Staged Compilation with Module Functors

TSUNG-JU CHIANG, University of Toronto, Canada
JEREMY YALLOP, University of Cambridge, United Kingdom
LEO WHITE, Jane Street Capital, United Kingdom
NINGNING XIE, University of Toronto, Canada

Multi-stage programming has been used in a wide variety of domains to eliminate the tension between abstraction and performance. However, the interaction of multi-stage programming features with features for programming-in-the-large remains understudied, hindering the full integration of multi-stage programming support into existing languages, and limiting the effective use of staging in large programs.

We take steps to remedy the situation by studying the extension of MacoCaml, a recent OCaml extension that supports compile-time code generation via *macros* and *quotations*, with module functors, the key mechanism in OCaml for assembling program components into larger units. We discuss design choices related to evaluation order, formalize our calculus via elaboration, and show that the design enjoys key metatheoretical properties: syntactic type soundness, elaboration soundness, and phase distinction. We believe that this study lays a foundation for the continued exploration and implementation of the OCaml macro system.

CCS Concepts: • **Software and its engineering** → **Macro languages**; **Modules / packages**; **Source code generation**; **Functional languages**; **Semantics**; • **Theory of computation** → **Type theory**.

Additional Key Words and Phrases: Staging, Macros, Modules, Compile-time code generation, OCaml

ACM Reference Format:
Tsung-Ju Chiang, Jeremy Yallop, Leo White, and Ningning Xie. 2024. Staged Compilation with Module Functors. *Proc. ACM Program. Lang.* 8, ICFP, Article 260 (August 2024), 35 pages. https://doi.org/10.1145/3674649

1 Introduction

Multi-stage programming, implemented in a variety of programming languages [Calcagno et al. 2003; Kiselyov 2014; Kovács 2022; Rompf and Odersky 2010; Sheard and Jones 2002; Syme 2006; Taha et al. 1998; Xie et al. 2022], is a widely-used approach to program generation. Program generation is the leading approach to resolving the tension between abstraction and performance, and language support for multi-stage programming often provides additional guarantees (e.g. that well-typed generating programs generate only well-scoped and well-typed code).

MacoCaml [Xie et al. 2023] is a recent extension to the OCaml language that supports compile-time multi-stage programming by combining a notion of macros with phase separation and quotation-based staging. The following code shows the classic power example of staging defined as a macro in MacoCaml (the syntax of MacoCaml will be explained in more detail in §2.1):

```
macro rec mpower n x =  (* int → int expr → int expr *)
  if n = 0 then << 1 >> else << $x * $(mpower (n - 1) x)>>
```

Authors' Contact Information: Tsung-Ju Chiang, University of Toronto, Toronto, Canada, tsungju.chiang@mail.utoronto.ca; Jeremy Yallop, University of Cambridge, Cambridge, United Kingdom, jeremy.yallop@cl.cam.ac.uk; Leo White, Jane Street Capital, London, United Kingdom, lwhite@janestreet.com; Ningning Xie, University of Toronto, Toronto, Canada, ningningxie@cs.toronto.edu.

ACM 2475-1421/2024/8-ART260
https://doi.org/10.1145/3674649

Intuitively speaking, `mpower` calculates x^n efficiently for any fixed `n` by unrolling and inlining all recursive calls, with help from the staging annotations for quotation (`<<>>`) and splicing (`$`). For example, `$(mpower 3 <<2>>)` generates `(2 * (2 * (2 * 1)))` at compile-time, which then calculates 2^3 efficiently at run-time. Moreover, MacoCaml supports macros inside modules. For example, `mpower` can be put inside a module `P` and later used as `P.mpower`.

However, MacoCaml does not support *module functors*, i.e. functions from modules to modules. Since all OCaml code resides within modules, functors are a key abstraction mechanism: they allow any definition — of a function, a class, a datatype, etc.— to be parameterized by any other definition. Since macros also reside within modules in MacoCaml, extending MacoCaml with functor support enables parameterizing macros by arbitrary OCaml definitions, and other definitions by macros, significantly increasing the expressive power and usefulness of macros.

Supporting functors with compile-time code generation involves new challenges: functors may include compile-time computations that depend on macros provided by their arguments; the computations cannot be evaluated before the macros are provided. Consider the following program:

```
module F(M : struct macro mpower : int → int expr → int expr end) = struct
  let x = $(M.mpower 3 <<2>>)  (* how to expand? *)
end
```

In this definition we do not know how to expand `$(mpower 3 <<2>>)`, since `mpower` is a macro from the functor argument `M` which has not yet been provided. Only when the functor `F` is applied, as `F(P)`, do we know that `M` is bound to `P`.

The remainder of this paper presents our answer to these challenges in the form of a language design that combines compile-time code generation and module functors. It is structured as follows:

- §2 presents a language design that provides a principled approach to combining functors and compile-time code generation by integrating functors with MacoCaml's macros.
- §3 formalizes a source calculus $maco^F$ with macros, quotation based staging, modules, functors, and references. Being feature-rich, $maco^F$ provides a foundation for our design to be integrated into a full-scale language.
- §4 presents a compilation target calculus $maco^F_{core}$, which enforces phase separation between modules and structures living in different phases. We prove syntactical type soundness.
- §5 presents an elaboration of $maco^F$ calculus to $maco^F_{core}$, with compile-time code generation and explicit compile-time heaps. The elaboration extends Xie et al. [2023] in several significant ways in order to support functors and functor applications.
- §6 establishes desirable properties of elaboration, including (a) elaboration soundness from $maco^F$ to $maco^F_{core}$ (Theorem 6.1); namely, well-typed source programs generate well-typed core programs; (b) *phase distinction* (Theorem 6.2); namely, compile-time computations do not interfere with run-time computations; and (c) elaboration preserves semantics (Theorem 6.3).
- §7 considers the extension of $maco^F$ with module imports and module subtyping.
- §8 envisions the integration of our design into MacoCaml, discussing practical considerations around compilation and optimization.
- Lastly, §9 surveys related work and §10 concludes.

Our formalism is detailed; for space reasons, some rules are left to the appendix. This work focuses on OCaml, but we hope it also sheds light on type-safe compile-time code generation for other languages.

2 Overview

This section reviews the key features of MacoCaml [Xie et al. 2023] (§2.1), discusses the challenges of combining compile-time evaluation and module functors (§2.2), and presents our design (§2.3).

2.1 Background: Compile-Time Code Generation in MacoCaml

Consider the following `power` function that takes `n` and `x`, and returns the value of x^n.

```
let rec power n x = if n = 0 then 1 else x * (power (n - 1) x)  (* int → int → int *)
```

The `power` function is a typical example used to motivate code generation as an approach to eliminating run-time overhead: while `power` can take any `n`, it involves run-time overhead for each recursive call. For example, `power 5 2` recurses 5 times before it returns `32`.

Programming with staging and macros. To balance abstraction and efficiency, MacoCaml [Xie et al. 2023] combines *quotation-based staging* [Sheard and Jones 2002; Taha et al. 1998] and a notion of *macros* with phase separation [Flatt 2002] for compile-time code generation. Specifically, we can define a macro version of the power function as:

```
macro rec mpower n x =  (* int → int expr → int expr *)
  if n = 0 then << 1 >> else << $x * $(mpower (n - 1) x)>>
```

Before explaining the syntax, we show that we can now define a specialized power function such as `power5` with `n` being `5`. At compile-time, all recursive calls to `mpower` will be unrolled and inlined, generating the definition for `power5` given in the comment:

```
let power5 x = $(mpower 5 <<x>>)  (* power5 x = x * (x * (x * (x * (x * 1)))) *)
```

Now calling `power5 2` generates the same result `32` as before, but with less run-time overhead.

Let definitions, macros, and staging. We now explain how let definitions, macros, and staging annotations work together in MacoCaml. The key ideas are:

- The *quotation* annotation `<<e>>` delays a computation `e`'s evaluation by turning it into its code expression, while the *splice* annotation `$(e)` forces evaluation of a code expression `e`. From a typing perspective, if `e` has type `t` then `<<e>>` has type `t expr`, where `expr` is the type constructor for code expressions. Dually, if `e` has type `t expr`, then `$(e)` has type `t`.
- In **let** definitions, we can splice **macro**-defined identifiers, while in **macro** definitions, we can quote **let**-defined identifiers. For local variables such as `x` there must be the same number of quotations as splices between the variable's binding site and its use site.
- Top-level splices (i.e. splices not surrounded by any quotations) are evaluated at compile-time.

In the example above, the top-level splice causes `mpower 5 <<x>>` to be evaluated to generate the definition of `power5` at compile-time.

Levels. More formally, definitions and staging are managed through the notion of a *level*. Roughly speaking, for an expression, its level is defined as the integer given by the number of quotes surrounding it minus the number of splices. Intuitively, levels correspond to the *evaluation phase* of an expression, where expressions of negative levels are evaluated at compile-time, while expressions of level 0 are evaluated at run-time.[1]

Let definitions and macros are then simply definitions at different levels: **let** definitions are type-checked and bound at level 0, while **macro**s are type-checked and bound at level -1.[2]

Typing is then associated with a level. To type-check `<<e>>` at level n, we type-check `e` at level $n + 1$; and to type-check `$(e)` at level n, we type-check `e` at level $n - 1$. A local variable (such as `x`) is associated with the level at which it is introduced.

[1] In the literature, work on staged programming (e.g. Xie et al. [2022]) often refers to level 0 as run-time and positive levels as future stages, while work on macros (e.g. Flatt [2002]) uses phase 1 as compile-time and phase 0 as runtime. To reduce confusion, in this work we use levels as a syntactic notion, and use compile-time and run-time for evaluation phases.

[2] We remark that the term *macro* in the context of MacoCaml means compile-time bindings, which differs from macros in systems like Racket; see Xie et al. [2023] for more discussion.

```
module type MONOID = sig
  type t
  val one : t
  val mul : t → t → t
  val show : t → string
end
```

```
module R(M : MONOID) = struct
  let rec rpower n x =  (* int → M.t → M.t *)
    if n = 0 then M.one
    else M.mul x (rpower (n - 1) x)
  let show_power5 x = M.show (rpower 5 x)
end
```

Fig. 1. A module functor

The type system ensures *well-stagedness* by enforcing *the level restriction*: a identifier or a variable can only be used at the level at which it is bound. As an example, in both `power` and `power5`, variables `n` and `x` are used at the level at which they are bound. The level restriction ensures that compile-time code generation does not depend on a value that is only known at run-time. In `power5`, the argument `n` is provided at compile-time, while `x` is provided at run-time, and the compile-time evaluation of `mpower 5 <<x>>` does not depend on the value of `x`.

To summarize, MacoCaml provides a unifying framework for macros and staging. MacoCaml also supports side-effects with references, modules, and module imports. We refer the reader to Xie et al. [2023] for more detailed explanations and discussion.

2.2 Combining Compile-Time Computations with Functors

However, MacoCaml does not yet support *module functors*. In this section we illustrate the difficulties of combining compile-time computations and functors.

Functors. A functor in OCaml is a parameterized module taking a module as an input and producing a module as output. Functors are used to abstract over module dependencies and assemble programs from typed components. Fig. 1 presents a signature `MONOID` of monoid values, which contains a type `t`, and definitions `one`, `mul`, and `show`.

We can then generalize the `power` function (§2.1), previously limited to integers, to `rpower` that works over a monoid. Specifically, the functor `R` on the right of Fig. 1 takes a `MONOID` module as an input, and returns a module containing a `rpower` function defined using `M.one` and `M.mul`. To get the original power function for integers, we can define an `Int` module and apply `R` to `Int`.

```
module Int = struct type t = int
                    let one = 1
                    let mul = ( * )
                    let show = string_of_int end
module RInt = R(Int)  (* RInt.rpower : int → int → int *)
```

However, this introduces a level of indirection, as `RInt.rpower` is a function resulting from a run-time functor application. We hope to eliminate the overhead using macros.

Compile-time computations in functors. Since MacoCaml supports macros inside modules, we may naturally want to generate the definition of `rpower` at compile-time for any specific monoid. To this end, envisaging an extension of MacoCaml to functors, we define `one` and `mul` to be macro members in `MONOID`, which get spliced inside the functor `F` (Fig. 2)

Note that type signatures of `one` and `mul` now take and return `exprs`. In the definition of functor `F`, the function `fpower` splices `M.one` and `M.mul`, so that they both get expanded at compile-time. Then, given an `Int` module with macro members, we would like to apply `F` to `Int` and generate at compile-time a `fpower` function that expands to exactly the integer-specific `power` definition (Fig. 3).

```
module type MONOID = sig
  type t
  macro one : t expr
  macro mul : t expr → t expr → t expr
  val show : t → string
end
```

```
module F(M : MONOID) = struct
  (* int → M.t → M.t *)
  let rec fpower n x =
    if n = 0 then $(M.one)
    else $(M.mul <<x>> <<fpower (n - 1) x>>)
  let show_power5 x = M.show (fpower 5 x)
end
```

Fig. 2. Compile-time computations inside a functor

```
module Int = struct type t = int
                    macro one = <<1>>
                    macro mul x y = << $(x) * $(y) >>
                    let show = string_of_int end
module FInt = F(Int)  (* let rec fpower n x = if n = 0 then 1
                                                else x * (fpower (n-1) x) *)
```

Fig. 3. Functor application with macros

Unfortunately, MacoCaml does not yet support macros either in functor arguments or functor bodies, so the program does not work. Furthermore, combining functors and MacoCaml's facilities for compile-time code generation is not entirely trivial. In particular, in MacoCaml top-level splices are evaluated at compile-time, but when `F` (Fig. 2) is compiled its argument `M` has not been provided, so it is not possible to evaluate `$(M.one)` (or `$(M.mul ...)`) in its body. It is only when `M` is instantiated with `Int` in the functor application `F(Int)` above that these definitions become available. More generally, the functor `F` can be applied multiple times, and thus may generate different run-time versions of `fpower` depending on the module argument `M`.

2.3 Our Approach

In this section, we present our key design that combines functors and macros. Along the way we explore the design space. We discuss practical considerations in §8.

Macros inside functors. What makes the definition of the functor `F` tricky in Fig. 2? We would like to evaluate the top-level splice `$(M.mul ...)`, but its evaluation depends on the macro definition from the functor argument `M`, which is not available when `F` is type-checked.

Our approach is to provide a design for functors that makes reasoning easy. That is,

Design Choice 1. *Top-level splices do not evaluate inside functors.*

Thus, the top-level splice in Fig. 2 does not evaluate.

This design does not only prevent evaluation in cases where dependencies make evaluation impossible. For example, it also prevents the following top-level splice from evaluating, even though the expression in the splice does not depend on macros from the functor argument.

```
module H(M : MONOID) = struct
  macro rec mpower n x = ...
  let power5 = fun x → $(mpower 5 <<x>>)  (* no expansion *)
end
```

This is not the only possible design: we could instead evaluate top-level splices that do not depend on macros from the functor argument. However, this alternative design is sensitive to code refactoring,

as any code change may introduce a dependency on the functor argument in a top-level splice, which then again blocks the top-level splice from evaluation, potentially causing surprising and unexpected results for programmers. For instance, in the example above, we might change the expression inside the top-level splice to `$(mpower 5 (M.mul <<x>> <<x>>))`, which then cannot evaluate. Design Choice 1 provides a uniform treatment of functors, making it easy for programmers to predict when top-level splices will (or will not) be evaluated.

On the other hand, with Design Choice 1, one may worry that turning a module into a functor changes the semantics of top-level splices. We believe that such a change is more expected and also aligns with the fact that functors are treated as values in OCaml; for example, a let definition such as `let x = print_endline "hello"` inside a structure will be evaluated when the structure is evaluated, but the same definition inside a functor will not be evaluated until the functor is applied.

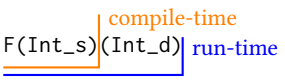
Functor applications. When do top-level splices inside a functor get evaluated, then? The answer is: during compilation, at the point where the functor is applied.

Design Choice 2. *Top-level splices inside a functor are evaluated during compilation at the point where the functor is applied.*

Applying this principle to our previous example with functor `H`, we expect `H(Int)` to expand `power5`. This case is straightforward, since `power5` does not actually depend on anything from `M`.

The example in Fig. 2, however, is more difficult. Specifically, functor applications such as `F(Int)` are not evaluated until run-time, but the top-level splices inside `F` need to happen at compile-time, with `M` instantiated to `Int` in this case.

Our solution is to split a functor argument into two arguments, one static and one dynamic, and evaluate the application to the static argument at compile-time:



Correspondingly, we also split each module into a dynamic module and a static one:

```
module Int_s = struct type t = int
                      macro one = <<1>>
                      macro mul x y = << $(x) * $(y) >> end

module Int_d = struct let show = string_of_int end
```

Fig. 4. Split the module `Int` (Fig. 3)

More specifically, the functor argument `Int` is split into two modules, `Int_s` and `Int_d`, where macros belong to the static module `Int_s`, while let definitions belong to the dynamic one `Int_d`. We expect the application `F(Int_s)` to evaluate at compile-time, evaluating top-level splices in `F`, and generating a functor that takes `Int_d` at run-time.

For this design to work out, we need to take care of a few details. First, how should we represent the two modules from splitting a module, where the two modules can be mutually dependent, as a macro can quote a let definition, and a let definition can splice a macro? We observe that when a let definition splices a macro, the macro must be part of a top-level splice. Thus, after compile-time evaluation, top-level splices have been evaluated and there will be no dependency of macro definitions on let definitions. Therefore, we can put the dynamic module in scope when typing the static part. While there will still be dependency from the dynamic module to the type

components in the static module, splitting a module with a type component has been studied before [Harper et al. 1989]; in this work, we focus on concerns specific to macros.[3]

A more problematic question is: if `FInt` (i.e. `F(Int_s)(Int_d)` after splitting) is itself passed as an argument to another functor, what should be the result of splitting `FInt`? In this particular case, `F` (and thus `FInt` also) has no static components, but more generally, a functor may contain macro definitions that may depend on macros as well as let definitions from its argument. To make the point clear, we extend `F` with a macro `showOne` that uses `M.show` and `M.one`:

```
module F(M : MONOID) = struct
  ...  (* same as before *)
  macro showOne () = <<M.show $(M.one)>>
end

FInt = F(Int)  (* with the extended F *)
```

Fig. 5. Extend functor `F` (Fig. 2) with a macro

To split `FInt` into a static and a dynamic module, clearly we cannot just leave the result of `F(Int_s)(Int_d)` to be a functor application `F'(Int_d)`, where `F'` is the result of `F(Int_s)`. We still need to look inside `F'` to retrieve its macro and let definitions. That is, we expect to split `Fint` also into two modules:

```
module FInt_s = struct
  macro showOne () = <<Int_d.show $(Int_s.one) >>       (* Int_s and Int_d inlined *)
end

module FInt_d = struct
  let rec fpower n x = if n = 0 then 1 else x * fpower (n - 1) x   (* Int_s expanded *)
  let show_power5 x = Int_d.show (fpower 5 x)                       (* Int_d inlined *)
end
```

Fig. 6. Split functor application `FInt` with `F` extended with macros (Fig. 5)

To this end, we do not only evaluate `F(Int_s)`, but also *inline* (without expanding or evaluating) the application to `Int_d` at compile-time. This is sufficient, as the functor `F` needs the *values* of macros from `Int_s` for compile-time evaluation, while it needs only the *names* of the let definitions from `Int_d`. Specifically, `Int_s.one` and `Int_s.mul` have both been expanded during compile-time evaluation of `fpower`, while `Int_d.show` appears in the result but its definition is not needed. By evaluating the application to `Int_s` and inlining the application to `Int_d`, we can successfully split `FInt` into Fig. 6 so that it can also be provided as an argument to another functor.

Functor applications to anonymous modules. As described above, for functor applications, we would like to split the module argument into a static and a dynamic module, evaluate the application to the static part, and inline the application to the dynamic part. However, the design does not take into consideration the case when the functor argument is an anonymous module, rather than a variable, in which case inlining is problematic. As an example, consider

[3]Intuitively speaking, we may split a module into three modules, one with only type components, one with macros, and the other with let definitions.

```
module FPrintShow = F(struct
  type t = int;; macro one = <<1>>;; macro mul x y = << $(x) * $(y) >>
  let show = print_endline "show"; string_of_int
end)
```

The main difference between this module argument and the previous `Int` (Fig. 3) module is that the argument here is an anonymous module, and also `show` has a side effect. Splitting the argument into a static one and a dynamic one, we have the dynamic one include the `show`. If `show` is used in `F` multiple times, then naively inlining it will cause `"show"` to be wrongly printed multiple times. A similar and slightly worse example is when a definition in the dynamic part creates a reference, in which case the reference will get wrongly duplicated, changing the semantics of the program.

Moreover, anonymous modules also causes another problem. Consider that the user wants to splice `showOne` from `FPrintShow` and observe the code generated:

```
$(FPrintShow.showOne ())  (* ???.show 1 *)
```

The generated program contains a reference to `show`. However, `show` is a function from the anonymous module and there is no way to refer to the anonymous module after the functor has been applied. How, then, can we print the generated program?

We resolve both problems by inserting the module argument as an additional module definition, which is given a fresh module name. This leads to our last design choice:

Design Choice 3. *Module arguments are inserted as additional module definitions.*

In the above example, we insert the anonymous module argument as an additional module, say `Anon`, which is then split into a static module `Anon_s` and `Anon_d`. The functor application becomes `F(Anon_s)(Anon_d)`, where the application to `Anon_s` gets evaluated, and `Anon_d` gets inlined.

We also need to expose the additional modules alongside `FPrintShow`, so that the generated program may refer to components of those modules (even though the modules themselves cannot be directly accessed by the user). Moreover, observe that the generated program can only ever refer to the dynamic part (in this case `M.show`) from the module arguments, but not the static part (in this case `M.one`, which has been expanded to `1`). We therefore only need to expose the dynamic part, not the static part, of those modules.

In the formalism that follows, we concretize Design Choice 3 in two ways. First, we insert the dynamic part of an anonymous module argument as an additional module. Again, the programmer cannot directly refer to the components of these modules; they are accessible only in code generated by compile-time computations. Second, we locally bind the static part of the argument to the functor body. For the example above, we will generate:

```
module Anon_d = struct let show = print_endline "show"; string_of_int end
```

Therefore

```
$(FPrintShow.showOne ()) (* Anon_d.show 1 *)
```

Summary. This section used examples to explain our key design that combines macros, compile-time computations, and functors. Fig. 7 shows the relations between the main code fragments presented in this section. The rest of the paper formalizes our design as a source calculus $maco^F$ (§3) which elaborates to a target calculus $maco^F_{core}$ (§4). §8 discusses practical aspects of compilation.

3 A Macro Calculus with Staging and Module Functors

This section presents $maco^F$, a macro calculus featuring staging and module functors. Our formalism is built on top of and thus shares syntax and judgments with the one in Xie et al. [2023], with the

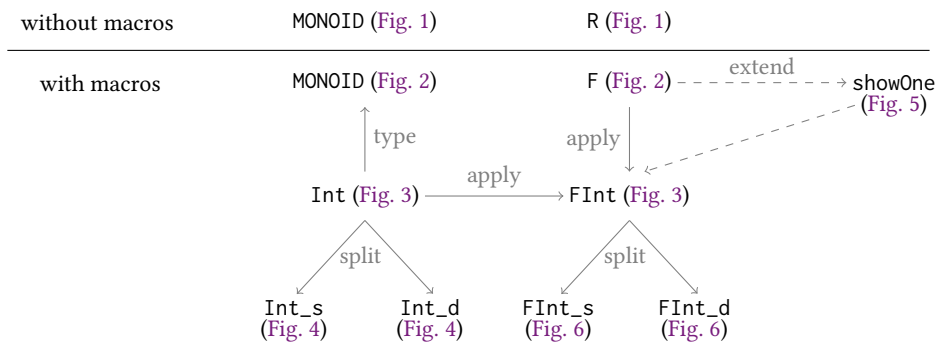


Fig. 7. Relations between main code fragments presented in §2

key extension being functors and functor applications, the main contribution of this work. We omit type definitions in the formalism, as they are largely orthogonal to our main focus. We discuss other extensions such as module imports in §7.

3.1 Syntax

Fig. 8 presents the syntax of $maco^F$. A module expression $\mathcal{M}$ can be a plain structure $\mathbf{struct}\,\mathcal{S}\,\mathbf{end}$, a module variable M, a module variable under path $p.M$, a functor abstraction $\mathbf{functor}(M : \Delta).\ \mathcal{M}$, or a functor application $\mathcal{M}_1\ \mathcal{M}_2$. We use $\mathcal{S}$ to denote a sequence of structure items, which may be empty ($\bullet$), or include module definitions $\mathbf{mod}\,M\ :\ \Delta\ =\ \mathcal{M}$, let definitions $\mathbf{def}\,k\ =\ e$, and macros $\mathbf{def}^{\downarrow}\,m\ =\ \lambda x\ :\ \tau.\ e$ that are always functions. We use **def** and $\mathbf{def}^{\downarrow}$ (instead of let and macro) to highlight that they are essentially definitions at different levels; let definitions are always typed at level 0 (run-time), and macros are typed at level -1 (compile-time). A module path p is a dot-separated sequence of module names which resolves to a module.

The term-level syntax is a lambda calculus extended with unit and integer types, mutable references, and staging constructs. An expression e can take the form of an integer literal i, a unit unit, a variable x, an abstraction $\lambda x : \tau.\ e$, or an application $e_1\ e_2$. Let definitions and macros are referenced by k and m respectively, and may be qualified by a path p. Mutable references can be created by ref e, dereferenced by !e, and assigned by $e_1 := e_2$. Lastly, $\langle e \rangle$ quotes an expression into a piece of code, and \$$e$ splices a code expression into the current stage.

A module type Δ is either an enclosed structure type $\mathbf{sig}\,\phi\,\mathbf{end}$ or a functor type $\Delta_1 \longrightarrow \Delta_2$. A structure type ϕ is a sequence of module types $M : \Delta$, let definition types $k : \tau$, and macro types $m : \tau$. Types τ include the integer type Int, the unit type Unit, functions $\tau_1 \longrightarrow \tau_2$, references Ref τ, and code type Code τ. In this work we focus on references for integers, i.e. Ref Int, as the main purpose of references is to model compile-time side-effects.

A type context Γ maps a module, a definition, and a macro to its type, and a local variable x to its type and level.

3.2 Typing Rules

Fig. 9 and 10 present the typing rules for modules, structures, and expressions in the source calculus.

Typing modules and structures. The typing judgments are mostly standard. The judgment $\Gamma \vdash \mathcal{M} : \Delta$ reads that under the typing context Γ, the module $\mathcal{M}$ has module type Δ. Rule M-STRUCT type-checks a structure. Rule M-VAR retrieves the type of a module variable from the context. Rule M-PMVAR gets the type of the module variable under a path. Note that paths are syntactically a subset of modules, and thus paths can be typed using the same judgment. The rule then gets

module	$\mathcal{M}$	::=	$\mathbf{struct}\,\mathcal{S}\,\mathbf{end} \mid M \mid p.M \mid \mathbf{functor}(M : \Delta).\ \mathcal{M} \mid \mathcal{M}_1\ \mathcal{M}_2$
structure items	$\mathcal{S}$	::=	$\bullet \mid \mathbf{mod}\ M : \Delta = \mathcal{M}; \mathcal{S} \mid \mathbf{def}\ k = e; \mathcal{S} \mid \mathbf{def}^{\downarrow}\ m = \lambda x : \tau.\ e; \mathcal{S}$
path	p	::=	$M \mid p.M$
expression	e	::=	$i \mid \mathsf{unit} \mid x \mid \lambda x : \tau.\ e \mid e_1\ e_2 \mid k \mid p.k \mid m \mid p.m$
		|	$\mathsf{ref}\ e \mid\ !e \mid e_1 := e_2 \mid \langle e \rangle \mid \$e$
module type	Δ	::=	$\mathbf{sig}\,\phi\,\mathbf{end} \mid \Delta_1 \to \Delta_2$
structure type	ϕ	::=	$\bullet \mid M : \Delta; \phi \mid k : \tau; \phi \mid m : \tau; \phi$
type	τ	::=	$\mathsf{Int} \mid \mathsf{Unit} \mid \tau_1 \to \tau_2 \mid \mathsf{Ref}\ \tau \mid \mathsf{Code}\ \tau$
context	Γ	::=	$\bullet \mid \Gamma, M : \Delta \mid \Gamma, k : \tau \mid \Gamma, m : \tau \mid \Gamma, x : (\tau, n)$

Fig. 8. Syntax in the source calculus *maco*.

$\boxed{\Gamma \vdash \mathcal{M} : \Delta}$ *(Typing module)*

$$\textsc{m-struct}\ \frac{\Gamma \vdash \mathcal{S} : \phi}{\Gamma \vdash \mathbf{struct}\,\mathcal{S}\,\mathbf{end} : \mathbf{sig}\,\phi\,\mathbf{end}} \qquad \textsc{m-mvar}\ \frac{M : \Delta \in \Gamma}{\Gamma \vdash M : \Delta} \qquad \textsc{m-pmvar}\ \frac{\Gamma \vdash p : \mathbf{sig}\,\phi\,\mathbf{end} \qquad M : \Delta \in \phi}{\Gamma \vdash p.M : \Delta}$$

$$\textsc{m-functor}\ \frac{\Gamma, M : \Delta_1 \vdash \mathcal{N} : \Delta_2}{\Gamma \vdash \mathbf{functor}\,(M : \Delta_1).\mathcal{N} : \Delta_1 \to \Delta_2} \qquad \textsc{m-app}\ \frac{\Gamma \vdash \mathcal{F} : \Delta_1 \to \Delta_2 \qquad \Gamma \vdash \mathcal{M} : \Delta_1}{\Gamma \vdash \mathcal{F}\ \mathcal{M} : \Delta_2}$$

$\boxed{\Gamma \vdash \mathcal{S} : \phi}$ *(Typing structure)*

$$\textsc{st-empty}\ \frac{}{\Gamma \vdash \bullet : \bullet} \qquad \textsc{st-def}\ \frac{\Gamma \vdash^{0} e : \tau \qquad \Gamma, k : \tau \vdash \mathcal{S} : \phi}{\Gamma \vdash (\mathbf{def}\ k = e; \mathcal{S}) : (k : \tau; \phi)} \qquad \textsc{st-macro}\ \frac{\Gamma \vdash^{-1} e : \tau \qquad \Gamma, m : \tau \vdash \mathcal{S} : \phi}{\Gamma \vdash (\mathbf{def}^{\downarrow}\ m = e; \mathcal{S}) : (m : \tau; \phi)}$$

$$\textsc{st-module}\ \frac{\Gamma \vdash \mathcal{M} : \Delta \qquad \Gamma, M : \Delta \vdash \mathcal{S} : \phi}{\Gamma \vdash (\mathbf{mod}\ M : \Delta = \mathcal{M}; \mathcal{S}) : (M : \Delta; \phi)}$$

Fig. 9. Typing of modules and structures

$\boxed{\Gamma \vdash^{n} e : \tau}$ *(Typing expression (excerpt))*

$$\textsc{quote}\ \frac{\Gamma \vdash^{n+1} e : \tau}{\Gamma \vdash^{n} \langle e \rangle : \mathsf{Code}\ \tau} \qquad \textsc{splice}\ \frac{\Gamma \vdash^{n-1} e : \mathsf{Code}\ \tau}{\Gamma \vdash^{n} \$e : \tau} \qquad \textsc{kvar}\ \frac{k : \tau \in \Gamma}{\Gamma \vdash^{0} k : \tau} \qquad \textsc{macro}\ \frac{m : \tau \in \Gamma}{\Gamma \vdash^{-1} m : \tau} \qquad \textsc{var}\ \frac{x : (\tau, n) \in \Gamma}{\Gamma \vdash^{n} x : \tau}$$

Fig. 10. Typing rules of expressions

the type of the module variable M from the path p. Rule M-FUNCTOR adds the type of the module argument $M : \Delta_1$ to the context to type-check the body $\mathcal{N}$. For rule M-APP, the first premise checks the type of $\mathcal{F}$ and the second one checks $\mathcal{M}$.

The judgment $\Gamma \vdash \mathcal{S} : \phi$ reads that under the typing context Γ, the structure $\mathcal{S}$ has structure type ϕ. Most rules are self-explanatory. Rule ST-DEF and ST-MACRO type-check let-definitions and macro definitions at level 0 and -1, respectively.

Typing expressions. For space reasons, we present only selected typing rules for staging-related constructs; the typing rules for other forms are standard. The judgment $\Gamma \vdash^{n} e : \tau$ reads that under the type context Γ, the expression e has type τ, at level n. Here we present selected rules. Rule QUOTE says that if e has type τ at level $n + 1$, then $\langle e \rangle$ has type $\mathsf{Code}\ \tau$ at level n. Dually, rule SPLICE is for splices: if e has type $\mathsf{Code}\ \tau$ at level $n - 1$, then $\$e$ has type τ at level n. Rule KVAR and rule MACRO

state that let bound variables and macros can be used only at level 0 and −1, respectively. Rule VAR captures the level restriction: a variable x can only be used at the level at which it is bound.

4 Compilation Target

The semantics of $maco^F$ is provided through an elaboration to a core calculus $maco^F_{core}$, during which we evaluate top-level splices. In this section, we give an overview of the core calculus $maco^F_{core}$; the elaboration will be presented later in §5. Intuitively speaking, $maco^F_{core}$ is a calculus with explicit syntax for modules (and structures) at different phases.

4.1 Syntax

Fig. 11 presents the syntax of the core calculus. We use colors to distinguish different syntax. Generally, symbols with different colors refer to different things, with the exception being core expressions and contexts, where dynamic and static ones share the syntax.

Modules are separated into dynamic modules $\mathcal{M}_d$ and static modules $\mathcal{M}_s$. We color the modules differently for clarity, though they share some syntax, especially expressions e. The dynamic modules $\mathcal{M}_d$ include dynamic structures $\mathbf{struct}\,\mathcal{S}_d\,\mathbf{end}$, module variables M_d, and qualified module variables $p_d.M_d$. We omit the definition for p_d, which is a dot-separated sequence of dynamic module names. The dynamic structures $\mathcal{S}_d$ are either empty ($\bullet$), or contain dynamic module definitions M_d and let definitions k. Notably, the let definition has an extra syntactic condition e^0 (explained in §4.3), which ensures that definitions have no top-level splices after compilation.

On the other hand, static modules $\mathcal{M}_s$ include static structures $\mathbf{struct}\,\mathcal{S}_s\,\mathbf{end}$, module variables M_s, and qualified module variables $p_s.M_s$, where p_s is a dot-separated sequence of static module names. Additionally, static modules include functors $\langle \Gamma, \mathbf{functor}(M : \Delta).\ \mathcal{M}\rangle$, which captures the source context Γ[4] and where the body of the functor is in the source syntax as it is not split yet. Moreover, static modules include local module bindings $\mathbf{let}\ M_s = \mathcal{M}_s\ \mathbf{in}\ \mathcal{M}_s'$. The static structures $\mathcal{S}_s$ are either empty ($\bullet$), or contain static module definitions M_s and macro definitions m. The extra syntactic condition v^0 (§4.3) ensures that macros are bound to values that contain no top-level splices. Notably, macros are always values after elaboration, as macros are functions, and there are no top-level splices. As a result, static modules and structures are also always values, since they contain only macro values and functors. (We could in principle substitute local module bindings, but since bindings are already values we leave them in the program.) Note that there are no functor applications in the syntax, as a functor application will have its static part reduced and the dynamic part inlined during elaboration.

For both types of module, we have a notion of module sequences. The module list $\mathcal{L}$ and C are a sequence of dynamic and static module variables, respectively. We write $\mapsto$ for C to highlight that static modules are always values. We also write $[C]\mathcal{M}_s$ to mean a sequence of local module definitions as in C followed by a module $\mathcal{M}_s$. For example, $[M_{s1} \mapsto \mathcal{M}_{s1}; M_{s2} \mapsto \mathcal{M}_{s2}]\mathcal{M}_s$ means $\mathbf{let}\ M_{s1} = \mathcal{M}_{s1}\ \mathbf{in}\ \mathbf{let}\ M_{s2} = \mathcal{M}_{s2}\ \mathbf{in}\ \mathcal{M}_s$

The expressions e are similar to the source expressions, but extended with locations l (highlighted in gray), which are the values of references.

Module types and structure types Δ_d, ϕ_d, Δ_s, and ϕ_s correspond to definitions of dynamic and static modules and structures, respectively. Note that Δ_s uses $\Delta_1 \to \Delta_2$ as the type for functors, as functors are not split. We omit types (τ, τ), as they are the same as types τ in the source calculus.

[4]Having the source context with the functor makes it easy to type-check the functor (rule CS-M-FUNCTOR). On the other hand, since we will establish elaboration soundness, an elaborated module will always be well-typed, in which case we do not have to capture the context but just assume that we work over well-typed core modules.

dynamic module	$\mathcal{M}_d$	::=	$\mathbf{struct}\ \mathcal{S}_d\ \mathbf{end} \mid M_d \mid p_d.M_d$
structure items	$\mathcal{S}_d$	::=	$\bullet \mid \mathbf{mod}\ M_d = \mathcal{M}_d; \mathcal{S}_d \mid \mathbf{def}\ k = e^0; \mathcal{S}_d$
path	p_d	::=	$M_d \mid p_d.M_d$
module list	$\mathcal{L}$	::=	$\bullet \mid \mathbf{mod}\ M_d = \mathcal{M}_d; \mathcal{L}$
module type	Δ_d	::=	$\mathbf{sig}\ \phi_d\ \mathbf{end}$
structure type	ϕ_d	::=	$\bullet \mid k : \tau; \phi_d \mid M : \Delta_d; \phi_d$
static module	$\mathcal{M}_s$	::=	$\mathbf{struct}\ \mathcal{S}_s\ \mathbf{end} \mid M_s \mid p_s.M_s \mid \langle \Gamma, \mathbf{functor}(M : \Delta).\ \mathcal{M} \rangle$
		|	$\mathbf{let}\ M_s = \mathcal{M}_s\ \mathbf{in}\ \mathcal{M}_s'$
structure items	$\mathcal{S}_s$	::=	$\bullet \mid \mathbf{mod}\ M_s = \mathcal{M}_s; \mathcal{S}_s \mid \mathbf{def}^{\downarrow}\ m = v^0; \mathcal{S}_s$
path	p_s	::=	$M_s \mid p_s.M_s$
module list	C	::=	$\bullet \mid M_s \mapsto \mathcal{M}_s; C$
module type	Δ_s	::=	$\mathbf{sig}\ \phi_s\ \mathbf{end} \mid \Delta_1 \longrightarrow \Delta_2$
structure type	ϕ_s	::=	$\bullet \mid m : \tau; \phi_s \mid M : \Delta_s; \phi_s$
expr	e, e	::=	$i \mid \mathsf{unit} \mid x \mid \lambda x : \tau.\ e \mid e_1\ e_2 \mid k \mid p_d.k \mid m \mid p_s.m$
		|	$\mathsf{ref}\ e \mid !e \mid e_1 := e_2 \mid \langle e \rangle \mid \$e \mid l$
context	Γ, Γ	::=	$\bullet \mid \Gamma, M_d : \Delta_d \mid \Gamma, M_s : \Delta_s \mid \Gamma, k : \tau \mid \Gamma, m : \tau \mid \Gamma, x : (\tau, n)$

Fig. 11. Syntax of the core calculus $maco^{F}_{core}$.

$\boxed{\sigma \mid \Gamma \vdash \mathcal{M}_d : \Delta_d}$ *(Typing dynamic module)*

CD-M-VAR
$$\frac{M_d : \Delta_d \in \Gamma}{\sigma \mid \Gamma \vdash M_d : \Delta_d}$$

CD-M-PATH
$$\frac{\sigma \mid \Gamma \vdash p_d : \mathbf{sig}\ \phi_d\ \mathbf{end} \qquad M : \Delta_d \in \phi_d}{\sigma \mid \Gamma \vdash p_d.M_d : \Delta_d}$$

CD-M-STRUCT
$$\frac{\sigma \mid \Gamma \vdash \mathcal{S}_d : \phi_d}{\sigma \mid \Gamma \vdash \mathbf{struct}\ \mathcal{S}_d\ \mathbf{end} : \mathbf{sig}\ \phi_d\ \mathbf{end}}$$

$\boxed{\sigma \mid \Gamma \vdash \mathcal{S}_d : \phi_d}$ *(Typing dynamic structure)*

CD-ST-EMPTY
$$\frac{}{\sigma \mid \Gamma \vdash \bullet : \bullet}$$

CD-ST-DEF
$$\frac{\sigma \mid \Gamma \vdash^{0} e : \tau \qquad \sigma \mid \Gamma, k : \tau \vdash \mathcal{S}_d : \phi_d}{\sigma \mid \Gamma \vdash (\mathbf{def}\ k = e; \mathcal{S}_d) : (k : \tau; \phi_d)}$$

CD-ST-MODULE
$$\frac{\sigma \mid \Gamma \vdash \mathcal{M}_d : \Delta_d \qquad \sigma \mid \Gamma, M_d : \Delta_d \vdash \mathcal{S}_d : \phi_d}{\sigma \mid \Gamma \vdash (\mathbf{mod}\ M_d = \mathcal{M}_d; \mathcal{S}_d) : (M_d : \Delta_d; \phi_d)}$$

$\boxed{\Gamma \vdash \mathcal{M}_s : \Delta_s}$ *(Typing static module)*

CS-M-VAR
$$\frac{M_s : \Delta_s \in \Gamma}{\Gamma \vdash M_s : \Delta_s}$$

CS-M-PATH
$$\frac{\Gamma \vdash p_s : \mathbf{sig}\ \phi_s\ \mathbf{end} \qquad M_s : \Delta_s \in \phi_s}{\Gamma \vdash p_s.M_s : \Delta_s}$$

CS-M-STRUCT
$$\frac{\Gamma \vdash \mathcal{S}_s : \phi_s}{\Gamma \vdash \mathbf{struct}\ \mathcal{S}_s\ \mathbf{end} : \mathbf{sig}\ \phi_s\ \mathbf{end}}$$

CS-M-FUNCTOR
$$\frac{\Gamma_1 \vdash \mathbf{functor}\ (M : \Delta_1).\mathcal{N} : \Delta_1 \longrightarrow \Delta_2}{\Gamma \vdash \langle \Gamma_1, \mathbf{functor}(M : \Delta_1).\ \mathcal{N} \rangle : \Delta_1 \longrightarrow \Delta_2}$$

CS-M-LOCAL
$$\frac{\Gamma \vdash \mathcal{M}_s : \Delta_s \qquad \Gamma, M_s : \Delta_s \vdash \mathcal{M}_s' : \Delta_s'}{\Gamma \vdash \mathbf{let}\ M_s = \mathcal{M}_s\ \mathbf{in}\ \mathcal{M}_s' : \Delta_s'}$$

$\boxed{\Gamma \vdash \mathcal{S}_s : \phi_s}$ *(Typing static structure)*

CS-ST-EMPTY
$$\frac{}{\Gamma \vdash \bullet : \bullet}$$

CS-ST-MACRO
$$\frac{\bullet \mid \Gamma \vdash^{-1} v : \tau \qquad \Gamma, m : \tau \vdash \mathcal{S}_s : \phi_s}{\Gamma \vdash (\mathbf{def}^{\downarrow}\ m = v; \mathcal{S}_s) : (m : \tau; \phi_s)}$$

CS-ST-MODULE
$$\frac{\Gamma \vdash \mathcal{M}_s : \Delta_s \qquad \Gamma, M_s : \Delta_s \vdash \mathcal{S}_s : \phi_s}{\Gamma \vdash (\mathbf{mod}\ M_s = \mathcal{M}_s; \mathcal{S}_s) : (M : \Delta_s; \phi_s)}$$

Fig. 12. Typing rules for modules and structures

$\boxed{\sigma \mid \Gamma \vdash^n e : \tau}$ *(Typing expression (excerpt))*

$$\textsc{c-quote}\ \frac{\sigma \mid \Gamma \vdash^{n+1} e : \tau}{\sigma \mid \Gamma \vdash^n \langle e \rangle : \mathsf{Code}\,\tau} \qquad \textsc{c-splice}\ \frac{\sigma \mid \Gamma \vdash^{n-1} e : \mathsf{Code}\,\tau}{\sigma \mid \Gamma \vdash^n \$e : \tau} \qquad \textsc{c-kvar}\ \frac{k : \tau \in \Gamma}{\sigma \mid \Gamma \vdash^0 k : \tau} \qquad \textsc{c-macro}\ \frac{m : \tau \in \Gamma}{\sigma \mid \Gamma \vdash^{-1} m : \tau}$$

$$\textsc{c-var}\ \frac{x : (\tau, n) \in \Gamma}{\sigma \mid \Gamma \vdash^n x : \tau} \qquad \textsc{c-loc}\ \frac{l \in \sigma}{\sigma \mid \Gamma \vdash^n l : \mathsf{Ref\ Int}}$$

Fig. 13. Typing rules for expressions

4.2 Typing

Fig. 12 and 13 present typing rules for modules, structures, and expressions in the core calculus.

The typing rules for dynamic modules $\mathcal{M}_d$ and structures $\mathcal{S}_d$ are mostly straightforward, and are very similar to the rules in the source calculus. The main difference is that since we have locations, the typing judgments are associated with a heap σ, defined in Fig. 14, which keeps track of references and maps locations to values (see rule C-LOC).

Interestingly, the typing judgments for static modules $\mathcal{M}_s$ and structures $\mathcal{S}_s$ are not associated with heaps; as we will see in §6, there cannot be any reference to the compile-time heap after elaboration. Also, since static modules and structures are values, they also do not evaluate. As such, expressions in the static part will not refer to locations. Rule CS-M-FUNCTOR type-checks functors using the captured source context. Finally, rule CS-M-LOCAL type-checks local module bindings, where we first type-check $\mathcal{M}_s$, and put its type into the context to type-check $\mathcal{N}_s$. Note that the type of $\mathcal{M}_s$ is not returned in the result type. We also write $\sigma \mid \Gamma \vdash \mathcal{L} : \phi_d$ (and $\Gamma \vdash \mathcal{C} : \phi_s$) that type-checks a list of modules and returns ϕ_d (and ϕ_s).

The typing judgment for expressions is more standard; here we show the two rules for staging constructs (rule C-QUOTE and C-SPLICE), the rules for identifiers and variables (rule C-KVAR, C-MACRO, and C-VAR), and one for locations (rule C-LOC), where a location is of type Ref Int, if the location appears in σ.

4.3 Dynamic Semantics

We now present the dynamic semantics of the core calculus. Since static modules and structures are already values, we only need to evaluate dynamic modules and structures.

Fig. 14 presents the definition of module values and structure values. The evaluation environments Ω_d and Ω_s store definitions needed for run-time and compile-time evaluation respectively. We also write Ω to stand for either Ω_d or Ω_s, when the distinction does not matter or it is obvious from the context which environment we refer to. We use light blue for heaps and evaluation environments for better clarity later in the elaboration rules, as they are mostly simply threaded through.

Level-annotated expressions and values. Fig. 15 defines *level-annotated expressions* and *values* [Calcagno et al. 2003; Taha et al. 1998]. The notation e^n means that e is an expression at level n, with $n \geq 0$. Notably, a splice $\$e$ is an expression at only positive levels $n+1$ (thus with $n+1 \geq 1$). Thus, e^0 in the dynamic structure items ensures that expressions do not have top-level splices.

Values v is a subset of expressions. The level-annotated *values* notation v^n means that v is a value at level n. Values v^0 at level 0 include literals i, units unit, locations l, and lambdas $\lambda x : \tau.\ e$ and quotations $\langle e \rangle$ given e^0; in other words, e is an expression at level 0 and thus it has no top-level splices. Moreover, values at higher levels (v^{n+1}) are the same set as expressions e^n. Intuitively, v^{n+1} is a value at level $n+1$ if it does not reduce at level $n+1$, and thus it can only have at most n nested

module value	$\mathcal{M}_{dv}$	::=	$\mathbf{struct}\ \mathcal{S}_{dv}\ \mathbf{end}$
structure value	$\mathcal{S}_{dv}$	::=	$\bullet \mid \mathbf{mod}\ M = \mathcal{M}_{dv}; \mathcal{S}_{dv} \mid \mathbf{def}\ k = v; \mathcal{S}_{dv}$
heap	σ	::=	$\bullet \mid \sigma, l \mapsto v$
evaluation env	Ω_d	::=	$\bullet \mid \Omega_d; M_d \mapsto \mathcal{M}_{dv} \mid \Omega_d; k \mapsto v$
	Ω_s	::=	$\bullet \mid \Omega_s; M_s \mapsto \mathcal{M}_s \mid \Omega_s; m \mapsto v$

Fig. 14. Dynamic module and structure values in the core calculus $maco_{core}$.

$\boxed{e^n\ (n{\geq}0)}$ *(level-annotated expressions)*

$$\overline{i^n} \qquad \overline{\mathsf{unit}^n} \qquad \overline{l^n} \qquad \overline{x^n} \qquad \overline{k^n} \qquad \overline{p.k^n} \qquad \overline{m^n} \qquad \overline{p.m^n} \qquad \frac{e^n}{(\lambda x : \tau.\ e)^n}$$

$$\frac{e_1^n \quad e_2^n}{(e_1\ e_2)^n} \qquad \frac{e^n}{(\mathsf{ref}\ e)^n} \qquad \frac{e_1^n \quad e_2^n}{(e_1 := e_2)^n} \qquad \frac{e^n}{(!e)^n} \qquad \frac{e^{n+1}}{(\langle e \rangle)^n} \qquad \frac{e^n}{(\$e)^{n+1}}$$

$\boxed{v^0}$ $\boxed{v^{n+1} ::= e^n\ (n{\geq}0)}$ *(level-annotated values)*

$$\overline{i^0} \qquad \overline{\mathsf{unit}^0} \qquad \overline{l^0} \qquad \frac{e^0}{(\lambda x : \tau.\ e)^0} \qquad \frac{e^0}{(\langle e \rangle)^0}$$

Fig. 15. Level-annotated expressions and values

splices, which is exactly expressions of e^n. As an example, if $\$e$ is an expression at level 1, then it is a value at level 2, which requires it to be inside more quotations.

Operational semantics. Fig. 16 presents the rules for evaluating dynamic modules and structures, where we simply reduce each let and submodule definition in the order they appear. Note that the notation $p_d \mapsto \mathcal{M}_{dv} \in \Omega_d$ used in rule EV-M-PMVAR is a chain of lookups, whose definition is given at the bottom of the figure. We use other similar lookup rules when looking for items inside modules. Moreover, we use $\Omega \vdash \lceil \cdot \rceil^{p_d}$ to prefix a path p_d to all variables that are defined in p_d; we often omit Ω when it is clear from the context. For example, if M_1 defines $\mathbf{def}\ k_1 = 1$ and $\mathbf{mod}\ M_2 = \mathbf{struct}\ \mathbf{def}\ k_2 = k_1\ \mathbf{end}$, then $M_1.M_2$ evaluates to $\mathbf{struct}\ \mathbf{def}\ k_2 = M_1.k_1\ \mathbf{end}$. The full definitions of those auxiliary judgments are put in the appendix.

We present a number of the more interesting dynamic semantics rules for expressions in Fig. 17. The judgment $\sigma_1 \mid \Omega \mid e_1 \xrightarrow{n} e_2 \mid \sigma_2$ reads: under heap σ_1 and evaluation environment Ω, evaluating e_1 at level n results to e_2 and updates the heap to σ_2. The rules are used both during compile-time (in rule E-CODEGEN in Fig. 19) for evaluating splices, and during run-time for evaluating the compiled code. Rule EV-QUOTE and EV-SPLICE update the evaluation level accordingly, so we may evaluate inside quotations and splices. Rule EV-SPLICECODE splices a quotation at level 1, at which point the splice and the quotation are both removed. The rest of the four rules concern definitions. Notably, the first two rules are for a dynamic evaluation environment only, while the last two are for a static evaluation environment.

4.4 Type Soundness

We prove syntactic type soundness [Wright and Felleisen 1994] of $maco^F_{core}$. First, we establish preservation. We use σ ok to mean that all values inside the heaps are well-typed (i.e. of type Int). Moreover, we use $\sigma \mid \Gamma \vdash \Omega_d$ and $\Gamma \vdash \Omega_s$ (Fig. 18) to mean that all definition in Ω_d (or Ω_s, respectively) are well-typed under σ and Γ.

Theorem 4.1 (Preservation). *Given* σ ok, *and* $\sigma \mid \Gamma \vdash \Omega_d$, *and* $\Gamma \vdash \Omega_s$,

$\boxed{\sigma_1 \mid \Omega \mid \mathcal{M}_d \longrightarrow \mathcal{M}_d' \mid \sigma_2}$ *(Evaluate module)*

EV-M-STRUCT
$$\frac{\sigma \mid \Omega_d \mid \mathcal{S}_d \longrightarrow \mathcal{S}_d' \mid \sigma'}{\sigma \mid \Omega_d \mid \mathbf{struct}\ \mathcal{S}_d\ \mathbf{end} \longrightarrow \mathbf{struct}\ \mathcal{S}_d'\ \mathbf{end} \mid \sigma'}$$

EV-M-MVAR
$$\frac{M \mapsto \mathcal{M}_{dv} \in \Omega_d}{\sigma \mid \Omega_d \mid M \longrightarrow \mathcal{M}_{dv} \mid \sigma}$$

EV-M-PMVAR
$$\frac{p_d.M \mapsto \mathcal{M}_{dv} \in \Omega_d}{\sigma \mid \Omega_d \mid p_d.M \longrightarrow \lceil \mathcal{M}_{dv} \rceil^{p_d} \mid \sigma}$$

$\boxed{\sigma_1 \mid \Omega \mid \mathcal{S}_d \longrightarrow \mathcal{S}_d' \mid \sigma_2}$ *(Evaluate structure)*

EV-ST-MODULE1
$$\frac{\sigma \mid \Omega_d \mid \mathcal{M}_d \longrightarrow \mathcal{M}_d' \mid \sigma'}{\sigma \mid \Omega_d \mid \mathbf{mod}\ M_d = \mathcal{M}_d; \mathcal{S}_d \longrightarrow \mathbf{mod}\ M_d = \mathcal{M}_d'; \mathcal{S}_d \mid \sigma'}$$

EV-ST-MODULE2
$$\frac{\sigma \mid \Omega_d; M_d \mapsto \mathcal{M}_{dv} \mid \mathcal{S}_d \longrightarrow \mathcal{S}_d' \mid \sigma'}{\sigma \mid \Omega_d \mid \mathbf{mod}\ M_d = \mathcal{M}_{dv}; \mathcal{S}_d \longrightarrow \mathbf{mod}\ M_d = \mathcal{M}_{dv}; \mathcal{S}_d' \mid \sigma'}$$

EV-ST-DEF1
$$\frac{\sigma \mid \Omega_d \mid e \xrightarrow{0} e' \mid \sigma'}{\sigma \mid \Omega_d \mid \mathbf{def}\ k = e; \mathcal{S}_d \longrightarrow \mathbf{def}\ k = e'; \mathcal{S}_d \mid \sigma'}$$

EV-ST-DEF2
$$\frac{\sigma \mid \Omega_d; k \mapsto v \mid \mathcal{S}_d \longrightarrow \mathcal{S}_d' \mid \sigma'}{\sigma \mid \Omega_d \mid \mathbf{def}\ k = v; \mathcal{S}_d \longrightarrow \mathbf{def}\ k = v; \mathcal{S}_d' \mid \sigma'}$$

$\boxed{p_d \mapsto \mathcal{M}_{dv} \in \Omega_d}$ *(Dynamic path lookup)*

LOOKUP-PD-VAR
$$\frac{}{M_d \mapsto \mathcal{M}_{dv} \in \Omega_{d1}; M \mapsto \mathcal{M}_{dv}; \Omega_{d2}}$$

LOOKUP-PD-PATH
$$\frac{p_d \mapsto \mathbf{struct}\ \mathcal{S}_{dv}\ \mathbf{end} \in \Omega_d \qquad M_d = \mathcal{M}_{dv} \in \mathcal{S}_{dv}}{p_d.M_d \mapsto \lceil \mathcal{M}_{dv} \rceil^{p_d} \in \Omega_d}$$

Fig. 16. Operational semantics of dynamic modules and structures

$\boxed{\sigma_1 \mid \Omega \mid e \xrightarrow{n} e' \mid \sigma_2}$ *(Evaluate expression (excerpt))*

EV-QUOTE
$$\frac{\sigma_1 \mid \Omega \mid e_1 \xrightarrow{n+1} e_2 \mid \sigma_2}{\sigma_1 \mid \Omega \mid \langle e_1 \rangle \xrightarrow{n} \langle e_2 \rangle \mid \sigma_2}$$

EV-SPLICE
$$\frac{\sigma_1 \mid \Omega \mid e_1 \xrightarrow{n} e_2 \mid \sigma_2}{\sigma_1 \mid \Omega \mid \$e_1 \xrightarrow{n+1} \$e_2 \mid \sigma_2}$$

EV-SPLICECODE
$$\frac{}{\sigma \mid \Omega \mid \$\langle v^1 \rangle \xrightarrow{1} v^1 \mid \sigma}$$

EV-KVAR
$$\frac{k \mapsto v \in \Omega_d}{\sigma \mid \Omega_d \mid k \xrightarrow{0} v \mid \sigma}$$

EV-PKVAR
$$\frac{p_d.k \mapsto v \in \Omega_d}{\sigma \mid \Omega_d \mid p_d.k \xrightarrow{0} \lceil v \rceil^{p_d} \mid \sigma}$$

EV-MACRO
$$\frac{m \mapsto v \in \Omega_s}{\sigma \mid \Omega_s \mid m \xrightarrow{0} v \mid \sigma}$$

EV-PMACRO
$$\frac{p_s.m \mapsto v \in \Omega_s}{\sigma \mid \Omega_s \mid p_s.m \xrightarrow{0} \lceil v \rceil^{p_s} \mid \sigma}$$

Fig. 17. Operational semantics of expressions

$\boxed{\sigma \mid \Gamma \vdash \Omega_d}$ *(run time)*

ENV-R-EMPTY
$$\frac{}{\sigma \mid \Gamma \vdash \bullet}$$

ENV-R-KVAR
$$\frac{\sigma \mid \Gamma \vdash \Omega \qquad \sigma \mid \Gamma \vdash^0 v : \tau}{\sigma \mid \Gamma \vdash \Omega; k \mapsto v}$$

ENV-R-MODULE
$$\frac{\sigma \mid \Gamma \vdash \Omega \qquad \sigma \mid \Gamma \vdash \mathcal{M}_d : \Delta_d}{\sigma \mid \Gamma \vdash \Omega; M_d \mapsto \mathcal{M}_d}$$

$\boxed{\Gamma \vdash \Omega_s}$ *(compile time)*

ENV-C-EMPTY
$$\frac{}{\Gamma \vdash \bullet}$$

ENV-C-MACRO
$$\frac{\Gamma \vdash \Omega \qquad \bullet \mid \Gamma \vdash^{-1} v : \tau}{\Gamma \vdash \Omega; m \mapsto v}$$

ENV-C-MODULE
$$\frac{\Gamma \vdash \Omega \qquad \Gamma \vdash \mathcal{M}_s : \Delta_s}{\Gamma \vdash \Omega; M_s \mapsto \mathcal{M}_s}$$

Fig. 18. Well-formedness of evaluation environments. Modules in the environment are checked similarly.

(modules) *if* $\sigma \mid \Gamma \vdash \mathcal{M}_d : \Delta_d$ *and* $\sigma \mid \Omega_d \mid \mathcal{M}_d \longrightarrow \mathcal{M}_d' \mid \sigma'$, *then* $\sigma' \mid \Gamma \vdash \mathcal{M}_d' : \Delta_d$.
(structures) *if* $\sigma \mid \Gamma \vdash \mathcal{S}_d : \phi_d$ *and* $\sigma \mid \Omega_d \mid \mathcal{S}_d \longrightarrow \mathcal{S}_d' \mid \sigma'$, *then* $\sigma' \mid \Gamma \vdash \mathcal{S}_d' : \phi_d$.
(expressions) *if* $\sigma \mid \Gamma \vdash^{n'} e : \tau$ *and* $\sigma \mid \Omega_d \mid e \xrightarrow{n} e' \mid \sigma'$ *or* $\sigma \mid \Omega_s \mid e \xrightarrow{n} e' \mid \sigma'$, *then* $\sigma' \mid \Gamma \vdash^{n'} e' : \tau$.

Note that in preservation for expressions, the level at which the expression is typed (n') and at which the expression is evaluated (n) can be different, and the theorem proves that every possible step preserves typing. As an example, $(\lambda x : \mathsf{Int}.\ x)\ 1$ is well-typed at any level, but only evaluates at level 0, and the evaluation result 1 is again well-typed at any level. In fact, we rely on this during elaboration (rule CODEGEN in Fig. 19), where an expression inside a top-level splice may be typed at a negative level while evaluated at level 0.

We then prove progress, which uses level-annotated expressions and values. The notation $\hat{\Gamma}$ is a judgment ensuring that Γ does not contain local variables (i.e. x).

Theorem 4.2 (Progress). *Given* $\hat{\Gamma}$, σ ok, $\sigma \mid \Gamma \vdash \Omega_d$, *and* $\Gamma \vdash \Omega_s$,
(modules) *if* $\sigma \mid \Gamma \vdash \mathcal{M}_d : \Delta_d$, *then either* $\mathcal{M}_d$ *is a value, or there exist* $\mathcal{M}_d'$ *and* σ' *such that* $\sigma \mid \Omega_d \mid \mathcal{M}_d \longrightarrow \mathcal{M}_d' \mid \sigma'$.
(structures) *if* $\sigma \mid \Gamma \vdash \mathcal{S}_d : \phi_d$, *then either* $\mathcal{S}_d$ *is a value, or there exist* $\mathcal{S}_d'$ *and* σ' *such that* $\sigma \mid \Omega_d \mid \mathcal{S}_d \longrightarrow \mathcal{S}_d' \mid \sigma'$.
(expressions) *if* $\sigma \mid \Gamma \vdash^{n'} e : \tau$ *where* e^n, *then either* e *is a value* v^n, *or there exist* e' *and* σ' *such that* $(n' < n \wedge \sigma \mid \Omega_s \mid e \xrightarrow{n} e' \mid \sigma')$ *or* $(n' \geq n \wedge \sigma \mid \Omega_d \mid e \xrightarrow{n} e' \mid \sigma')$.

Notably, progress for expressions must hold for both compile-time and run-time: when the typing level is smaller than the evaluation level, it indicates compile-time evaluation (e.g. top-level splices are type-checked at a negative level but evaluated at level 0) and thus the evaluation requires Ω_s; otherwise, it is at run-time and the evaluation requires Ω_d.

5 Elaboration

So far we have introduced the source calculus $maco^F$ (§3) and the core calculus $maco^F_{core}$ (§4). In this section, we describe an elaboration from $maco^F$ to $maco^F_{core}$, the key contribution of this paper. We establish desirable properties of elaboration in §6.

Recall the three key design choices described in §2.3:

Design Choice 1 Top-level splices do not evaluate inside functors.
Design Choice 2 Top-level splices inside a functor are evaluated during compilation at the point where the functor is applied.
Design Choice 3 Module arguments are inserted as additional module definitions.

At a high level, the elaboration, presented in Fig. 19, 20, and 21, implements these designs by doing a few things at the same time:

(a) Compile-time code generation for top-level splices 1) when not inside functors (§5.1); or 2) when functors are applied (§5.2.2).
(b) Split a module (or a structure) into a dynamic one and a static one (§5.2.1;§5.3);
(c) Insert dynamic module arguments, and locally bind static module arguments (§5.3).

We go through these steps in the rest of this section, but we first present elaboration of types.

Elaboration of types. The following shows how types in the source are elaborated to types in the core, where $(\phi)_d$ and $(\Delta)_d$ get the dynamic part of the type, and $(\phi)_s$ and $(\Delta)_s$ get the static part:

$$(\bullet)_d = \bullet \quad (k : \tau; \phi)_d = k : \tau; (\phi)_d \quad (m : \tau; \phi)_d = (\phi)_d \quad (M : \Delta; \phi)_d = M_d : (\Delta)_d; (\phi)_d$$
$$(\bullet)_s = \bullet \quad (k : \tau; \phi)_s = (\phi)_s \quad (m : \tau; \phi)_s = m : \tau; (\phi)_s \quad (M : \Delta; \phi)_s = M_s : (\Delta)_s; (\phi)_s$$
$$(\mathbf{sig}\ \phi\ \mathbf{end})_d = \mathbf{sig}\ (\phi)_d\ \mathbf{end} \qquad (\Delta_1 \longrightarrow \Delta_2)_d = \mathbf{sig}\ \bullet\ \mathbf{end}$$
$$(\mathbf{sig}\ \phi\ \mathbf{end})_s = \mathbf{sig}\ (\phi)_s\ \mathbf{end} \qquad (\Delta_1 \longrightarrow \Delta_2)_s = \Delta_1 \longrightarrow \Delta_2$$

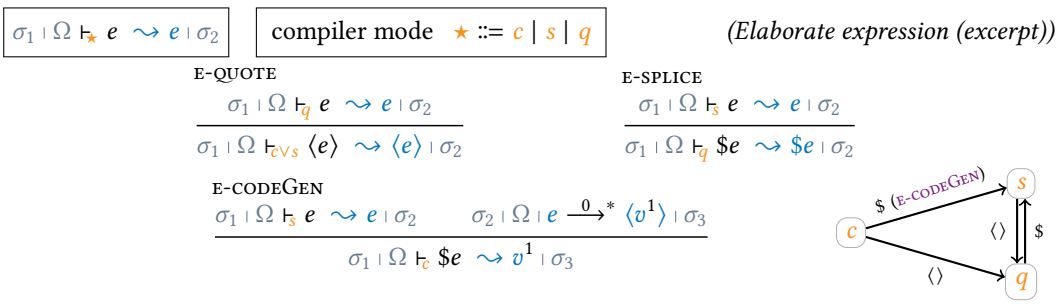


Fig. 19. Elaboration of expressions

Specifically, $k : \tau$ appears in the dynamic type but not in the static type; dually, $m : \tau$ appears in the static type but not the dynamic type. A module variable M elaborates to M_d and M_s respectively for the dynamic and the static part. Lastly, the functor type generates an empty dynamic module, while returning the same functor type for the static type.

5.1 Compile-Time Code Generation

As described, top-level splices are evaluated during compilation [Xie et al. 2023].

Fig. 19 presents the elaboration rule of expressions. The judgment $\sigma_1 \mid \Omega \vdash_\star e \rightsquigarrow e \mid \sigma_2$ reads: under the heap σ_1, the evaluation context Ω, the expression e under mode $\star$ elaborates to e, and updates the heap to σ_2. In contrast to the corresponding source typing judgement, the judgment threads through a compile-time heap, since compile-time evaluation can allocate references, keeps track of the (static) evaluation environment Ω, and is associated with a compiler mode $\star$. Moreover, elaboration does not need a level number.

The definition of compiler modes appears at the top of Fig. 19; modes manages compile-time code generation for top-level splices, similar to the *typing state* in Template Haskell [Sheard and Jones 2002]. The transitions between the three modes are shown on the right of the figure.

More specifically, when elaborating expressions inside structures (rule E-ST-DEF and E-ST-MACRO in Fig. 20), the compiler is in mode c. If the compiler then encounters a splice, it must be a top-level splice $\$e$. In this case, the compiler applies rule E-CODEGEN to evaluate the top-level splice, which first elaborates e to e, and then evaluates e into a quotation $\langle v^1 \rangle$ that cannot be reduced further, where v is a value at level 1 (thus $\langle v^1 \rangle$ is a value at level 0). The compiler then removes the quotation and inserts v^1 as the elaboration result. On the other hand, if the compiler encounters a quotation, it applies rule E-QUOTE, which transitions into mode q. Once inside mode q, the program can transition to s (rule E-SPLICE) and also go back to q (rule E-QUOTE), but it cannot go back to c. This way, only top-level splices trigger compile-time code generation.[5]

Lastly, we remark that rule E-CODEGEN applies at any level: there can be top-level splices inside a let definition at level 0 (rule E-ST-DEF) or a macro definition at level -1 (rule E-ST-MACRO).

[5]As a side note, the transition disallows nested quotes and splices [Xie et al. 2023]. Extending the transition with e.g. nested quotations is easy, but supporting nested splices is subtler. In particular, one may expect an expression $\$(\$e_1)$ with more nested splices to be evaluated before another expression $\$e_2$, which is however difficult to model during typing. Typed Template Haskell (TTH) [Xie et al. 2022] solves this by lifting splices according to their levels to the top-level; in the above case, e_1 will be put before e_2. Integrating a similar approach is future work. We also mark that both Template Haskell [Sheard and Jones 2002] and Scala [Stucki et al. 2018] disallow nested splices.

$\boxed{\sigma_1 \mid \Omega \mid \Gamma \vdash \mathcal{M} \rightsquigarrow \mathcal{L} \mid \mathcal{M}_d \mid \mathcal{M}_s \mid \sigma_2}$ *(Elaborate module)*

$$\textsc{E-M-STRUCT}\ \frac{\sigma_1 \mid \Omega \mid \Gamma \vdash \mathcal{S} \rightsquigarrow \mathcal{S}_d \mid \mathcal{S}_s \mid \sigma_2}{\sigma_1 \mid \Omega \mid \Gamma \vdash \mathbf{struct}\ \mathcal{S}\ \mathbf{end} \rightsquigarrow \bullet \mid \mathbf{struct}\ \mathcal{S}_d\ \mathbf{end} \mid \mathbf{struct}\ \mathcal{S}_s\ \mathbf{end} \mid \sigma_2}$$

$$\textsc{E-M-MVAR}\ \frac{}{\sigma \mid \Omega \mid \Gamma \vdash M \rightsquigarrow \bullet \mid M_d \mid M_s \mid \sigma} \qquad \textsc{E-M-PMVAR}\ \frac{\sigma \mid \Omega \mid \Gamma \vdash p \rightsquigarrow \bullet \mid p_d \mid p_s \mid \sigma}{\sigma \mid \Omega \mid \Gamma \vdash p.M \rightsquigarrow \bullet \mid p_d.M_d \mid p_s.M_s \mid \sigma}$$

$$\textsc{E-M-FUNCTOR}\ \frac{}{\sigma \mid \Omega \mid \Gamma \vdash \mathbf{functor}\ (M : \Delta_1).\mathcal{N} \rightsquigarrow \bullet \mid \mathbf{struct}\ \bullet\ \mathbf{end} \mid \langle \Gamma, \mathbf{functor}(M : \Delta_1).\ \mathcal{N} \rangle \mid \sigma}$$

$$\textsc{E-M-APP}\ \frac{\sigma_1 \mid \Omega \mid \Gamma \vdash \mathcal{F} \rightsquigarrow \mathcal{L}_1 \mid \mathcal{F}_d \mid \mathcal{F}_s \mid \sigma_2 \qquad \sigma_2 \mid \Omega \mid \Gamma \vdash \mathcal{M} \rightsquigarrow \mathcal{L}_2 \mid \mathcal{M}_d \mid \mathcal{M}_s \mid \sigma_3 \qquad \sigma_3 \mid \Omega \vdash \mathcal{F}_s \bullet \mathcal{M}_d \mid \mathcal{M}_s \Longrightarrow \mathcal{L}_3 \mid \mathcal{N}_d \mid \mathcal{N}_s \mid \sigma_4}{\sigma_1 \mid \Omega \mid \Gamma \vdash \mathcal{F}\ \mathcal{M} \rightsquigarrow \mathcal{L}_1; \mathcal{L}_2; \mathcal{L}_3 \mid \mathcal{N}_d \mid \mathcal{N}_s \mid \sigma_4}$$

$\boxed{\sigma_1 \mid \Omega \mid \Gamma \vdash \mathcal{S} \rightsquigarrow \mathcal{S}_d \mid \mathcal{S}_s \mid \sigma_2}$ *(Elaborate structure)*

$$\textsc{E-ST-EMPTY}\ \frac{}{\sigma \mid \Omega \mid \Gamma \vdash \bullet \rightsquigarrow \bullet \mid \bullet \mid \sigma} \qquad \textsc{E-ST-DEF}\ \frac{\sigma_1 \mid \Omega \vdash_c e \rightsquigarrow e \mid \sigma_2 \qquad \sigma_2 \mid \Omega \mid \Gamma, k : \tau \vdash \mathcal{S} \rightsquigarrow \mathcal{S}_d \mid \mathcal{S}_s \mid \sigma_3}{\sigma_1 \mid \Omega \mid \Gamma \vdash (\mathbf{def}\ k = e; \mathcal{S}) \rightsquigarrow \mathbf{def}\ k = e; \mathcal{S}_d \mid \mathcal{S}_s \mid \sigma_3}$$

$$\textsc{E-ST-MACRO}\ \frac{\sigma_1 \mid \Omega \vdash_c e \rightsquigarrow v \mid \sigma_2 \qquad \sigma_2 \mid \Omega; m \mapsto v \mid \Gamma, m : \tau \vdash \mathcal{S} \rightsquigarrow \mathcal{S}_d \mid \mathcal{S}_s \mid \sigma_3}{\sigma_1 \mid \Omega \mid \Gamma \vdash (\mathbf{def}^{\downarrow}\ m = e; \mathcal{S}) \rightsquigarrow \mathcal{S}_d \mid \mathbf{def}^{\downarrow}\ m = v; \mathcal{S}_s \mid \sigma_3}$$

$$\textsc{E-ST-MODULE}\ \frac{\sigma_1 \mid \Omega \mid \Gamma \vdash \mathcal{M} \rightsquigarrow \mathcal{L} \mid \mathcal{M}_d \mid \mathcal{M}_s \mid \sigma_2 \qquad \sigma_2 \mid \Omega; M_s \mapsto (\lceil \mathcal{M}_s \rceil^{M_d}) \mid \Gamma, M : \Delta \vdash \mathcal{S} \rightsquigarrow \mathcal{S}_d \mid \mathcal{S}_s \mid \sigma_3}{\sigma_1 \mid \Omega \mid \Gamma \vdash (\mathbf{mod}\ M : \Delta = \mathcal{M}; \mathcal{S}) \rightsquigarrow \mathcal{L}; \mathbf{mod}\ M_d = \mathcal{M}_d; \mathcal{S}_d \mid \mathbf{mod}\ M_s = \lceil \mathcal{M}_s \rceil^{M_d}; \mathcal{S}_s \mid \sigma_3}$$

Fig. 20. Elaboration of modules and structures

$\boxed{\sigma_1 \mid \Omega \vdash \mathcal{F}_s \bullet \mathcal{M}_d \mid \mathcal{M}_s \Longrightarrow \mathcal{L} \mid \mathcal{M}_d \mid \mathcal{M}_s \mid \sigma_2}$ *(Functor applications (excerpt))*

$$\textsc{E-APP-FUNCTOR}\ \frac{\sigma_1 \mid \Omega; C; M_s \mapsto \lceil \mathcal{M}_s \rceil^{M_d} \mid \Gamma, M : \Delta_1 \vdash \mathcal{N} \rightsquigarrow \mathcal{L} \mid \mathcal{N}_d \mid \mathcal{N}_s \mid \sigma_2}{\sigma_1 \mid \Omega \vdash [C]\langle \Gamma, \mathbf{functor}(M : \Delta).\ \mathcal{N} \rangle \bullet \mathcal{M}_d \mid \mathcal{M}_s \Longrightarrow \mathbf{mod}\ M_d = \mathcal{M}_d; \mathcal{L} \mid \mathcal{N}_d \mid [C; M_s \mapsto \mathcal{M}_s]\mathcal{N}_s \mid \sigma_2}$$

$$\textsc{E-APP-MVAR1}\ \frac{M_s \mapsto \mathcal{F}_s \in \Omega \qquad \sigma_1 \mid \Omega \vdash \mathcal{F}_s \bullet \mathcal{M}_d \mid \mathcal{M}_s \Longrightarrow \mathcal{L} \mid \mathcal{N}_d \mid \mathcal{N}_s \mid \sigma_2}{\sigma_1 \mid \Omega \vdash [C]M_s \bullet \mathcal{M}_d \mid \mathcal{M}_s \Longrightarrow \mathcal{L} \mid \mathcal{N}_d \mid \mathcal{N}_s \mid \sigma_2} \qquad \textsc{E-APP-MVAR2}\ \frac{M_s \mapsto \mathcal{F}_s \in C \qquad \sigma_1 \mid \Omega \vdash [C]\mathcal{F}_s \bullet \mathcal{M}_d \mid \mathcal{M}_s \Longrightarrow \mathcal{L} \mid \mathcal{N}_d \mid \mathcal{N}_s \mid \sigma_2}{\sigma_1 \mid \Omega \vdash [C]M_s \bullet \mathcal{M}_d \mid \mathcal{M}_s \Longrightarrow \mathcal{L} \mid \mathcal{N}_d \mid \mathcal{N}_s \mid \sigma_2}$$

Fig. 21. Elaboration of functor applications

5.2 Elaboration of Modules

Fig. 20 presents the elaboration rules of modules. The judgment $\sigma_1 \mid \Omega \mid \Gamma \vdash \mathcal{M} \rightsquigarrow \mathcal{L} \mid \mathcal{M}_d \mid \mathcal{M}_s \mid \sigma_2$ reads: under the heap σ_1, the evaluation context Ω, and the type context Γ, the module $\mathcal{M}$ elaborates to a module list $\mathcal{L}$, a dynamic module $\mathcal{M}_d$, and a static module $\mathcal{M}_s$, and updates the heap to σ_2.

5.2.1 Module Splitting. We start by explaining the elaboration rules for modules. Rule E-M-STRUCT simply elaborates the structure body. The module list $\mathcal{L}$ in this case, and in all the rules other than rule E-M-APP, is always empty. For a module variable (rule E-M-MVAR) or a qualified module

variable (rule E-M-PMVAR), we separate a variable M into two corresponding module variables M_d and M_s. Recall that paths are a subset of modules, which can be elaborated via module elaboration.

For functors (rule E-M-FUNCTOR), since they are values, we split a functor into an empty dynamic module, while the static module captures the current type context Γ and the original functor.

Rule E-M-APP applies a functor to a module argument. At this point, since the functor is applied, we would like the top-level splices inside the functor to be evaluated. The elaboration achieves that in a few steps. First, it elaborates the functor module $\mathcal{F}$ into a module list $\mathcal{L}_1$, a dynamic module $\mathcal{F}_d$, and a static module $\mathcal{F}_s$. We can safely ignore $\mathcal{F}_d$, as in the case when a module returns a functor, its dynamic part is always empty. Similarly, it elaborates the argument module $\mathcal{M}$ into a module list $\mathcal{L}_2$, a dynamic module $\mathcal{M}_d$, and a static module $\mathcal{M}_s$. Then, the application judgment given in Fig. 21 (explained below) applies the functor to the argument, generating another module list $\mathcal{L}_3$, a dynamic module $\mathcal{N}_d$, and a static module $\mathcal{N}_s$. Finally, the rule returns the concatenation of all module lists $\mathcal{L}_1; \mathcal{L}_2; \mathcal{L}_3$, and the final dynamic module $\mathcal{N}_d$ and static module $\mathcal{N}_s$.

5.2.2 Functor Applications. Fig. 21 presents the application rules used in rule E-M-APP. The judgment $\sigma_1 \mid \Omega \vdash \mathcal{F}_s \bullet \mathcal{M}_d \mid \mathcal{M}_s \Longrightarrow \mathcal{L} \mid \mathcal{M}_d \mid \mathcal{M}_s \mid \sigma_2$ reads: under the heap σ_1 and the evaluation context Ω, applying the functor $\mathcal{F}_s$ to an argument with dynamic part $\mathcal{M}_d$ and static part $\mathcal{M}_s$ returns a module list $\mathcal{L}$, a dynamic module $\mathcal{M}_d$, a static module $\mathcal{M}_s$, and updates the heap to σ_2. Since $\mathcal{F}_s$ has a functor type, it must be either a functor, a module variable, or a qualified module variable. Fig. 21 presents the rules for the first two cases; the case with qualified module variables works similarly.

Rule E-APP-FUNCTOR is the most interesting rule. In this case, we know that we are applying a functor to an argument, and as such the top-level splices inside the functor should get evaluated. We therefore split the body of the functor $\mathcal{N}$, under the captured type context Γ extended with a type for the argument M; we assume M is fresh thanks to alpha-renaming. There are a few notable things. We first elaborate M into two module variables M_s and M_d.[6] We extend the evaluation environment Ω with the module list $\mathcal{C}$ as well as the binding $M_s \mapsto \lceil \mathcal{M}_s \rceil^{M_d}$, since the static part of the module argument M_s may be required for evaluating top-level splices inside $\mathcal{N}$. Notably, since the static part of a module can quote names from the dynamic part, we use the notation $\lceil \cdot \rceil^{M_d}$ to prefix M_d to all variables that are defined in M_d.

Moreover, we need to insert the module argument into the elaboration result. Note that the dynamic part $\mathcal{M}_d$ and the static part $\mathcal{M}_s$ are treated differently. Specifically, here is where the module list $\mathcal{L}$ extends: we put $\mathcal{M}_d$ to be before $\mathcal{L}$. As we will see, the insertion order is important for ensuring that splitting preserves the semantics of the program, which we will discuss in §6.3. On the other hand, the static module $M_s \mapsto \mathcal{M}_s$ is bound locally.

Rule E-APP-MVAR1 and E-APP-MVAR2 deal with the cases where the module being applied is a module variable. In such cases, the module variable may appear inside the evaluation environment Ω, or in the locally bound modules $\mathcal{C}$, and the rule recurses. In the former case, the locally bound module variables $\mathcal{C}$ are not useful anymore and can be ignored.

5.3 Elaboration of Structures

Lastly, Fig. 20 also presents the elaboration rules for structures. The judgment $\sigma_1 \mid \Omega \mid \Gamma \vdash \mathcal{S} \rightsquigarrow \mathcal{S}_d \mid \mathcal{S}_s \mid \sigma_2$ elaborates a structure $\mathcal{S}$ into a dynamic structure $\mathcal{S}_d$ and a static structure $\mathcal{S}_s$. An empty structure produces an empty dynamic and an empty static structure (rule E-ST-EMPTY).

Moreover, splitting a structure puts let definitions into the dynamic part (rule E-ST-DEF), and macro definitions into the static part (rule E-ST-MACRO). Rule E-ST-MODULE uses the elaboration rule of modules. Interestingly, the rule also inserts the module list $\mathcal{L}$ produced by the submodule

[6] When the arguments are module variables or qualified module variables, we can use those variables directly; here we leave the rule uniform for simplicity.

into the containing structure, returning the result dynamic submodule $\mathcal{L}$; **mod** $M_d = \mathcal{M}_d; \mathcal{S}_d$. This is valid syntax, as we are returning a structure. Again, the order in which $\mathcal{L}$ is inserted is important for preserving the semantics.

5.4 Example

To demonstrate how elaboration works, we apply the elaboration rules to the example in §2.3. We omit the type definition, and assume elaboration works on recursive bindings and if-expressions. Specifically, consider elaborating the functor application $\mathcal{F}\ \mathcal{M}$, where

$$
\begin{array}{l}
\mathcal{F} = \textbf{functor}(M : \Delta).\ \mathcal{N} \\
\Delta = \textbf{sig}\ (one : \mathsf{Code\ Int}; mul : \mathsf{Code\ Int} \to \mathsf{Code\ Int} \to \mathsf{Code\ Int}; show : \mathsf{Int} \to \mathsf{String})\ \textbf{end} \\
\mathcal{N} = \textbf{struct}\ (\ \textbf{def}\ fpower = \lambda n : \mathsf{Int}.\ \lambda x : \mathsf{Int}.\textbf{if}\ n = 0\ \textbf{then}\ \$(M.one)\ \textbf{else}\ \$(M.mul\ \langle x \rangle\ \langle fpower\ (n-1)\ x \rangle); \\
\qquad\qquad \textbf{def}\ show\_power5 = \lambda x : \mathsf{Int}.\ M.show\ (fpower\ 5\ x); \\
\qquad\qquad \textbf{def}^{\downarrow}\ showOne = \lambda x : \mathsf{Unit}.\ \langle M.show\ \$(M.one) \rangle\ )\ \textbf{end} \\
\mathcal{M} = \textbf{struct}\ (\textbf{def}^{\downarrow}\ one = \langle 1 \rangle; \textbf{def}^{\downarrow}\ mul = \lambda x : \mathsf{Code\ Int}.\ \lambda y : \mathsf{Code\ Int}.\ \langle \$(x) * \$(y) \rangle; \\
\qquad\qquad \textbf{def}\ show = string\_of\_int)\ \textbf{end}
\end{array}
$$

Elaborating functor applications. We first elaborate the functor, which produces an empty dynamic module, and a static module that captures the typing context with the functor:

$$\textcircled{A}\ \frac{}{\sigma \mid \Omega \mid \Gamma \vdash \mathcal{F} \rightsquigarrow \bullet \mid \textbf{struct}\ \bullet\ \textbf{end} \mid \langle \Gamma, \textbf{functor}(M : \Delta).\ \mathcal{N} \rangle \mid \sigma}\ \textsc{e-m-functor}$$

We then elaborate the module argument. Since the argument is a structure with a list of items, we simply separate the let definition from the macro definitions; we omit the detailed derivations.

$$
\begin{array}{l}
\mathcal{M}_d = (\textbf{struct}\ \textbf{def}\ show = string\_of\_int\ \textbf{end}) \\
\mathcal{M}_s = (\textbf{struct}\ \textbf{def}^{\downarrow}\ one = \langle 1 \rangle; \textbf{def}^{\downarrow}\ mul = \lambda x : \mathsf{Code\ Int}.\ \lambda y : \mathsf{Code\ Int}.\ \langle \$(x) * \$(y) \rangle\ \textbf{end})
\end{array}
$$

$$\textcircled{B}\ \frac{\cdots\cdots}{\sigma \mid \Omega \mid \Gamma \vdash \mathcal{M} \rightsquigarrow \bullet \mid \mathcal{M}_d \mid \mathcal{M}_s \mid \sigma}\ \textsc{e-m-struct}$$

Combining these two steps in the rule for functor applications (rule E-M-APP), we now need to derive $\sigma \mid \Omega \vdash \langle \Gamma, \textbf{functor}(M : \Delta).\ \mathcal{N} \rangle \bullet \mathcal{M}_d \mid \mathcal{M}_s$ using rule E-APP-FUNCTOR. At this point, rule E-APP-FUNCTOR puts $M_s \mapsto \mathcal{M}_s$ into the evaluation environment, making macros from $\mathcal{M}_s$ available when elaborating the functor body $\mathcal{N}$. Moreover, note that $\mathcal{N}_s$ keeps track of the binding $M_s \mapsto \mathcal{M}_s$.

$$
\begin{array}{l}
\mathcal{N}_d = \textbf{struct}\ (\ \textbf{def}\ fpower = \lambda n : \mathsf{Int}.\ \lambda x : \mathsf{Int}.\textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ x * (fpower\ (n-1)\ x); \\
\qquad\qquad \textbf{def}\ show\_power5 = \lambda x : \mathsf{Int}.\ M_d.show\ (fpower\ 5\ x)\ )\ \textbf{end} \\
\mathcal{N}_s = \textbf{struct}\ (\ \textbf{def}^{\downarrow}\ showOne = \lambda x : \mathsf{Unit}.\ \langle M_d.show\ \$(M_s.one) \rangle\ )\ \textbf{end}
\end{array}
$$

$$\textcircled{C}\ \frac{\dfrac{}{\sigma \mid \Omega; M_s \mapsto \mathcal{M}_s \mid \Gamma, M : \Delta \vdash \mathcal{N} \rightsquigarrow \bullet \mid \mathcal{N}_d \mid \mathcal{N}_s \mid \sigma}\ \textsc{e-m-struct}}{\sigma \mid \Omega \vdash \langle \Gamma, \textbf{functor}(M : \Delta).\ \mathcal{N} \rangle \bullet \mathcal{M}_d \mid \mathcal{M}_s \Longrightarrow \textbf{mod}\ M_d = \mathcal{M}_d \mid \mathcal{N}_d \mid [M_s \mapsto \mathcal{M}_s]\mathcal{N}_s \mid \sigma}\ \textsc{e-app-functor}$$

Putting the three derivations together, we have:

$$\frac{\textcircled{A} \quad \textcircled{B} \quad \textcircled{C}}{\sigma \mid \Omega \mid \Gamma \vdash \mathcal{F}\ \mathcal{M} \rightsquigarrow \textbf{mod}\ M_d = \mathcal{M}_d \mid \mathcal{N}_d \mid [M_s \mapsto \mathcal{M}_s]\mathcal{N}_s \mid \sigma}\ \textsc{e-m-app}$$

Compile-time code generation. In the derivation for ©, since M_s is available, top-level splices are evaluated when elaborating $\mathcal{N}$ into $\mathcal{N}_d$. As an example, we look at the **else** part of *fpower*, which applies the code generation rule E-CODEGEN:

$$\frac{\sigma \mid \Omega; M_s \mapsto \mathcal{M}_s \vdash_{\mathsf{s}} M.mul\ \langle x \rangle\ \langle fpower\ (n-1)\ x \rangle \rightsquigarrow M_s.mul\ \langle x \rangle\ \langle fpower\ (n-1)\ x \rangle \mid \sigma \quad \sigma \mid \Omega; M_s \mapsto \mathcal{M}_s \mid M_s.mul\ \langle x \rangle\ \langle fpower\ (n-1)\ x \rangle \xrightarrow{0}{}^{*} \langle x * (fpower\ (n-1)\ x) \rangle \mid \sigma}{\sigma \mid \Omega; M_s \mapsto \mathcal{M}_s \vdash_{\mathsf{c}} \$(M.mul\ \langle x \rangle\ \langle fpower\ (n-1)\ x \rangle) \rightsquigarrow x * (fpower\ (n-1)\ x) \mid \sigma}\ \textsc{e-codeGen}$$

Here, the elaboration part does nothing. The evaluation shows how quotations and splices work together at compile-time. More concretely, we first have

$$\sigma \mid \Omega; M_s \mapsto \mathcal{M}_s \mid M_s.mul\ \langle x \rangle\ \langle fpower\ (n-1)\ x \rangle \xrightarrow{0}{}^{*}\ \langle \$(\langle x \rangle) * \$(\langle fpower\ (n-1)\ x \rangle) \rangle \mid \sigma$$

At this point, we evaluate inside quotations, and rule EV-SPLICECODE cancels out splice-quotation pairs at level 1:

$$\dfrac{\dfrac{\dfrac{}{\sigma \mid \Omega; M_s \mapsto \mathcal{M}_s \mid \$(\langle x \rangle) \xrightarrow{1}{}^{*}\ x \mid \sigma}\ \textsc{ev-spliceCode} \quad \cdots\cdots}{\sigma \mid \Omega; M_s \mapsto \mathcal{M}_s \mid \$(\langle x \rangle) * \$(\langle fpower\ (n-1)\ x \rangle) \xrightarrow{1}{}^{*}\ x * (fpower\ (n-1)\ x) \mid \sigma}\ \textsc{ev-app1,ev-app2}}{\sigma \mid \Omega; M_s \mapsto \mathcal{M}_s \mid \langle \$(\langle x \rangle) * \$(\langle fpower\ (n-1)\ x \rangle) \rangle \xrightarrow{0}{}^{*}\ \langle x * (fpower\ (n-1)\ x) \rangle \mid \sigma}\ \textsc{ev-quote}$$

The evaluation result is then given to rule E-CODEGEN, which removes the quotation.

6 Metatheory

In this section, we discuss desirable properties for the elaboration process, including elaboration soundness (§6.1), phase distinction (§6.2), and that elaboration preserves semantics (§6.3).

6.1 Elaboration Soundness and Phase Distinction

The following theorem establishes elaboration soundness for modules, structures, and expressions.

Theorem 6.1 (Elaboration Soundness). *Given* σ_1 ok *and* $\bullet \vdash \Omega_s$,
(modules) *if* $\bullet \vdash \mathcal{M} : \Delta$, *and* $\sigma_1 \mid \Omega_s \mid \bullet \vdash \mathcal{M} \rightsquigarrow \mathcal{L} \mid \mathcal{M}_d \mid \mathcal{M}_s \mid \sigma_2$,
then (1) $\bullet \mid \bullet \vdash \mathcal{L} : \phi_d$, *(2)* $\bullet \mid \phi_d \vdash \mathcal{M}_d : \Delta_d$, *(3)* $\Delta_d <: (\Delta)_d$, *and (4)* $\phi_d, \Delta_d \vdash \mathcal{M}_s : (\Delta)_s$.
(structures) *if* $\bullet \vdash \mathcal{S} : \phi$, *and* $\sigma_1 \mid \Omega_s \mid \bullet \vdash \mathcal{S} \rightsquigarrow \mathcal{S}_d \mid \mathcal{S}_s \mid \sigma_2$,
then (1) $\bullet \mid \bullet \vdash \mathcal{S}_d : \phi_d$, *(2)* $\phi_d <: (\phi)_d$, *and (3)* $\phi_d \vdash \mathcal{S}_s : (\phi)_s$.
(expressions) *if* $\bullet \vdash^n e : \tau$, *and* $\sigma \mid \Omega_s \vdash_\star e \rightsquigarrow e \mid \sigma'$, *then* $\bullet \mid \bullet \vdash^n e : \tau$.

There are a few notable things. First, notice that the elaboration results are type-checked under an *empty* heap. This property is important as, in practice, we may compile a program in one environment, and then run the compiled program in a different environment. It is consequently important that the compiled program does not refer to any values from the compile-time heap. Intuitively, this holds as any locations generated at compile-time will be level -1, which cannot be referred to in the compiled program. This suggests that compile-time heaps (both σ_1 and σ_2) can be safely discarded after compilation.

Second, since elaboration of structures insert the module list $\mathcal{L}$ into the elaboration result, in the elaboration soundness of modules and structures, the type Δ_d or ϕ_d can potentially contain more items than $(\Delta)_d$ or $(\phi)_d$; we use $<:$ to denote such a relation.[7]

Moreover, since ϕ_d is a list of item types, we put it inside a context to mean that we add all its items to the context. Similarly, we put Δ_d in a context to mean that since we know it has shape **sig** ϕ_d **end**, we put ϕ_d in the context. For example, in elaboration soundness for modules (2), we add ϕ_d to the context to type-check $\mathcal{M}_d$. which has type Δ_d. Similarly, we put both ϕ_d and Δ_d in the context to type-check $\mathcal{M}_s$, which has type $(\Delta)_s$. Note that while $\mathcal{M}_d$ depends on $\mathcal{L}$, $\mathcal{M}_s$ depends on both $\mathcal{L}$ and $\mathcal{M}_d$.

6.2 Phase Distinction

Xie et al. [2023] proved a *phase distinction* theorem for the core calculus for MacoCaml, which says that compile-time only computations are not needed for run-time evaluation. Specifically, with the erasure notation $[\![\cdot]\!]$ that erases the static part (i.e. macros) of a module, we have:

[7] In a system with module subtyping, we could type-check e.g. $\mathcal{M}_d$ with $(\Delta)_d$; see more discussion in §7.

Theorem 6.2 (Phase Distinction in Xie et al. [2023]). *Given* σ ok, $\sigma \mid \Gamma \vdash \Omega$ *and* $\sigma \mid \Gamma \vdash \mathcal{M} : \Delta$, *if* $\sigma \mid \Omega \mid \mathcal{M} \longrightarrow \mathcal{M}' \mid \sigma'$, *then* $\sigma \mid [\![\Omega]\!] \mid [\![\mathcal{M}]\!] \longrightarrow [\![\mathcal{M}']\!] \mid \sigma'$,

Therefore, after elaboration, the macros can be erased before evaluation.

With our formalism, this property is evident. In $maco^{F}_{core}$, since each module is explicitly divided into a dynamic module and a static module, no additional erasure is needed. Moreover, the elaboration soundness of modules (Theorem 6.1) makes clear that the dynamic part does not depend on the static part at all. In other words, our formalism enjoys phase distinction by construction.

6.3 Elaboration Preserves Semantics

The elaboration process produces an additional module list $\mathcal{L}$ that is inserted into a structure (rule E-ST-MODULE). The module list consists of lifted module arguments (rule E-APP-FUNCTOR). One may wonder whether lifting and inserting those dynamic modules could change the semantics of a program, especially since our calculus includes side-effects.

We prove that elaboration preserves the semantics — or, more precisely, the observable side-effects — of expressions. To this end, we first consider a direct operational semantics for the source calculus. Specifically, if the source program has no macros, splices, or quotations, we can define a direct operational semantics denoted as $\mathcal{M}_1 \longrightarrow \mathcal{M}_2$, which is similar to the semantics of the dynamic modules in the core calculus, but extended with evaluation rules for functor applications $\mathcal{M}_1\ \mathcal{M}_2$, which first evaluate $\mathcal{M}_1$, and then $\mathcal{M}_2$, and finally reduce the application:[8]

$$\sigma \mid \Omega_d \mid (\textbf{functor}\ (M : \Delta).\mathcal{N})\ \mathcal{M}_v \longrightarrow \mathcal{N}[M \mapsto \mathcal{M}_v] \mid \sigma$$

If we are given a source program that has no macros, splices, or quotations, we can now evaluate it in two different ways: (1) we can evaluate it using the direct operational semantics, or (2) we can first elaborate it into the core calculus, which involves module splitting and insertion, and then we evaluate the elaborated result according to the semantics of the core calculus. Ideally, if we lift and insert module bindings in the order they would have been evaluated by the above semantics, these two evaluation strategies should produce the same side-effects. To make this more precise, we assume that rule EV-REF creates locations (l) in a deterministic order. This allows for a direct comparison of the order in which ref e expressions are evaluated. Then we prove:

Theorem 6.3 (Elaboration Preserves Semantics). *Given* $\sigma \mid \bullet \mid \mathcal{M} \longrightarrow^{*} \mathcal{M}_v \mid \sigma'$, *if* $\bullet \mid \bullet \mid \Gamma \vdash \mathcal{M} \rightsquigarrow \mathcal{L} \mid \mathcal{M}_d \mid \mathcal{M}_s \mid \sigma_1$, *and* $\sigma \mid \bullet \mid \mathcal{L} \longrightarrow^{*} \mathcal{L}_v \mid \sigma_2$, *and* $\sigma_2 \mid \mathcal{L}_v \mid \mathcal{M}_d \longrightarrow^{*} \mathcal{M}_{dv} \mid \sigma''$, *then* $\sigma' = \sigma''$.

Namely, whether we first evaluate the module according to the direct operational semantics, or first elaborate it to the core and evaluate first the generated module list and then the dynamic module (where we write $\mathcal{L}_v$ in an evaluation context to mean that the bindings are put in the environment), we get the same resulting heaps σ' and σ''. This lemma applies to source programs without macros or staging, as the semantics of programs with macros or staging are *defined* in terms of elaboration. In the above lemma, we could also relate $\mathcal{M}_v$ to $\mathcal{M}_s$, $\mathcal{L}_v$, and $\mathcal{M}_{dv}$, but here we are mainly concerned about the produced side-effects.

7 Extensions

This work focuses on the combination of functors and compile-time code generation. A full module system includes additional features, and in this section we discuss the most relevant ones.

[8] This rule provides a substitution based semantics for functor applications, assuming path resolution during substitution. An alternative way is to add $M \mapsto \mathcal{M}_v$ to the evaluation environment Ω_d to evaluate $\mathcal{N}$, which would require the operational semantics to be changed to also return an updated evaluation environment.

Module imports. MacoCaml supports module imports as well as *shifted* module imports [Flatt 2002], where modules can be imported at compile-time. This allows writing interesting programs which manipulate and share references across splices. Consider a module `Term`

```
(* term.ml *)
struct let id = ref 0 end
```

Importing the module at compile-time shifts all its definitions by -1, which allows us to use `id` when defining macros:

```
(* program.ml *)
module ~N = Term            (* import Term at compile-time as N *)
macro incr () = N.id := !N.id + 1          (* N.id at level -1 *)
```

Whenever `incr` is called inside a top-level splice, the reference `id` will get incremented.

Our formalism has been set up to be compatible with module imports, with the heaps threaded through elaboration. Supporting unshifted module imports is relatively straightforward, where we import the compiled form of a module $\mathcal{L} \mid \mathcal{M}_d \mid \mathcal{M}_s$. In our current setting, the compiled module includes the inserted modules to preserve type soundness, while those modules should not be directly accessible by the user. Importing shifted modules is subtler. If we import a module at compile-time, its corresponding $\mathcal{L}$ and $\mathcal{M}_d$ become static components (at level -1), while $\mathcal{M}_s$ will be at level -2. Moreover, the module importing the shifted module may itself be imported at compile-time by another module, putting $\mathcal{M}_s$ at level -3. In such a case, splitting needs to return n modules where n is the number of levels we have. We leave such a formalism to future work.

Module subtyping. So far we have focused on the situation where every module exports all the definitions in its body, including additionally inserted modules. In systems supporting *module subtyping* [Mitchell and Harper 1988], we can choose to selectively export definitions, and also not to expose the inserted modules, by ascribing a signature to a module that lists only a subset of the module's components. The problem with module subtyping in the context of MacoCaml and *maco*^{F} is that generated code may refer to names that are not in scope, having been hidden by ascription. As a simple example, in the following fragment, the call to `M.m` on the last line generates a reference to `secret`, which is not accessible outside the definition of `M`:

```
module M : sig macro m : unit → int expr end
  = struct let secret = 1 ;; macro m () = <<secret>> end
$(M.m ())   (* M.secret not in scope *)
```

To deal with this extrusion, the MacoCaml compiler (but not the published formalism) implements *path closures* that transform each macro that quotes module-local definitions so that the quoted names are exported in a module that is inserted alongside the macro. Following the path closure transformation, `M` is transformed to export an additional module `Closure_1`, and the call to `M.m` generates the name `M.Closure_1.secret`. We plan to formalize path closures in the future.

We also anticipate that module subtyping will be helpful for simplifying some aspects of our metatheory; in particular, it will allow us to state that the dynamic part of a module has type $(\Delta)_d$ rather than $\Delta_d <: (\Delta)_d$.

8 Integration into OCaml

We consider some practical issues related to the compilation of a programs in a *maco*^{F}-style language — that is, a language that supports both module functors and macros — and to the incorporation of our design into the existing OCaml toolchain.

8.1 Compilation of Functors: Parameterization vs Instantiation

The OCaml compiler, like some other ML-family implementations such as SML/NJ, compiles the definitions of functors independently of their applications [Appel and MacQueen 1994; Leroy 1994]. In other words, functors are compiled like functions: a functor like `R` in Fig. 1 (§2.2) is compiled and represented in the same way as a function from a record with three fields (`one`, `mul`, `show`) to a record with two fields (`rpower`, `show_power5`). With this *parametric* approach, values are represented uniformly at run time, and the compilation of the code that acts upon those values cannot depend on the types of the values: the `rpower` function in the body of `R` is compiled to a single piece of code that is re-used unchanged in every application of `R`.

The parametric approach enjoys several advantages over an *instantiation*-based model, where the body of the functor is recompiled every time the functor is applied. These advantages include small code and strong support for separate compilation, leading to short compilation times. However, the parametric approach also suffers from some drawbacks: in particular, it compiles functions produced by functor applications to less efficient code than equivalent functions that were written without functor abstraction. For example, in the module `RInt`, the indirect call `M.mul` to a function passed as an argument is a good deal less efficient than a direct use of the built-in operator `+`, and the code for `rpower` is not specialized to the `int` type supplied in the functor application[9]. For reasons such as these, some aggressively-optimising compilers follow an instantiation-based approach: examples include Mlton (which dubs it *defunctorization* [Weeks 2006]) and MLKit (which calls it *static interpretation* [Elsman 1999]).

8.2 Compilation of Functors Involving Macros

With the addition of macros to the OCaml language, the parametric compilation scheme is no longer sufficient. For example, in the functor `F` in Fig. 2, the code generated for `fpower` depends on the parameters `M.one` and `M.mul`: clearly, it is not possible to compile `fpower` to a single piece of code. Switching to an *instantiation*-based compilation model for functors resolves the issue, and it is not difficult to show that such a model is also correct for existing OCaml code, since it amounts to inlining each functor at every application type.

The instantiation scheme corresponds to the approach to the compilation of functors set out in §5, which eliminates functor applications by elaboration (Fig. 21). However, switching wholesale to instantiation-based compilation for functors would not be acceptable to the OCaml community, since it would lead to substantial increases in code size and compilation times.

Fortunately, the addition of macros does not require changing the way that most functors are compiled. For functors that make no use of macros, clearly no change is needed: the existing scheme is sufficient. Functors that additionally export macros do not necessarily need to use instantiation, either: it is only those functors that import macros and use them in functions defined within the bodies of the functors that require a switch to the instantiation-based model.

In other words, we treat the elaboration in §5 as a *specification* of compilation. For functors that import macros the compiler can follow the elaboration scheme directly; for functors that do not import macros, the compiler can continue to use the parametric approach, which is semantically equivalent in such cases.

Where code size remains a concern, we might even further disaggregate functor bodies according to the dependencies of their components, using parameterization-based compilation for those parts of the body that do not (directly or indirectly) depend on macros supplied by the argument, and the instantiation-based scheme for those parts that do.

[9]For `int` the lack of specialization does not matter much in OCaml; for other types the lack of specialization might lead to missed opportunities for unboxing optimisations, for example.

9 Related Work

The work most closely related to ours, Xie et al. [2023], forms the foundation for our development.

9.1 Combining Modules with Staged Definitions

Suwa and Igarashi [2024] present a staging calculus with a ML-style module system, but without mutable references. With their design, a definition declared at level n is elaborated into an ordinary definition (i.e. at level 0) but with its bound expression wrapped inside n quotations. While their elaboration produces well-typed programs, it makes all definitions to be used in a call-by-name manner. As a result, certain desirable properties such as Theorem 6.3 do not hold, and it is unclear how their design applies to OCaml.

9.2 Staging Modules

Another line of work connecting functors with staging [Inoue et al. 2016; Sato and Kameyama 2021; Sato et al. 2020; Watanabe and Kameyama 2018] addresses the challenge of supporting quotations that generate modules and functors, which is complementary to our design for modules and functors that contain macros.

9.3 Typed Quotation-Based Macros

The typed quotations that macros in $maco^F$ use to construct code were initially designed for run-time code generation in MetaML [Taha 1999], and derive from temporal logic [Davies 1996]; our Code τ type constructor corresponds to the $\bigcirc$ operator in Davies' $\lambda^{\bigcirc}$, which models *open* future-stage terms. Our code quotations inherit various pleasant guarantees from MetaML: for example, binding in quotations is lexically-scoped and avoids inadvertent capture. A separate body of work on staged languages (e.g. by Jang et al. [2022] and Murase et al. [2023]) derive from modal logic [Davies and Pfenning 1996], where the $\Box$ operator models *closed* future-stage terms, via contextual modal logic [Nanevski et al. 2008], where the contextual operator $[\Psi]$ models future-stage terms that draw variables from a context Ψ.

MetaML's quotations have been adapted for compile-time code generation a number of times: MacroML [Ganz et al. 2001] adapts quotations to define staged macros that are translated to MetaML programs by elaboration; Staged Notational Definitions [Taha and Johann 2003] extend MacroML to better support alpha-conversion; Stucki et al. [2018] use typed code quotations for both compile-time and run-time code generation in a single system.

None of these works examines the interplay between functors and macros. In contrast, Xie et al. [2022] formally study the interaction between compile-time staging and type class constraints (which correspond to a kind of implicit functor), and propose a new construct, *staged type class constraints*, that resolves an elaboration failure in the implementation of Typed Template Haskell. The particular problem that staged type class constraints address does not arise in $maco^F$, which does not support implicit functors, and Xie et al. are principally interested in typing and elaboration, rather than the evaluation of staged programs that is a key focus of the current paper.

9.4 Multi-Stage Programming with Functors

Among users of MetaML-family languages there is an established pattern of combining functors with quotation-based run-time code generation to build parameterized libraries in which parameterization does not introduce any overhead into the generated code. Type classes in Haskell play a similar role to ML functors — in fact, there is a well-known correspondence between the two [Wehr and Chakravarty 2008] — so we also note applications of type classes with quotation here.

Carette and Kiselyov [2005] showed that the combination of functor abstraction and quotation-based staging allow the definition of *families* of algorithms without any abstraction overhead, taking Gaussian elimination as a representative example. Carette et al. [2011] makes similar use of functors to support configuration of the code generator in a generative geometric kernel. The seminal work by Carette et al. [2009], which introduced the *tagless final* approach to defining domain-specific languages (DSLs), uses functors to abstract over the interpretations of a DSL; several of the interpretations make use of quotation-based staging to compile terms in the DSL to efficient code. The particular combination of functor abstraction and code generation used in the tagless final approach has been applied to generate code for a wide variety of domains; for example, Suzuki et al. [2016] use it to efficiently embed a relational query language into OCaml and Tokuda and Kameyama [2023] use it to generate efficient algorithms for post-quantum cryptography.

Yallop [2017] uses *modular implicits* [White et al. 2014] (a kind of first-class functor that supports implicit instantiation) to structure the code of a generative generic programming library. Yallop et al. [2018b] uses type classes with Typed Template Haskell's code quotations to provide a modular library for optimization of terms in various algebras.

In contrast to these works, most existing OCaml libraries[10] for compile-time code generation do not make use of functors, because they are based on preprocessors such as Camlp5 [de Rauglaudre 2007] (which transforms concrete syntax) and ppx [White 2013] (which transforms abstract syntax), which are not aware of binding scopes, types, modules, etc., and so do not take account of the context in which invocations of code-generating functions appear. Two notable exceptions are the library operating system *Mirage* [Radanne et al. 2019], which uses functors and code generation to generate specialized operating system kernels by assembling typed components, and the *ctypes* foreign function library [Yallop et al. 2018a], which uses functors to abstract over the code-generating interpretations of a DSL that describes the interface between OCaml and C code.

We believe that the $maco^F$ design offers significant benefits for these existing applications. For compile-time code generation libraries, $maco^F$ offers several advantages over tools such as ppx: it is binding-aware, type-safe (in the sense that well-typed macros always produce well-typed code), and fully integrated into the module system of the host language, allowing code generators to be structured as typed libraries rather than external tools. For existing typed MetaML-family libraries, $maco^F$'s integration of macros with functors supports precise control over evaluation order (since functor applications trigger evaluation of splices in functor bodies), arbitrary dependencies between dynamic and static components of modules, and simplified run-time behaviour compared to systems such as MetaOCaml, which require either a runtime with an embedded compiler or code that is executed by being first printed to a file. In $maco^F$, macros enjoy the same integration into the module system as functions; they differ only in the phase at which they are available for execution.

9.5 Module Elaboration and Phase Splitting

Our formalisation follows a tradition of defining the semantics of module systems by elaboration into a core typed calculus; key previous examples include work by Harper and Stone [2000], which elaborates both the core and the module system of Standard ML into a module-based calculus, and by Rossberg et al. [2014], which shows that ML-family module systems can be elaborated into plain System Fω, an insight that Suwa and Igarashi [2024] use in elaborating their staged calculus.

9.5.1 Phase Splitting. Harper et al. [1989] introduced the notion of *phase splitting* for decomposing modules and functors into separate compile-time and run-time components, effectively showing that the ML module system did not need the full power of the dependent type systems that were

[10] Compile-time code generation is widely used in OCaml: at the time of writing there are over 200 libraries available via the OPAM package manager that depend on the ppx utility library `ppxlib`.

traditionally used to model it. Our work here is directly inspired by theirs, but the addition of macros extends the notion of *compile-time*, which originally referred to static analysis such as type-checking, to include full computation.

The key benefit of phase splitting, i.e. the disentangling of dependencies between static and dynamic components of modules, has been used to improve the precision of various analyses in tricky typing problems. Shao [1999] uses phase splitting to improve type sharing information across functor applications in an extension of Standard ML's module system to support higher-order functors. Dreyer et al. [2001] discuss module splitting in the context of recursive module declarations, noting that the lack of splitting in the Harper-Stone elaboration makes analysis of dependencies and signature equivalence unnecessarily difficult. As they note, the problem is resolved by Crary et al. [1999], who use phase splitting to show that dependencies of terms on types in recursive modules can be removed by elaboration; only dependencies of types on types are "essentially" recursive. The elaboration of ML into System Fω in the work by Rossberg et al. [2014] can also be viewed as a kind of splitting, since the target calculus clearly separates types from terms. The elaboration was subsequently used as the basis of 1ML, a redesign of ML in which the core and module languages are united and modules are first class [Rossberg 2018].

Besides these uses in formalisation and analysis, functor splitting is sometimes also used as an implementation technique, e.g. in the Flint/ML compiler [Shao 1997].

Finally, Thiemann [1999] notes an interesting equivalence between phase splitting for modules and specialization in partial evaluation. He equates functors in the work by Harper et al. [1989] with specialization-time functions whose applications should be reduced before run-time, and draws a connection between the application of phase splitting to recursive modules by Crary et al. [1999] and the computation of a specialization-time fixpoint.

10 Conclusion

We have presented the design and implementation of $maco^F$, a calculus that supports a novel combination of typed macros, quotation-based staging, and module functors, and given its type-preserving elaboration into $maco^F_{core}$. Our elaboration is structured around a splitting transformation inspired by the phase splitting transformation of Harper et al. [1989], but extended to support compile-time computation.

Future work. Our focus in this paper is on the combination of module functors as found in ML-family languages with compile-time quotation-based staging found in MacoCaml. As §3 says, type declarations are largely orthogonal to our focus on elaboration and evaluation, and so omitted. Extending the language to support type declarations will require consideration of the design question as to whether types, like values, should inhabit stage-indexed universes as proposed by Kovács [2022], or a single universe that is invariant across stages, as in MetaOCaml [Taha 1999].

Our calculus also lacks support for the module subtyping [Mitchell and Harper 1988] that is characteristic of ML-style module systems, and that introduces additional challenges when combined with quotation-based staging. In particular, since module subtyping supports hiding exported names during signature ascription, it is possible for macros in a module to expose **let**-bound names into a scope where (having been hidden by signature ascription) they are not visible (§7), compromising the subject reduction property. We anticipate extending our calculus to support the *path closures* described by Xie et al. [2023] to address this unsound interaction.

Acknowledgments

We thank the anonymous reviewers for helpful comments, and Dmitrij Szamozvancev for feedback on a draft. This work is funded by the Natural Sciences and Engineering Research Council of Canada, by Jane Street Capital, and by Ahrefs.

A Complete Rules

A.1 Source Typing

$\boxed{\Gamma \vdash \mathcal{M} : \Delta}$ *(Typing module)*

$$\textsc{M-STRUCT}\ \frac{\Gamma \vdash \mathcal{S} : \phi}{\Gamma \vdash \mathbf{struct}\ \mathcal{S}\ \mathbf{end} : \mathbf{sig}\ \phi\ \mathbf{end}} \qquad \textsc{M-MVAR}\ \frac{M : \Delta \in \Gamma}{\Gamma \vdash M : \Delta} \qquad \textsc{M-PMVAR}\ \frac{\Gamma \vdash p : \mathbf{sig}\ \phi\ \mathbf{end} \qquad M : \Delta \in \phi}{\Gamma \vdash p.M : \Delta}$$

$$\textsc{M-FUNCTOR}\ \frac{\Gamma, M : \Delta_1 \vdash \mathcal{N} : \Delta_2}{\Gamma \vdash \mathbf{functor}\ (M : \Delta_1).\mathcal{N} : \Delta_1 \to \Delta_2} \qquad \textsc{M-APP}\ \frac{\Gamma \vdash \mathcal{F} : \Delta_1 \to \Delta_2 \qquad \Gamma \vdash \mathcal{M} : \Delta_1}{\Gamma \vdash \mathcal{F}\ \mathcal{M} : \Delta_2}$$

$\boxed{\Gamma \vdash \mathcal{S} : \phi}$ *(Typing structure)*

$$\textsc{ST-EMPTY}\ \frac{}{\Gamma \vdash \bullet : \bullet} \qquad \textsc{ST-DEF}\ \frac{\Gamma \vdash^0 e : \tau \qquad \Gamma, k : \tau \vdash \mathcal{S} : \phi}{\Gamma \vdash (\mathbf{def}\ k = e; \mathcal{S}) : (k : \tau; \phi)} \qquad \textsc{ST-MACRO}\ \frac{\Gamma \vdash^{-1} e : \tau \qquad \Gamma, m : \tau \vdash \mathcal{S} : \phi}{\Gamma \vdash (\mathbf{def}^{\downarrow}\ m = e; \mathcal{S}) : (m : \tau; \phi)}$$

$$\textsc{ST-MODULE}\ \frac{\Gamma \vdash \mathcal{M} : \Delta \qquad \Gamma, M : \Delta \vdash \mathcal{S} : \phi}{\Gamma \vdash (\mathbf{mod}\ M : \Delta = \mathcal{M}; \mathcal{S}) : (M : \Delta; \phi)}$$

$\boxed{\Gamma \vdash^n e : \tau}$ *(Typing expression)*

$$\textsc{LIT}\ \frac{}{\Gamma \vdash^n i : \mathsf{Int}} \qquad \textsc{UNIT}\ \frac{}{\Gamma \vdash^n \mathsf{unit} : \mathsf{Unit}} \qquad \textsc{VAR}\ \frac{x : (\tau, n) \in \Gamma}{\Gamma \vdash^n x : \tau} \qquad \textsc{KVAR}\ \frac{k : \tau \in \Gamma}{\Gamma \vdash^0 k : \tau} \qquad \textsc{MACRO}\ \frac{m : \tau \in \Gamma}{\Gamma \vdash^{-1} m : \tau}$$

$$\textsc{PKVAR}\ \frac{\Gamma \vdash p : \mathbf{sig}\ \phi\ \mathbf{end} \qquad k : \tau \in \phi}{\Gamma \vdash^0 p.k : \tau} \qquad \textsc{PMACRO}\ \frac{\Gamma \vdash p : \mathbf{sig}\ \phi\ \mathbf{end} \qquad m : \tau \in \phi}{\Gamma \vdash^{-1} p.m : \tau}$$

$$\textsc{ABS}\ \frac{\Gamma, x : (\tau_1, n) \vdash^n e : \tau_2}{\Gamma \vdash^n \lambda x : \tau_1.\ e : \tau_1 \to \tau_2} \qquad \textsc{APP}\ \frac{\Gamma \vdash^n e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash^n e_2 : \tau_1}{\Gamma \vdash^n e_1\ e_2 : \tau_2} \qquad \textsc{REF}\ \frac{\Gamma \vdash^n e : \mathsf{Int}}{\Gamma \vdash^n \mathsf{ref}\ e : \mathsf{Ref\ Int}}$$

$$\textsc{GET}\ \frac{\Gamma \vdash^n e : \mathsf{Ref\ Int}}{\Gamma \vdash^n !e : \mathsf{Int}} \qquad \textsc{SET}\ \frac{\Gamma \vdash^n e_1 : \mathsf{Ref\ Int} \qquad \Gamma \vdash^n e_2 : \mathsf{Int}}{\Gamma \vdash^n e_1 := e_2 : \mathsf{Unit}} \qquad \textsc{QUOTE}\ \frac{\Gamma \vdash^{n+1} e : \tau}{\Gamma \vdash^n \langle e \rangle : \mathsf{Code}\ \tau} \qquad \textsc{SPLICE}\ \frac{\Gamma \vdash^{n-1} e : \mathsf{Code}\ \tau}{\Gamma \vdash^n \$e : \tau}$$

A.2 Core Typing

$\boxed{\sigma \mid \Gamma \vdash \mathcal{M}_d : \Delta_d}$ *(Typing dynamic module)*

$$\textsc{CD-M-VAR}\ \frac{M_d : \Delta_d \in \Gamma}{\sigma \mid \Gamma \vdash M_d : \Delta_d} \qquad \textsc{CD-M-PATH}\ \frac{\sigma \mid \Gamma \vdash p_d : \mathbf{sig}\ \phi_d\ \mathbf{end} \qquad M : \Delta_d \in \phi_d}{\sigma \mid \Gamma \vdash p_d.M_d : \Delta_d}$$

$$\textsc{CD-M-STRUCT}\ \frac{\sigma \mid \Gamma \vdash \mathcal{S}_d : \phi_d}{\sigma \mid \Gamma \vdash \mathbf{struct}\ \mathcal{S}_d\ \mathbf{end} : \mathbf{sig}\ \phi_d\ \mathbf{end}}$$

$\boxed{\sigma \mid \Gamma \vdash \mathcal{S}_d : \phi_d}$ *(Typing dynamic structure)*

$$\textsc{CD-ST-EMPTY}\ \frac{}{\sigma \mid \Gamma \vdash \bullet : \bullet} \qquad \textsc{CD-ST-DEF}\ \frac{\sigma \mid \Gamma \vdash^0 e : \tau \qquad \sigma \mid \Gamma, k : \tau \vdash \mathcal{S}_d : \phi_d}{\sigma \mid \Gamma \vdash (\mathbf{def}\ k = e; \mathcal{S}_d) : (k : \tau; \phi_d)}$$

CD-ST-MODULE
$$\frac{\sigma \mid \Gamma \vdash \mathcal{M}_d : \Delta_d \qquad \sigma \mid \Gamma, M_d : \Delta_d \vdash \mathcal{S}_d : \phi_d}{\sigma \mid \Gamma \vdash (\mathbf{mod}\, M_d = \mathcal{M}_d; \mathcal{S}_d) : (M_d : \Delta_d; \phi_d)}$$

$\boxed{\Gamma \vdash \mathcal{M}_s : \Delta_s}$ *(Typing static module)*

CS-M-VAR
$$\frac{M_s : \Delta_s \in \Gamma}{\Gamma \vdash M_s : \Delta_s}$$

CS-M-PATH
$$\frac{\Gamma \vdash p_s : \mathbf{sig}\, \phi_s\, \mathbf{end} \qquad M_s : \Delta_s \in \phi_s}{\Gamma \vdash p_s.M_s : \Delta_s}$$

CS-M-STRUCT
$$\frac{\Gamma \vdash \mathcal{S}_s : \phi_s}{\Gamma \vdash \mathbf{struct}\, \mathcal{S}_s\, \mathbf{end} : \mathbf{sig}\, \phi_s\, \mathbf{end}}$$

CS-M-FUNCTOR
$$\frac{\Gamma_1 \vdash \mathbf{functor}\, (M : \Delta_1).\mathcal{N} : \Delta_1 \to \Delta_2}{\Gamma \vdash \langle \Gamma_1, \mathbf{functor}(M : \Delta_1).\ \mathcal{N} \rangle : \Delta_1 \to \Delta_2}$$

CS-M-LOCAL
$$\frac{\Gamma \vdash \mathcal{M}_s : \Delta_s \qquad \Gamma, M_s : \Delta_s \vdash \mathcal{M}_s' : \Delta_s'}{\Gamma \vdash \mathbf{let}\, M_s = \mathcal{M}_s\, \mathbf{in}\, \mathcal{M}_s' : \Delta_s'}$$

$\boxed{\Gamma \vdash \mathcal{S}_s : \phi_s}$ *(Typing static structure)*

CS-ST-EMPTY
$$\frac{}{\Gamma \vdash \bullet : \bullet}$$

CS-ST-MACRO
$$\frac{\bullet \mid \Gamma \vdash^{-1} v : \tau \qquad \Gamma, m : \tau \vdash \mathcal{S}_s : \phi_s}{\Gamma \vdash (\mathbf{def}^{\downarrow}\, m = v; \mathcal{S}_s) : (m : \tau; \phi_s)}$$

CS-ST-MODULE
$$\frac{\Gamma \vdash \mathcal{M}_s : \Delta_s \qquad \Gamma, M_s : \Delta_s \vdash \mathcal{S}_s : \phi_s}{\Gamma \vdash (\mathbf{mod}\, M_s = \mathcal{M}_s; \mathcal{S}_s) : (M : \Delta_s; \phi_s)}$$

$\boxed{\sigma \mid \Gamma \vdash^n e : \tau}$ *(Typing expression)*

C-LIT
$$\frac{}{\sigma \mid \Gamma \vdash^n i : \mathsf{Int}}$$

C-UNIT
$$\frac{}{\sigma \mid \Gamma \vdash^n \mathsf{unit} : \mathsf{Unit}}$$

C-VAR
$$\frac{x : (\tau, n) \in \Gamma}{\sigma \mid \Gamma \vdash^n x : \tau}$$

C-ABS
$$\frac{\sigma \mid \Gamma, x : (\tau_1, n) \vdash^n e : \tau_2}{\sigma \mid \Gamma \vdash^n \lambda x : \tau_1.\ e : \tau_1 \to \tau_2}$$

C-APP
$$\frac{\sigma \mid \Gamma \vdash^n e_1 : \tau_1 \to \tau_2 \qquad \sigma \mid \Gamma \vdash^n e_2 : \tau_1}{\sigma \mid \Gamma \vdash^n e_1\, e_2 : \tau_2}$$

C-KVAR
$$\frac{k : \tau \in \Gamma}{\sigma \mid \Gamma \vdash^0 k : \tau}$$

C-MACRO
$$\frac{m : \tau \in \Gamma}{\sigma \mid \Gamma \vdash^{-1} m : \tau}$$

C-PKVAR
$$\frac{\Gamma \vdash p_d : \mathbf{sig}\, \phi_d\, \mathbf{end} \qquad k : \tau \in \phi_d}{\sigma \mid \Gamma \vdash^0 p_d.k : \tau}$$

C-PMACRO
$$\frac{\Gamma \vdash p_s : \mathbf{sig}\, \phi_s\, \mathbf{end} \qquad m : \tau \in \phi_s}{\sigma \mid \Gamma \vdash^{-1} p_s.m : \tau}$$

C-REF
$$\frac{\sigma \mid \Gamma \vdash^n e : \mathsf{Int}}{\sigma \mid \Gamma \vdash^n \mathsf{ref}\, e : \mathsf{Ref\ Int}}$$

C-ASSIGN
$$\frac{\sigma \mid \Gamma \vdash^n e_1 : \mathsf{Ref\ Int} \qquad \sigma \mid \Gamma \vdash^n e_2 : \mathsf{Int}}{\sigma \mid \Gamma \vdash^n e_1 := e_2 : \mathsf{Unit}}$$

C-DEREF
$$\frac{\sigma \mid \Gamma \vdash^n e : \mathsf{Ref\ Int}}{\sigma \mid \Gamma \vdash^n !e : \mathsf{Int}}$$

C-QUOTE
$$\frac{\sigma \mid \Gamma \vdash^{n+1} e : \tau}{\sigma \mid \Gamma \vdash^n \langle e \rangle : \mathsf{Code}\, \tau}$$

C-SPLICE
$$\frac{\sigma \mid \Gamma \vdash^{n-1} e : \mathsf{Code}\, \tau}{\sigma \mid \Gamma \vdash^n \$e : \tau}$$

C-LOC
$$\frac{l \in \sigma}{\sigma \mid \Gamma \vdash^n l : \mathsf{Ref\ Int}}$$

A.3 Core Dynamic Semantics

$\boxed{\sigma_1 \mid \Omega \mid \mathcal{M}_d \longrightarrow \mathcal{M}_d' \mid \sigma_2}$ *(Evaluate module)*

EV-M-STRUCT
$$\frac{\sigma \mid \Omega_d \mid \mathcal{S}_d \longrightarrow \mathcal{S}_d' \mid \sigma'}{\sigma \mid \Omega_d \mid \mathbf{struct}\, \mathcal{S}_d\, \mathbf{end} \longrightarrow \mathbf{struct}\, \mathcal{S}_d'\, \mathbf{end} \mid \sigma'}$$

EV-M-MVAR
$$\frac{M \mapsto \mathcal{M}_{dv} \in \Omega_d}{\sigma \mid \Omega_d \mid M \longrightarrow \mathcal{M}_{dv} \mid \sigma}$$

EV-M-PMVAR
$$\frac{p_d.M \mapsto \mathcal{M}_{dv} \in \Omega_d}{\sigma \mid \Omega_d \mid p_d.M \longrightarrow \lceil \mathcal{M}_{dv} \rceil^{p_d} \mid \sigma}$$

$\boxed{\sigma_1 \mid \Omega \mid \mathcal{S}_d \longrightarrow \mathcal{S}_d' \mid \sigma_2}$ *(Evaluate structure)*

EV-ST-MODULE1
$$\frac{\sigma \mid \Omega_d \mid \mathcal{M}_d \longrightarrow \mathcal{M}_d' \mid \sigma'}{\sigma \mid \Omega_d \mid \mathbf{mod}\, M_d = \mathcal{M}_d; \mathcal{S}_d \longrightarrow \mathbf{mod}\, M_d = \mathcal{M}_d'; \mathcal{S}_d \mid \sigma'}$$

EV-ST-MODULE2
$$\frac{\sigma \mid \Omega_d; M_d \mapsto \mathcal{M}_{dv} \mid \mathcal{S}_d \longrightarrow \mathcal{S}_d' \mid \sigma'}{\sigma \mid \Omega_d \mid \mathbf{mod}\ M_d = \mathcal{M}_{dv}; \mathcal{S}_d \longrightarrow \mathbf{mod}\ M_d = \mathcal{M}_{dv}; \mathcal{S}_d' \mid \sigma'}$$

EV-ST-DEF1
$$\frac{\sigma \mid \Omega_d \mid e \xrightarrow{0} e' \mid \sigma'}{\sigma \mid \Omega_d \mid \mathbf{def}\ k = e; \mathcal{S}_d \longrightarrow \mathbf{def}\ k = e'; \mathcal{S}_d \mid \sigma'}$$

EV-ST-DEF2
$$\frac{\sigma \mid \Omega_d; k \mapsto v \mid \mathcal{S}_d \longrightarrow \mathcal{S}_d' \mid \sigma'}{\sigma \mid \Omega_d \mid \mathbf{def}\ k = v; \mathcal{S}_d \longrightarrow \mathbf{def}\ k = v; \mathcal{S}_d' \mid \sigma'}$$

$\sigma_1 \mid \Omega \mid e \xrightarrow{n} e' \mid \sigma_2$ *(Evaluate expression)*

EV-APP1
$$\frac{\sigma_1 \mid \Omega \mid e_1 \xrightarrow{n} e_1' \mid \sigma_2}{\sigma_1 \mid \Omega \mid e_1\ e_2 \xrightarrow{n} e_1'\ e_2 \mid \sigma_2}$$

EV-APP2
$$\frac{\sigma_1 \mid \Omega \mid e \xrightarrow{n} e' \mid \sigma_2}{\sigma_1 \mid \Omega \mid v^n\ e \xrightarrow{n} v^n\ e' \mid \sigma_2}$$

EV-ABS
$$\frac{\sigma \mid \Omega \mid e \xrightarrow{n+1} e' \mid \sigma}{\sigma \mid \Omega \mid \lambda x : \tau.\ e \xrightarrow{n+1} \lambda x : \tau.\ e' \mid \sigma}$$

EV-BETA
$$\frac{}{\sigma \mid \Omega \mid (\lambda x : \tau.\ e)\ v \xrightarrow{0} e[x \mapsto v] \mid \sigma}$$

EV-REF1
$$\frac{\sigma_1 \mid \Omega \mid e \xrightarrow{n} e' \mid \sigma_2}{\sigma_1 \mid \Omega \mid \mathbf{ref}\ e \xrightarrow{n} \mathbf{ref}\ e' \mid \sigma_2}$$

EV-REF
$$\frac{l \notin \sigma}{\sigma \mid \Omega \mid \mathbf{ref}\ v \xrightarrow{0} l \mid \sigma, l \mapsto v}$$

EV-ASSIGN1
$$\frac{\sigma_1 \mid \Omega \mid e_1 \xrightarrow{n} e_1' \mid \sigma_2}{\sigma_1 \mid \Omega \mid e_1 := e_2 \xrightarrow{n} e_1' := e_2 \mid \sigma_2}$$

EV-ASSIGN2
$$\frac{\sigma_1 \mid \Omega \mid e \xrightarrow{n} e' \mid \sigma_2}{\sigma_1 \mid \Omega \mid v^n := e \xrightarrow{n} v^n := e' \mid \sigma_2}$$

EV-ASSIGN
$$\frac{l \in \sigma}{\sigma \mid \Omega \mid l := v \xrightarrow{0} \mathbf{unit} \mid \sigma[l \mapsto v]}$$

EV-DEREF1
$$\frac{\sigma_1 \mid \Omega \mid e \xrightarrow{n} e' \mid \sigma_2}{\sigma_1 \mid \Omega \mid !e \xrightarrow{n} !e' \mid \sigma_2}$$

EV-DEREF
$$\frac{l \mapsto v \in \sigma}{\sigma \mid \Omega \mid !l \xrightarrow{0} v \mid \sigma}$$

EV-QUOTE
$$\frac{\sigma_1 \mid \Omega \mid e_1 \xrightarrow{n+1} e_2 \mid \sigma_2}{\sigma_1 \mid \Omega \mid \langle e_1 \rangle \xrightarrow{n} \langle e_2 \rangle \mid \sigma_2}$$

EV-SPLICE
$$\frac{\sigma_1 \mid \Omega \mid e_1 \xrightarrow{n} e_2 \mid \sigma_2}{\sigma_1 \mid \Omega \mid \$e_1 \xrightarrow{n+1} \$e_2 \mid \sigma_2}$$

EV-SPLICECODE
$$\frac{}{\sigma \mid \Omega \mid \$\langle v^1 \rangle \xrightarrow{1} v^1 \mid \sigma}$$

EV-KVAR
$$\frac{k \mapsto v \in \Omega_d}{\sigma \mid \Omega_d \mid k \xrightarrow{0} v \mid \sigma}$$

EV-PKVAR
$$\frac{p_d.k \mapsto v \in \Omega_d}{\sigma \mid \Omega_d \mid p_d.k \xrightarrow{0} \lceil v \rceil^{p_d} \mid \sigma}$$

EV-MACRO
$$\frac{m \mapsto v \in \Omega_s}{\sigma \mid \Omega_s \mid m \xrightarrow{0} v \mid \sigma}$$

EV-PMACRO
$$\frac{p_s.m \mapsto v \in \Omega_s}{\sigma \mid \Omega_s \mid p_s.m \xrightarrow{0} \lceil v \rceil^{p_s} \mid \sigma}$$

$p_d \mapsto \mathcal{M}_{dv} \in \Omega_d$ *(Dynamic path lookup)*

LOOKUP-PD-VAR
$$\frac{}{M_d \mapsto \mathcal{M}_{dv} \in \Omega_{d1}; M \mapsto \mathcal{M}_{dv}; \Omega_{d2}}$$

LOOKUP-PD-PATH
$$\frac{p_d \mapsto \mathbf{struct}\ \mathcal{S}_{dv}\ \mathbf{end} \in \Omega_d \qquad M_d = \mathcal{M}_{dv} \in \mathcal{S}_{dv}}{p_d.M_d \mapsto \lceil \mathcal{M}_{dv} \rceil^{p_d} \in \Omega_d}$$

$p_d.k \mapsto v \in \Omega_d$ *(Dynamic definition lookup)*

LOOKUP-K-KVAR
$$\frac{p_s \mapsto \mathbf{struct}\ \mathcal{S}_{dv}\ \mathbf{end} \in \Omega_d \qquad k = v \in \mathcal{S}_{dv}}{p_s.k \mapsto v \in \Omega_d}$$

$p_s \mapsto \mathcal{M}_s \in \Omega_s$ *(Static path lookup)*

LOOKUP-PS-VAR1
$$\frac{}{M_s \mapsto [C](\mathbf{struct}\ \mathcal{S}_s\ \mathbf{end}) \in \Omega_1; M_s \mapsto [C](\mathbf{struct}\ \mathcal{S}_s\ \mathbf{end}); \Omega_2}$$

LOOKUP-PS-VAR2
$$\frac{M_s' \mapsto \mathcal{M}_s \in \Omega_1; C}{M_s \mapsto [C]\mathcal{M}_s \in \Omega_1; M_s \mapsto [C]M_s'; \Omega_2}$$

LOOKUP-PS-VAR3
$$\frac{p_s.M_s' \mapsto \mathcal{M}_s \in \Omega_1; C}{M_s \mapsto [C]\mathcal{M}_s \in \Omega_1; M_s \mapsto [C]p_s.M_s'; \Omega_2}$$

$$\textsc{lookup-ps-path}\quad \frac{p_s \mapsto [C](\mathbf{struct}\ \mathcal{S}_s\ \mathbf{end}) \in \Omega \qquad M_s = \mathcal{M}_s\ \in\ \mathcal{S}_s}{p_s.M_s \mapsto [C](\lceil \mathcal{M}_s \rceil^{p_s}) \in \Omega}$$

C is a subset of Ω_s, so we also write $p_s \mapsto \mathcal{M}_s \in C$.

$\boxed{p_s.m \mapsto v \in \Omega_s}$ *(Static definition lookup)*

$$\textsc{lookup-m-kvar}\quad \frac{p_s \mapsto [C]\mathbf{struct}\ \mathcal{S}_s\ \mathbf{end} \in \Omega_s \qquad m = v\ \in\ \mathcal{S}_s}{p_s.m \mapsto [C]v \in \Omega_s}$$

$[C]v$ substitutes the macro definitions of C in v.

Prefixing a path			
$\Omega_d \vdash \lceil \mathcal{M}_d \rceil^{p_d}$	$=$	$\lceil \mathcal{M}_d \rceil^{\mathcal{S}_{dv}}_{p_d}$	where $p_d \mapsto \mathbf{struct}\ \mathcal{S}_{dv}\ \mathbf{end} \in \Omega_d$
$\lceil \mathcal{M}_d \rceil^{\bullet}_{p_d}$	$=$	$\mathcal{M}_d$	
$\lceil \mathcal{M}_d \rceil^{\mathbf{def}\ k=v;\mathcal{S}_{dv}}_{p_d}$	$=$	$\lceil \mathcal{M}_d[k \mapsto p_d.k] \rceil^{\mathcal{S}_{dv}}_{p_d}$	
$\lceil \mathcal{M}_d \rceil^{\mathbf{mod}\ M_d=\mathcal{M}_{dv};\mathcal{S}_{dv}}_{p_d}$	$=$	$\lceil \mathcal{M}_d[M_d \mapsto p_d.M_d] \rceil^{\mathcal{S}_{dv}}_{p_d}$	
$\Omega_d \vdash \lceil v \rceil^{p_d}$	$=$	$\lceil v \rceil^{\mathcal{S}_{dv}}_{p_d}$	where $p_d \mapsto \mathbf{struct}\ \mathcal{S}_{dv}\ \mathbf{end} \in \Omega_d$
$\lceil v \rceil^{\bullet}_{p_d}$	$=$	v	
$\lceil v \rceil^{\mathbf{def}\ k=v;\mathcal{S}_{dv}}_{p_d}$	$=$	$\lceil v[k \mapsto p_d.k] \rceil^{\mathcal{S}_{dv}}_{p_d}$	
$\lceil v \rceil^{\mathbf{mod}\ M_d=\mathcal{M}_{dv};\mathcal{S}_{dv}}_{p_d}$	$=$	$\lceil v[M_d \mapsto p_d.M_d] \rceil^{\mathcal{S}_{dv}}_{p_d}$	
$\Omega_s \vdash \lceil v \rceil^{p_s}$	$=$	$\lceil v \rceil^{\mathcal{S}_s}_{p_s}$	where $p_s \mapsto [C]\mathbf{struct}\ \mathcal{S}_s\ \mathbf{end} \in \Omega_s$
$\lceil v \rceil^{\bullet}_{p_s}$	$=$	v	
$\lceil v \rceil^{\mathbf{def}^{\downarrow}\ m=v;\mathcal{S}_s}_{p_s}$	$=$	$\lceil v[m \mapsto p_s.m] \rceil^{\mathcal{S}_s}_{p_s}$	
$\lceil v \rceil^{\mathbf{mod}\ M_s=\mathcal{M}_s;\mathcal{S}_s}_{p_s}$	$=$	$\lceil v[M_s \mapsto p_s.M_s] \rceil^{\mathcal{S}_s}_{p_s}$	

A.4 Elaboration

$\boxed{\sigma_1 \mid \Omega \mid \Gamma \vdash \mathcal{M} \rightsquigarrow \mathcal{L} \mid \mathcal{M}_d \mid \mathcal{M}_s \mid \sigma_2}$ *(Elaborate module)*

$$\textsc{e-m-struct}\quad \frac{\sigma_1 \mid \Omega \mid \Gamma \vdash \mathcal{S} \rightsquigarrow \mathcal{S}_d \mid \mathcal{S}_s \mid \sigma_2}{\sigma_1 \mid \Omega \mid \Gamma \vdash \mathbf{struct}\ \mathcal{S}\ \mathbf{end} \rightsquigarrow \bullet \mid \mathbf{struct}\ \mathcal{S}_d\ \mathbf{end} \mid \mathbf{struct}\ \mathcal{S}_s\ \mathbf{end} \mid \sigma_2}$$

$$\textsc{e-m-mvar}\quad \frac{}{\sigma \mid \Omega \mid \Gamma \vdash M \rightsquigarrow \bullet \mid M_d \mid M_s \mid \sigma} \qquad \textsc{e-m-pmvar}\quad \frac{\sigma \mid \Omega \mid \Gamma \vdash p \rightsquigarrow \bullet \mid p_d \mid p_s \mid \sigma}{\sigma \mid \Omega \mid \Gamma \vdash p.M \rightsquigarrow \bullet \mid p_d.M_d \mid p_s.M_s \mid \sigma}$$

$$\textsc{e-m-functor}\quad \frac{}{\sigma \mid \Omega \mid \Gamma \vdash \mathbf{functor}\ (M : \Delta_1).\mathcal{N} \rightsquigarrow \bullet \mid \mathbf{struct}\ \bullet\ \mathbf{end} \mid \langle \Gamma, \mathbf{functor}(M : \Delta_1).\ \mathcal{N} \rangle \mid \sigma}$$

$$\textsc{e-m-app}\quad \frac{\sigma_1 \mid \Omega \mid \Gamma \vdash \mathcal{F} \rightsquigarrow \mathcal{L}_1 \mid \mathcal{F}_d \mid \mathcal{F}_s \mid \sigma_2 \qquad \sigma_2 \mid \Omega \mid \Gamma \vdash \mathcal{M} \rightsquigarrow \mathcal{L}_2 \mid \mathcal{M}_d \mid \mathcal{M}_s \mid \sigma_3 \qquad \sigma_3 \mid \Omega \vdash \mathcal{F}_s \bullet \mathcal{M}_d \mid \mathcal{M}_s \Longrightarrow \mathcal{L}_3 \mid \mathcal{N}_d \mid \mathcal{N}_s \mid \sigma_4}{\sigma_1 \mid \Omega \mid \Gamma \vdash \mathcal{F}\ \mathcal{M} \rightsquigarrow \mathcal{L}_1; \mathcal{L}_2; \mathcal{L}_3 \mid \mathcal{N}_d \mid \mathcal{N}_s \mid \sigma_4}$$

$\boxed{\sigma_1 \mid \Omega \mid \Gamma \vdash \mathcal{S} \rightsquigarrow \mathcal{S}_d \mid \mathcal{S}_s \mid \sigma_2}$ *(Elaborate structure)*

E-ST-EMPTY

$$\frac{}{\sigma \mid \Omega \mid \Gamma \vdash \bullet \rightsquigarrow \bullet \mid \bullet \mid \sigma}$$

E-ST-DEF

$$\frac{\sigma_1 \mid \Omega \vdash_c e \rightsquigarrow e \mid \sigma_2 \qquad \sigma_2 \mid \Omega \mid \Gamma, k : \tau \vdash \mathcal{S} \rightsquigarrow \mathcal{S}_d \mid \mathcal{S}_s \mid \sigma_3}{\sigma_1 \mid \Omega \mid \Gamma \vdash (\mathbf{def}\ k = e; \mathcal{S}) \rightsquigarrow \mathbf{def}\ k = e; \mathcal{S}_d \mid \mathcal{S}_s \mid \sigma_3}$$

E-ST-MACRO

$$\frac{\sigma_1 \mid \Omega \vdash_c e \rightsquigarrow v \mid \sigma_2 \qquad \sigma_2 \mid \Omega; m \mapsto v \mid \Gamma, m : \tau \vdash \mathcal{S} \rightsquigarrow \mathcal{S}_d \mid \mathcal{S}_s \mid \sigma_3}{\sigma_1 \mid \Omega \mid \Gamma \vdash (\mathbf{def}^{\downarrow}\ m = e; \mathcal{S}) \rightsquigarrow \mathcal{S}_d \mid \mathbf{def}^{\downarrow}\ m = v; \mathcal{S}_s \mid \sigma_3}$$

E-ST-MODULE

$$\frac{\sigma_1 \mid \Omega \mid \Gamma \vdash \mathcal{M} \rightsquigarrow \mathcal{L} \mid \mathcal{M}_d \mid \mathcal{M}_s \mid \sigma_2 \qquad \sigma_2 \mid \Omega; M_s \mapsto (\lceil \mathcal{M}_s \rceil^{\mathcal{M}_d}) \mid \Gamma, M : \Delta \vdash \mathcal{S} \rightsquigarrow \mathcal{S}_d \mid \mathcal{S}_s \mid \sigma_3}{\sigma_1 \mid \Omega \mid \Gamma \vdash (\mathbf{mod}\ M : \Delta = \mathcal{M}; \mathcal{S}) \rightsquigarrow \mathcal{L}; \mathbf{mod}\ M_d = \mathcal{M}_d; \mathcal{S}_d \mid \mathbf{mod}\ M_s = \lceil \mathcal{M}_s \rceil^{\mathcal{M}_d}; \mathcal{S}_s \mid \sigma_3}$$

$\boxed{\sigma_1 \mid \Omega \vdash \mathcal{F}_s \bullet \mathcal{M}_d \mid \mathcal{M}_s \Longrightarrow \mathcal{L} \mid \mathcal{M}_d \mid \mathcal{M}_s \mid \sigma_2}$ *(Functor applications)*

E-APP-FUNCTOR

$$\frac{\sigma_1 \mid \Omega; C; M_s \mapsto \lceil \mathcal{M}_s \rceil^{\mathcal{M}_d} \mid \Gamma, M : \Delta_1 \vdash \mathcal{N} \rightsquigarrow \mathcal{L} \mid \mathcal{N}_d \mid \mathcal{N}_s \mid \sigma_2}{\sigma_1 \mid \Omega \vdash [C]\langle \Gamma, \mathbf{functor}(M : \Delta).\ \mathcal{N} \rangle \bullet \mathcal{M}_d \mid \mathcal{M}_s \Longrightarrow \mathbf{mod}\ M_d = \mathcal{M}_d; \mathcal{L} \mid \mathcal{N}_d \mid [C; M_s \mapsto \mathcal{M}_s]\mathcal{N}_s \mid \sigma_2}$$

E-APP-MVAR1

$$\frac{M_s \mapsto \mathcal{F}_s \in \Omega \qquad \sigma_1 \mid \Omega \vdash \mathcal{F}_s \bullet \mathcal{M}_d \mid \mathcal{M}_s \Longrightarrow \mathcal{L} \mid \mathcal{N}_d \mid \mathcal{N}_s \mid \sigma_2}{\sigma_1 \mid \Omega \vdash [C]M_s \bullet \mathcal{M}_d \mid \mathcal{M}_s \Longrightarrow \mathcal{L} \mid \mathcal{N}_d \mid \mathcal{N}_s \mid \sigma_2}$$

E-APP-MVAR2

$$\frac{M_s \mapsto \mathcal{F}_s \in C \qquad \sigma_1 \mid \Omega \vdash [C]\mathcal{F}_s \bullet \mathcal{M}_d \mid \mathcal{M}_s \Longrightarrow \mathcal{L} \mid \mathcal{N}_d \mid \mathcal{N}_s \mid \sigma_2}{\sigma_1 \mid \Omega \vdash [C]M_s \bullet \mathcal{M}_d \mid \mathcal{M}_s \Longrightarrow \mathcal{L} \mid \mathcal{N}_d \mid \mathcal{N}_s \mid \sigma_2}$$

E-APP-PMVAR1

$$\frac{p_s.M_s \mapsto \mathcal{F}_s \in \Omega \qquad \sigma_1 \mid \Omega \vdash \lceil \mathcal{F}_s \rceil^{p_s} \bullet \mathcal{M}_d \mid \mathcal{M}_s \Longrightarrow \mathcal{L} \mid \mathcal{N}_d \mid \mathcal{N}_s \mid \sigma_2}{\sigma_1 \mid \Omega \vdash [C]p_s.M_s \bullet \mathcal{M}_d \mid \mathcal{M}_s \Longrightarrow \mathcal{L} \mid \mathcal{N}_d \mid \mathcal{N}_s \mid \sigma_2}$$

E-APP-PMVAR2

$$\frac{p_s.M_s \mapsto \mathcal{F}_s \in C \qquad \sigma_1 \mid \Omega \vdash [C](\lceil \mathcal{F}_s \rceil^{p_s}) \bullet \mathcal{M}_d \mid \mathcal{M}_s \Longrightarrow \mathcal{L} \mid \mathcal{N}_d \mid \mathcal{N}_s \mid \sigma_2}{\sigma_1 \mid \Omega \vdash [C]p_s.M_s \bullet \mathcal{M}_d \mid \mathcal{M}_s \Longrightarrow \mathcal{L} \mid \mathcal{N}_d \mid \mathcal{N}_s \mid \sigma_2}$$

$\boxed{\sigma_1 \mid \Omega \vdash_\star e \rightsquigarrow e \mid \sigma_2}$ *(Elaborate expression)*

E-LIT

$$\frac{}{\sigma \mid \Omega \vdash_\star i \rightsquigarrow i \mid \sigma}$$

E-UNIT

$$\frac{}{\sigma \mid \Omega \vdash_\star \mathsf{unit} \rightsquigarrow \mathsf{unit} \mid \sigma}$$

E-VAR

$$\frac{}{\sigma \mid \Omega \vdash_\star x \rightsquigarrow x \mid \sigma}$$

E-KVAR

$$\frac{}{\sigma \mid \Omega \vdash_\star k \rightsquigarrow k \mid \sigma}$$

E-MACRO

$$\frac{}{\sigma \mid \Omega \vdash_\star m \rightsquigarrow m \mid \sigma}$$

E-PKVAR

$$\frac{k : \tau \in \phi}{\sigma \mid \Omega \vdash_\star p.k \rightsquigarrow (p)_d.k \mid \sigma}$$

E-PMACRO

$$\frac{}{\sigma \mid \Omega \vdash_\star p.m \rightsquigarrow (p)_s.m \mid \sigma}$$

E-ABS

$$\frac{\sigma_1 \mid \Omega \vdash_\star e \rightsquigarrow e \mid \sigma_2}{\sigma_1 \mid \Omega \vdash_\star \lambda x : \tau_1.\ e \rightsquigarrow \lambda x : \tau.\ e \mid \sigma_2}$$

E-APP

$$\frac{\sigma_1 \mid \Omega \vdash_\star e_1 \rightsquigarrow e_1 \mid \sigma_2 \qquad \sigma_2 \mid \Omega \vdash_\star e_2 \rightsquigarrow e_2 \mid \sigma_3}{\sigma_1 \mid \Omega \vdash_\star e_1\ e_2 \rightsquigarrow e_1\ e_2 \mid \sigma_3}$$

E-REF

$$\frac{\sigma_1 \mid \Omega \vdash_\star e \rightsquigarrow e \mid \sigma_2}{\sigma_1 \mid \Omega \vdash_\star \mathsf{ref}\ e \rightsquigarrow \mathsf{ref}\ e \mid \sigma_2}$$

E-GET

$$\frac{\sigma_1 \mid \Omega \vdash_\star e \rightsquigarrow e \mid \sigma_2}{\sigma_1 \mid \Omega \vdash_\star !e \rightsquigarrow !e \mid \sigma_2}$$

E-SET

$$\frac{\sigma_1 \mid \Omega \vdash_\star e_1 \rightsquigarrow e_1 \mid \sigma_2 \qquad \sigma_2 \mid \Omega \vdash_\star e_2 \rightsquigarrow e_2 \mid \sigma_3}{\sigma_1 \mid \Omega \vdash_\star e_1 := e_2 \rightsquigarrow e_1 := e_2 \mid \sigma_3}$$

E-QUOTE

$$\frac{\sigma_1 \mid \Omega \vdash_q e \rightsquigarrow e \mid \sigma_2}{\sigma_1 \mid \Omega \vdash_{c \vee s} \langle e \rangle \rightsquigarrow \langle e \rangle \mid \sigma_2}$$

E-SPLICE

$$\frac{\sigma_1 \mid \Omega \vdash_s e \rightsquigarrow e \mid \sigma_2}{\sigma_1 \mid \Omega \vdash_q \$e \rightsquigarrow \$e \mid \sigma_2}$$

E-CODEGEN

$$\frac{\sigma_1 \mid \Omega \vdash_s e \rightsquigarrow e \mid \sigma_2 \qquad \sigma_2 \mid \Omega \mid e \xrightarrow{0}{}^{*} \langle v^1 \rangle \mid \sigma_3}{\sigma_1 \mid \Omega \vdash_c \$e \rightsquigarrow v^1 \mid \sigma_3}$$

References

Andrew W. Appel and David B. MacQueen. 1994. Separate Compilation for Standard ML. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*, Vivek Sarkar, Barbara G. Ryder, and Mary Lou Soffa (Eds.). ACM, 13–23. https://doi.org/10.1145/178243.178245

Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. 2003. Implementing multi-stage languages using ASTs, gensym, and reflection. In *International Conference on Generative Programming and Component Engineering*. Springer, 57–76.

Jacques Carette, Mustafa Elsheikh, and W. Spencer Smith. 2011. A generative geometric kernel. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2011, Austin, TX, USA, January 24-25, 2011*, Siau-Cheng Khoo and Jeremy G. Siek (Eds.). ACM, 53–62. https://doi.org/10.1145/1929501.1929510

Jacques Carette and Oleg Kiselyov. 2005. Multi-stage Programming with Functors and Monads: Eliminating Abstraction Overhead from Generic Code. In *Generative Programming and Component Engineering, 4th International Conference, GPCE 2005, Tallinn, Estonia, September 29 - October 1, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3676)*, Robert Glück and Michael R. Lowry (Eds.). Springer, 256–274. https://doi.org/10.1007/11561347_18

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

Karl Crary, Robert Harper, and Sidd Puri. 1999. What is a Recursive Module?. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, May 1-4, 1999*, Barbara G. Ryder and Benjamin G. Zorn (Eds.). ACM, 50–63. https://doi.org/10.1145/301618.301641

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

Rowan Davies and Frank Pfenning. 1996. A Modal Analysis of Staged Computation. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, Hans-Juergen Boehm and Guy L. Steele Jr. (Eds.). ACM Press, 258–270. https://doi.org/10.1145/237721.237788

Daniel de Rauglaudre. 2007. *Camlp5 - Reference Manual*. Institut National de Recherche en Informatique et Automatique.

Derek R. Dreyer, Robert Harper, and Karl Crary. 2001. *Toward a Practical Type Theory for Recursive Modules*. Technical Report CMU-CS-01-112. School of Computer Science, Carnegie Mellon University.

Martin Elsman. 1999. Static Interpretation of Modules. In *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September 27-29, 1999*, Didier Rémy and Peter Lee (Eds.). ACM, 208–219. https://doi.org/10.1145/317636.317800

Matthew Flatt. 2002. Composable and Compilable Macros: You Want It When?. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming* (Pittsburgh, PA, USA) *(ICFP '02)*. Association for Computing Machinery, New York, NY, USA, 72–83. https://doi.org/10.1145/581478.581486

Steven E. Ganz, Amr Sabry, and Walid Taha. 2001. Macros as Multi-Stage Computations: Type-Safe, Generative, Binding Macros in MacroML. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001*, Benjamin C. Pierce (Ed.). ACM, 74–85. https://doi.org/10.1145/507635.507646

Robert Harper, John C Mitchell, and Eugenio Moggi. 1989. Higher-order modules and the phase distinction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 341–354.

Robert Harper and Christopher A. Stone. 2000. A type-theoretic interpretation of standard ML. In *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, Gordon D. Plotkin, Colin Stirling, and Mads Tofte (Eds.). The MIT Press, 341–388.

Jun Inoue, Oleg Kiselyov, and Yukiyoshi Kameyama. 2016. Staging beyond Terms: Prospects and Challenges. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (St. Petersburg, FL, USA) *(PEPM '16)*. Association for Computing Machinery, New York, NY, USA, 103–108. https://doi.org/10.1145/2847538.2847548

Junyoung Jang, Samuel Gélineau, Stefan Monnier, and Brigitte Pientka. 2022. Mœbius: metaprogramming using contextual types: the stage where system f can pattern match on itself. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–27. https://doi.org/10.1145/3498700

Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml. In *Functional and Logic Programming*, Michael Codish and Eijiro Sumii (Eds.). Springer International Publishing, Cham, 86–102. https://doi.org/10.1007/978-3-319-07151-0_6

András Kovács. 2022. Staged compilation with two-level type theory. *Proc. ACM Program. Lang.* 6, ICFP (2022), 540–569. https://doi.org/10.1145/3547641

Xavier Leroy. 1994. Manifest types, modules, and separate compilation. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 109–122.

John C. Mitchell and Robert Harper. 1988. The Essence of ML. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, Jeanne Ferrante and Peter Mager (Eds.). ACM Press, 28–46. https://doi.org/10.1145/73560.73563

Yuito Murase, Yuichi Nishiwaki, and Atsushi Igarashi. 2023. Contextual Modal Type Theory with Polymorphic Contexts. In *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 13990)*, Thomas Wies (Ed.). Springer, 281–308. https://doi.org/10.1007/978-3-031-30044-8_11

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Trans. Comput. Log.* 9, 3 (2008), 23:1–23:49. https://doi.org/10.1145/1352582.1352591

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

Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering* (Eindhoven, The Netherlands) *(GPCE '10)*. ACM, New York, NY, USA, 127–136. https://doi.org/10.1145/1868294.1868314

Andreas Rossberg. 2018. 1ML - Core and modules united. *J. Funct. Program.* 28 (2018), e22. https://doi.org/10.1017/S0956796818000205

Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. 2014. F-ing modules. *J. Funct. Program.* 24, 5 (2014), 529–607. https://doi.org/10.1017/S0956796814000264

Yuhi Sato and Yukiyoshi Kameyama. 2021. Type-Safe Generation of Modules in Applicative and Generative Styles. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Chicago, IL, USA) *(GPCE 2021)*. Association for Computing Machinery, New York, NY, USA, 184–196. https://doi.org/10.1145/3486609.3487209

Yuhi Sato, Yukiyoshi Kameyama, and Takahisa Watanabe. 2020. Module Generation without Regret. In *Proceedings of the 2020 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (New Orleans, LA, USA) *(PEPM 2020)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3372884.3373160

Zhong Shao. 1997. An Overview of the FLINT/ML Compiler. Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97), Amsterdam, The Netherlands.

Zhong Shao. 1999. Transparent Modules with Fully Syntactic Signatures. In *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September 27-29, 1999*, Didier Rémy and Peter Lee (Eds.). ACM, 220–232. https://doi.org/10.1145/317636.317801

Tim Sheard and Simon Peyton Jones. 2002. Template Meta-Programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell* (Pittsburgh, Pennsylvania) *(Haskell '02)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/581690.581691

Nicolas Stucki, Aggelos Biboudis, and Martin Odersky. 2018. A practical unification of multi-stage programming and macros. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2018, Boston, MA, USA, November 5-6, 2018*, Eric Van Wyk and Tiark Rompf (Eds.). ACM, 14–27. https://doi.org/10.1145/3278122.3278139

Takashi Suwa and Atsushi Igarashi. 2024. An ML-Style Module System for Cross-Stage Type Abstraction in Multi-stage Programming. In *Functional and Logic Programming - 17th International Symposium, FLOPS 2024, Kumamoto, Japan, May 15-17, 2024, Proceedings (Lecture Notes in Computer Science, Vol. 14659)*, Jeremy Gibbons and Dale Miller (Eds.). Springer, 237–272. https://doi.org/10.1007/978-981-97-2300-3_13

Kenichi Suzuki, Oleg Kiselyov, and Yukiyoshi Kameyama. 2016. Finally, safely-extensible and efficient language-integrated query. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Martin Erwig and Tiark Rompf (Eds.). ACM, 37–48. https://doi.org/10.1145/2847538.2847542

Don Syme. 2006. Leveraging. NET meta-programming components from F# integrated queries and interoperable heterogeneous execution. In *Proceedings of the 2006 workshop on ML*. 43–54.

Walid Taha. 1999. *Multi-Stage Programming: Its Theory and Applications*. Ph. D. Dissertation. Halmstad University, Sweden. https://urn.kb.se/resolve?urn=urn:nbn:se:hh:diva-15052

Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. 1998. Multi-stage programming: Axiomatization and type safety. In *International Colloquium on Automata, Languages, and Programming*. Springer, 918–929.

Walid Taha and Patricia Johann. 2003. Staged Notational Definitions. In *Generative Programming and Component Engineering, Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2830)*, Frank Pfenning and Yannis Smaragdakis (Eds.). Springer, 97–116. https://doi.org/10.1007/978-3-540-39815-8_6

Peter Thiemann. 1999. Interpreting Specialization in Type Theory. In *Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, San Antonio, Texas, USA, January 22-23, 1999. Technical report BRICS-NS-99-1*, Olivier Danvy (Ed.). University of Aarhus, 30–43.

Ryo Tokuda and Yukiyoshi Kameyama. 2023. Generating Programs for Polynomial Multiplication with Correctness Assurance. In *Proceedings of the 2023 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation, PEPM 2023, Boston, MA, USA, January 16-17, 2023*, Edwin C. Brady and Jens Palsberg (Eds.). ACM, 27–40. https://doi.org/10.1145/3571786.3573017

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

Stephen Weeks. 2006. Whole-program compilation in MLton. In *Proceedings of the 2006 Workshop on ML* (Portland, Oregon, USA) *(ML '06)*. Association for Computing Machinery, New York, NY, USA, 1. https://doi.org/10.1145/1159876.1159877

Stefan Wehr and Manuel M. T. Chakravarty. 2008. ML Modules and Haskell Type Classes: A Constructive Comparison. In *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5356)*, G. Ramalingam (Ed.). Springer, 188–204. https://doi.org/10.1007/978-3-540-89330-1_14

Leo White. 2013. Extension points for OCaml. OCaml Users and Developers Workshop.

Leo White, Frédéric Bour, and Jeremy Yallop. 2014. Modular implicits. In *Proceedings ML Family/OCaml Users and Developers workshops, ML/OCaml 2014, Gothenburg, Sweden, September 4-5, 2014 (EPTCS, Vol. 198)*, Oleg Kiselyov and Jacques Garrigue (Eds.). 22–63. https://doi.org/10.4204/EPTCS.198.2

Andrew K Wright and Matthias Felleisen. 1994. A syntactic approach to type soundness. *Information and computation* 115, 1 (1994), 38–94.

Ningning Xie, Matthew Pickering, Andres Löh, Nicolas Wu, Jeremy Yallop, and Meng Wang. 2022. Staging with Class: A Specification for Typed Template Haskell. *Proc. ACM Program. Lang.* 6, POPL, Article 61 (jan 2022), 30 pages. https://doi.org/10.1145/3498723

Ningning Xie, Leo White, Olivier Nicole, and Jeremy Yallop. 2023. MacoCaml: Staging Composable and Compilable Macros. *Proc. ACM Program. Lang.* 7, ICFP, Article 209 (aug 2023), 45 pages. https://doi.org/10.1145/3607851

Jeremy Yallop. 2017. Staged generic programming. *Proc. ACM Program. Lang.* 1, ICFP (2017), 29:1–29:29. https://doi.org/10.1145/3110273

Jeremy Yallop, David Sheets, and Anil Madhavapeddy. 2018a. A modular foreign function interface. *Sci. Comput. Program.* 164 (2018), 82–97. https://doi.org/10.1016/J.SCICO.2017.04.002

Jeremy Yallop, Tamara von Glehn, and Ohad Kammar. 2018b. Partially-static data as free extension of algebras. *Proc. ACM Program. Lang.* 2, ICFP (2018), 100:1–100:30. https://doi.org/10.1145/3236795

Received 2024-02-28; accepted 2024-06-18