genlet for generating mutually-recursive definitions. Their study influenced MetaOCaml, and the latest MetaOCaml 4.11 now provides primitives to facilitate let-rec insertion. The primitives can mark the scope of the let rec definition that will be generated. In our language, recursive value components are simply bound with let expression and inserted using genlet [1], but controlling destinations of genlet may be useful to tackle challenges left open, such as avoiding duplicate code at the top level and generating recursive modules (mutually-recursive functors).

[1] This thesis does not define a translation rule for the let-rec value component, but our implementations support it.

Chapter 10

Conclusion

In this thesis, we have studied multi-stage programming languages for generating and manipulating code of modules. We have analyzed the code-explosion problem in the previous study and proposed an approach that uses the MetaOCaml's genlet primitive to share code among modules. We have designed two languages that allow generating modules in two different styles, which are translated based on the approach. We have defined the languages and the translations, and proved that one of the translations preserves typing. We have implemented program translators based on our translations and conducted a few experiments against a microbenchmark. The result shown that our translations give an space- and time-efficient code for applications which need repeated applications of functors to modules.

Our contributions in this paper are summarized as follows. First, we 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 Watanabe et al.'s language to allow code generation including nested modules and abstract types, and gave complete definitions including **run_module**. Also, we have given the proof of type preservation of the translation. Fourth, we have introduced a language with second-class modules and applicative functors, which is essential for staging existing OCaml programs.

In future work, we plan to extend our source languages to more realistic ones, such as side effects, polymorphism, sharing constraints, and mutually-recursive modules. We also plan to develop practical applications in large using our languages. We think that MirageOS is one of such applications where approximately 10-times nested functor applications are used in practice. Investigating theoretical foundation of genlet is also an interesting future work.

Acknowledgements

I am grateful to my supervisor Yukiyoshi Kameyama for valuable discussions and feedbacks. I would like to thank Hiroshi Unno and other members of Programming Logic Group for helpful comments and supports. I would also like to thank Oleg Kiselyov for suggestions and constructive comments. Finally, I would also like to express my gratitude to my family for their moral support and warm encouragements.

Bibliography

[1] 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. 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*, pages 238–261, 2015.

[2] Oleg Kiselyov. 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*, pages 86–102, 2014.

[3] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *Commun. ACM*, 55(6):121–130, 2012.

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

[5] Tim Sheard and Simon L. Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, page 1–16, New York, NY, USA, 2002. Association for Computing Machinery.

[6] Gregory Neverov and Paul Roe. Metaphor: A multi-stage, object-oriented programming language. In Gabor Karsai and Eelco Visser, editors, *Generative Programming and Component Engineering: Third International Conference, GPCE 2004, Vancouver, Canada, October 24-28, 2004. Proceedings*, volume 3286 of *Lecture Notes in Computer Science*, pages 168–185. Springer, 2004.

[7] Oleg Kiselyov. Reconciling abstraction with high performance: A MetaOCaml approach. *Foundations and Trends in Programming Languages*, 5(1):1–101, 2018.

[8] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. Stream fusion, to completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, page 285–299, New York, NY, USA, 2017. Association for Computing Machinery.

[9] Tiark Rompf and Nada Amin. Functional pearl: A sql to c compiler in 500 lines of code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, page 2–9, New York, NY, USA, 2015. Association for Computing Machinery.

[10] Oleg Kiselyov, Kedar N. Swadi, and Walid Taha. A methodology for generating verified combinatorial circuits. In *EMSOFT 2004, September 27-29, 2004, Pisa, Italy, Fourth ACM International Conference On Embedded Software, Proceedings*, pages 249–258, 2004.

[11] Gabriel Radanne, Thomas Gazagnaire, Anil Madhavapeddy, Jeremy Yallop, Richard Mortier, Hannes Mehnert, Mindy Preston, and David J. Scott. Programming unikernels in the large via functor driven development. *CoRR*, abs/1905.02529, 2019.

[12] Jeremy Yallop. 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*, pages 85–96, 2016.

[13] Kenichi Suzuki, Oleg Kiselyov, and Yukiyoshi Kameyama. 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*, pages 37–48, 2016.

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

[15] Jun Inoue, Oleg Kiselyov, and Yukiyoshi Kameyama. 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*, pages 103–108, 2016.

[16] Takahisa Watanabe and Yukiyoshi Kameyama. 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*, pages 60–66, 2018.

[17] Derek Dreyer. *Understanding and Evolving the ML Module System*. PhD thesis, USA, 2005. AAI3166274.

[18] Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*, pages 30–50, 2003.

[19] Olivier Danvy. Pragmatic aspects of type-directed partial evaluation. Technical report, BRICS Report Series RS-96-15, May 1996.

[20] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990.*, pages 151–160, 1990.

[21] Oleg Kiselyov. Delimited control in ocaml, abstractly and concretely. *Theor. Comput. Sci.*, 435:56–76, 2012.

[22] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. Shifting the stage - staging with delimited control. *J. Funct. Program.*, 21(6):617–662, 2011.

[23] Oleg Kiselyov. Let-insertion as a primitive. `http://okmij.org/ftp/ML/MetaOCaml.html#genlet`, 2017. (Accessed on January 10, 2021).

[24] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, page 142–153, New York, NY, USA, 1995. Association for Computing Machinery.

[25] Rowan Davies. 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*, pages 184–195, 1996.

[26] Xavier Leroy. A modular module system. *J. Funct. Program.*, 10(3):269–303, 2000.

[27] Yuhi Sato, Yukiyoshi Kameyama, and Takahisa Watanabe. Module generation without regret. In Casper Bach Poulsen and Zhenjiang Hu, editors, *Proceedings of the 2020 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM@POPL 2020, New Orleans, LA, USA, January 20, 2020*, pages 1–13. ACM, 2020.

[28] Yuhi Sato. lambda-mg. `https://github.com/4423/lambda-mg`, 2020. (Accessed on January 10, 2021).

[29] Yuhi Sato. lambda-ma. `https://github.com/4423/lambda-ma`, 2020. (Accessed on January 10, 2021).

[30] Oleg Kiselyov. 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.*, pages 1–22, 2015.

[31] Lionel Parreaux and Amir Shaikhha. Multi-stage programming in the large with staged classes. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2020, page 35–49, New York, NY, USA, 2020. Association for Computing Machinery.

[32] Jeremy Yallop and Leo White. Modular macros. OCaml Users and Developers Workshop, 2015.

[33] Xavier Leroy et al. Chapter 21. Optimisation with Flambda. The OCaml system release 4.09, 2019.

[34] MLton, a whole program optimizing compiler for Standard ML. `http://www.mlton.org`, 1997. (Accessed on January 10, 2021).

[35] Jeremy Yallop and Oleg Kiselyov. 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*, pages 75–81, 2019.