MacoCaml: Staging Composable and Compilable Macros

NINGNING XIE, University of Toronto, Canada
LEO WHITE, Jane Street Capital, UK
OLIVIER NICOLE, Tarides, France
JEREMY YALLOP, University of Cambridge, UK

We introduce MacoCaml, a new design and implementation of compile-time code generation for the OCaml language. MacoCaml features a novel combination of macros with phase separation and quotation-based staging, where macros are considered as compile-time bindings, expression cross evaluation phases using staging annotations, and compile-time evaluation happens inside top-level splices. We provide a theoretical foundation for MacoCaml by formalizing a typed source calculus *maco* that supports interleaving typing and compile-time code generation, references with explicit compile-time heaps, and modules. We study various crucial properties including *soundness* and *phase distinction*. We have implemented MacoCaml in the OCaml compiler, and ported two substantial existing libraries to validate our implementation.

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, Compile-time code generation, OCaml

1 INTRODUCTION

Program generation is a powerful and expressive approach to eliminating abstraction overhead and improving program performance, which has been studied and implemented in a variety of languages with different forms, such as C++ templates [Abrahams and Gurtovoy 2004], macros [Burmako 2013; Clinger and Rees 1991; Flatt 2002; Kohlbecker et al. 1986], or multi-stage programming for compile time [Kovács 2022; Sheard and Jones 2002; Xie et al. 2022] and runtime code generation [Calcagno et al. 2003b; Kiselyov 2014; Rompf and Odersky 2010; Taha et al. 1998].

This paper presents the design and implementation of MacoCaml, an extension of OCaml with support for *compile-time code generation*. We offer the following contributions:

- **A unifying framework for phases and staging**: In the design space of compile-time code generation, there has been work supporting *macros* with phase separation [Flatt 2002] and work supporting *quotation-based staging* [Sheard and Jones 2002], but the two notions are often considered separate. We present a novel, simple, yet effective combination of the two techniques, where macros are considered as compile-time bindings, expressions cross evaluation phases using staging annotations, and macro invocation is part of compile-time evaluation implied by *top-level splices* (§2.1 and 2.2). This way, we unify macros and staging in a single framework, enabling users to apply the unified abstraction to build reusable, composable, and maintainable programs. Moreover, we show that staged macros integrate smoothly with features found in real-world languages, including *module imports* (§2.3) and *computations with side effects* (§2.4).
- **A comprehensive formalism of a feature-rich macro calculus**: We provide a theoretical foundation for MacoCaml by formalizing a typed source calculus *maco* (§3). Staging calculi in

ICFP '23, Sept 4–9, 2023, Seattle, WA

ACM ISBN 978-1-4503-XXXX-X/18/06.
https://doi.org/XXXXXXX.XXXXXXX

the literature often focus on a minimal set of features, but building MacoCaml on the OCaml language forces us to confront several practical issues: interleaving of typing and compile-time code generation, references with explicit compile-time heaps, and modules. We believe that *maco* is the first typed formalism to address such a rich staging feature set. The calculus describes the essence of the macro system in OCaml, laying the foundation for a language feature to be integrated into a full-scale language, allowing for further extensions that use or build on top of the OCaml macro system.

- **Soundness and phase distinction for staged macros**: To model compile-time evaluation, we formalize a core calculus $maco_{core}$ as the compilation target for *maco*, and present a *type-directed elaboration* from *maco* to $maco_{core}$ (§4). Separating the source and the core calculi allows us to distinguish modules from their compiled forms, making the phase separation explicit. We establish key properties of our design, including 1) *type soundness* of $maco_{core}$ (§4.4), 2) *elaboration soundness* from *maco* to $maco_{core}$ (§4.5), and 3) *phase distinction* (§4.5). Thus we formally establish properties essential for safe and modular programming: well-typed source programs generate well-typed core programs. Moreover, the theorems, to our best knowledge, are the first to formally reason about macros with compile-time heaps and prove that compile-time computations do not interfere with runtime computations. Specifically, we show that compile-time heaps can be discarded after compilation, and compile-time computations can be erased before runtime evaluation.
- **A working implementation for OCaml**: We provide an implementation of MacoCaml in the OCaml compiler, following the key ideas in our calculus (§5). The implementation is detailed, and the modified compiler is included as supplementary materials. All examples presented in the paper work in the compiler. To validate our implementation, we have ported two substantial existing libraries: *Strymonas* for *stream fusion* [Kiselyov et al. 2017], and the OCaml library for typed formatting [Vaugon 2013]. The results show that MacoCaml works in practice and can be applied to large-scale implementations.

We discuss related work in the rich design space of staging and macros in §6, and conclude and discuss future work in §7. Our formalism is detailed, and some rules are elided for space reasons. The complete set of rules and all proofs of stated theorems are provided in the anonymous supplementary materials. While this work focuses on OCaml, we believe this study can help with the design and formalism of staging or macros in other programming languages.

2 OVERVIEW OF MACOCAML

This section gives an overview and summarizes the key aspects of MacoCaml. All examples are written in OCaml syntax, and all code compiles and runs as expected in our implementation.

2.1 Programming with Staging and Macros

Consider defining a `power` function that calculates x^n:

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

For example, `power 5 2` returns `32`, as expected. However, while this function can take an arbitrary integer `n`, the abstraction comes with low-level performance overhead for each recursive call.

Macros provide an expressive way to balance abstraction and efficiency. Below we define power as a macro in MacoCaml, assuming that `n` is statically known:

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

macro mpower n = << fun x → $(mpower' n <<x>>) >>  (* int -> (int -> int) expr *)
```

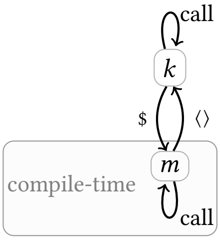


Fig. 1. The call relation between let definitions k and macros m; we use ⟨⟩ for quotation and $ for splicing.

In MacoCaml, macros are always functions, which are the most common case in practice. The macro definition makes use of *staging* annotations: `<<e>>` are *quotations* that delay an expression's computation by turning it into its code representation, while `$e` are *splices* that trigger evaluation of a code representation. From a typing perspective, if `e : t`, then `<<e>> : t expr`; conversely, if `e : t expr`, then `$e : t`. Splicing a call to the macro generates at compile-time a specialized power function with `n` equal to `5`:

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

Calling `power5 2` produces `32`, the same result as before, but now with less runtime overhead as all function calls to `mpower` have been unrolled and inlined.

2.2 Evaluation Phases

The power example illustrates the first key design of our system: we divide program evaluation into two *phases*, where let definitions are *runtime* bindings and macros are *compile-time* bindings. Staging annotations are used to cross phases. The design provides a novel view of macros, unifying macros and staging in a single framework.[1]

Call relation. Fig. 1 depicts the call relation between let definitions (represented as k) and macros (represented as m), where ⟨⟩ is quoting and $ is splicing. As we have seen, we can splice a macro application in a let definition, as in the definition of `power5`. Conversely, we can quote the application of a let-defined function (and splice it elsewhere in the program), delaying its computation to runtime, e.g. `macro delayed32 () = <<power5 2>>`.

The figure features direct relations; in practice we can have splices inside quotations and vice versa, such as `let n = $(<<1 + $(mpower 5) 2>>)`.

Leveled bindings and well-stagedness. Formally, we manage definitions using the notion of a *level*. The level of an expression is defined as the integer given by the number of quotes surrounding it minus the number of splices: quotation increases the level of an expression, while splicing decreases it. Intuitively speaking, levels indicate the evaluation phase of expressions: expressions of negative levels are compile-time expressions, and expressions of level 0 are runtime expressions.[2]

MacoCaml further features leveled bindings: *let definitions are bindings at level* 0*, while macros are bindings at level* −1*. Well-stagedness* specifies that a definition can only be used at the level it is defined. Specifically, the right-hand side of a let definition is type-checked under level 0, and the definition itself can be used at level 0. Similarly, the right-hand side of a macro definition is type-checked under level −1, and the macro itself can be called at level −1. This explains why we

[1] We use *macros* to mean compile-time bindings, which differ from *macros* in systems like Racket; see §6 for more discussion.

[2] In the literature, work on macros (e.g. Flatt [2002]) refers to phase 1 as compile-time and phase 0 as runtime, while work on staged programming (e.g. Xie et al. [2022]) refers to level 0 as runtime and positive levels as future stages. In this work, we use numbers for levels, and write *compile-time* and *runtime* for evaluation phases. We consider levels a syntactic notion used during typechecking, and phases an operational notion as to where programs are run. For example, type-checking (`1+2,<<3>>`) involves two levels (1 + 2 at level 0 and 3 at level 1), while evaluating it at runtime results in (`3,<<3>>`).

can splice macros inside let definitions, or quote let definitions inside macros. This way, we unify macros and staging in a single framework, and all bindings simply follow the same requirement specified by well-stagedness.

Macros and compile-time evaluation. Now the question is: where exactly does a compile-time computation happen? In MacoCaml, compile-time computation happens inside *top-level splices*, namely, splices without surrounding quotations. In the example for power, the top-level splice `$(mpower 5)` generates the definition for `power5` during compilation. In other words, macro applications (such as `mpower 5`) *alone* do not force any compile-time evaluation; rather, macro invocation is part of compile-time evaluation. We explain the key idea with the following definitions:

```
let power5 = $(mpower 5)                                   (* well-typed and mpower expanded *)
let err = mpower 5               (* error: mpower is defined at level -1 but called at 0 *)
macro mpower5 () = mpower 5                           (* well-typed and mpower not expanded *)
macro merr () = $(mpower 5) (* error: mpower is defined at level -1 but called at -2 *)
```

Recall that macros are defined at level −1, allowing macros to be spliced inside top-level splices in let definitions or called inside macro definitions. Here, only `power5` expands `mpower 5` as it is inside a top-level splice; `mpower5` is well-typed but does not expand `mpower`; `err` and `merr` are ill-staged.

The examples demonstrate that MacoCaml provides a systematic approach to combining macros with staging. This distinguishes MacoCaml from macro systems where macro invocation is itself a compile-time computation, in which case `err` and `mpower5` might expand `mpower`.

Interleaving typing and compile-time computations. In MacoCaml, compile-time evaluation happens during typing. As we will see, interleaving typing and compile-time evaluation poses challenges to the system's type safety. A similar design is used in Template Haskell [Sheard and Jones 2002], but unlike Template Haskell where the result of splicing a code value may not type-check, MacoCaml offers a static guarantee: we prove that well-typed programs always generate well-typed programs.

Lastly, we note that since all top-level splices are evaluated during typing, compiled modules no longer contain top-level splices — though they may still contain splices inside quotations, as in the definition of `mpower`. Moreover, as macros are functions, after compilation, macros are always values. We will revisit this observation when we discuss module imports (§2.3 and 2.4).

2.3 Modules and Imports

In the `power` example, all definitions were defined in a single module. In practice, programmers organize programs into separate modules for maintainability, and import modules to use their definitions. However, this separation means that the levels of definitions are determined by the imported modules that they inhabit. For example, importing a let definition means that the definition can only be used at level 0 but not -1. Changing the level involves changing the let definition to a macro, which requires the source of the imported module to be under our control, which may not be the case. Further, we may wish to import and use a definition at both compile-time and runtime, and it would be tedious and error-prone to give essentially the same definition at both phases.

As an example, consider a module `Term`:

```
(* term.ml *)
type var = int
type term = Var of var | Lam of var * term | App of term * term
let id = ref 0
let fresh () = id := !id + 1; !id
```

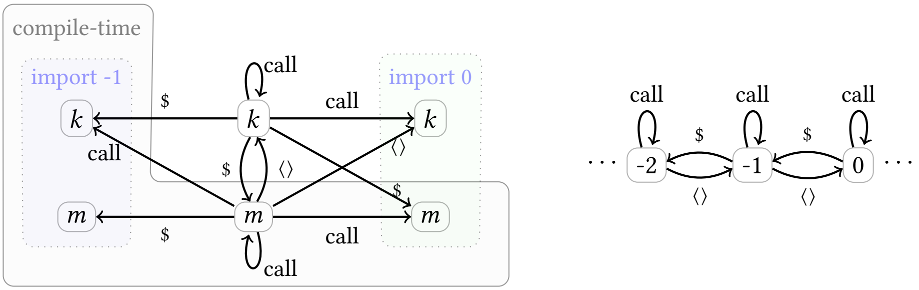


Fig. 2. Left: The call relation between definitions across modules. The module in the middle is the one currently being defined, and the one to the left is imported at level -1, and the one to the right is imported at level 0. Definitions in the gray box are compile-time computations. Right: The call relation between definitions across levels.

The type `term` defines a datatype for lambda terms. The definition `id` defines an integer reference, initialized to 0, and every call to `fresh` generates a fresh `variable`. Suppose we want to import this module to define a macro that builds a representation of function $\lambda x.\ \lambda y.\ x\ y$, calling `fresh` twice:

```
macro apply () = let x = fresh () in
  let y = fresh () in
  Lam (x, Lam (y, App (Var x, Var y)))
```

As written, this program is ill-staged, since the imported function `fresh` is a let definition defined at level 0, but used in the definition of the macro `apply` at level -1.

The module import system. To solve this issue, we allow importing modules at different levels, following Flatt [2002]. Specifically, when a module is imported at runtime (or level 0), all definitions are at the same level as if they were defined in the current module. Namely, let definitions are imported at level 0 and macros are imported at level -1. More interestingly, when a module is imported at compile-time (or level -1), all definitions will have their levels shifted by -1. That is, its let definitions are imported at level -1, while its macros are imported at level -2.

Therefore, importing `Term` at level -1 allows `apply` to call `fresh`. In MacoCaml, we write[3]:

```
module DT  = Term  (* Term is imported at level 0 as DT *)
module ~ST = Term  (* Term is imported at level -1 as ~ST *)
```

Now we can define the `apply` macro as follows[4]:

```
(* apply.ml *)
module ~ST = Term
macro apply () = let x = ~ST.fresh () in
  let y = ~ST.fresh () in
  ~ST.Lam (x, ~ST.Lam (y, ~ST.App (~ST.Var x, ~ST.Var y)))
```

Combining macros, staging, and modules. The module import system of Flatt [2002] fits extremely well into our framework. In particular, since let definitions and macros are simply bindings at different levels, importing modules at a specific level simply shifts the levels of the bindings. Well-stagedness, as before, is managed through levels, with staging annotations that can adjust levels in an expression. In this work, we focus on module imports at two levels, 0 and -1, which are the most practical and are sufficient to encode many interesting applications, but the essential idea can be generalized to support importing modules at more levels.

[3] MacoCaml also supports **open** Term and **open**~ Term (§5.1), in which case definitions need not be qualified.

[4] Types and datatypes are accessible from any level in MacoCaml, similar to, e.g., Xie et al. [2022]. In the future we would like to investigate giving levels to types and datatypes.

The updated call relation between definitions across modules is given on the left of Fig. 2. Definitions in the gray box are compile-time computations. That is, they can be spliced at top-level; notice how they are all targets of some splice arrow. Importing modules makes more definitions available for compile-time computations: let definitions can splice let definitions imported at level -1, and macros can splice macros imported at level -2. Note that the relation is not defined in an ad-hoc way; rather, it is *derived* following the rule of levels, as shown on the right of Fig. 2. In principle, these relations can extend to an arbitrary number of levels.

Finally, we note that only compiled modules can be imported. As all top-level splices are evaluated during compilation, imported modules never contain top-level splices. We have seen in Fig. 2 the call relation between definitions across modules. The call relation inside a *compiled* module can be simplified as the original call relation with splice edges removed to indicate the absence of top-level splices. An example is given as the module (a) at the top of Fig. 3; details about the figure can be ignored for now and will be introduced in the next section. Recall that the call relation features direct relations, and there can still be non-top-level splices in a compiled module.

2.4 Compile-time Side Effects

The `Term` and `Apply` example demonstrates another key aspect of our design: compile-time evaluation may perform side effects such as allocation and updates of references.

At the point when `apply` is spliced, we expect the reference in `id` from `Term` to have been initialized. Consequently, `id` ought to have been evaluated at that point. Indeed, as specified in Flatt [2002], a module imported at level -1 needs to be evaluated during compilation. The alternative semantics of evaluating imported definitions whenever they are spliced would produce quite unexpected results. In our example, every time `~ST.fresh` was called, the alternative semantics would allocate a new fresh reference for `id`, causing `~ST.fresh` to return `1` twice and making the final result $\lambda x.\ \lambda x.\ x\ x$, instead of $\lambda x.\ \lambda y.\ x\ y$, which is not the programmer's intention. By evaluating modules imported at level -1, the reference in `id` from `Term` will be properly initialized in the heap.

Compile-time evaluation with references means that compilation needs a *compile-time heap*. Moreover, we observe that the two calls to `fresh` in `apply` use the same reference `id`, and the state of the reference should be updated between the two calls — the first time `fresh ()` is called, it returns `1`, and the second time it returns `2`. This suggests that the compile-time heap needs to be threaded through compilation, and after every step of compile-time computation, the possibly updated heap will be used for the rest of the computation.

A potential problem with compile-time heaps. Compile-time heaps require some care to avoid problems. Consider:

```
module ~ST = Term
macro fresh_err () = ST.id := !ST.id + 1; <<!ST.id>>  (* error: id is bound at level -1
                                                         but used at level 0 *)
```

In this definition, `fresh_err` returns the result of `id` inside a quotation. Consider what would happen if this definition was accepted: the reference `id` evaluates to some location `loc` in the compile-time heap, and splicing `fresh_err` expands to code that refers to the location:

```
let err () = $(fresh_err ())  (* expands to (!loc) at compile-time*)
```

Now evaluating `err ()` will raise an error at runtime — we cannot find the location `loc` as the compile-time heap is no longer available! In this case, fortunately, the definition for `fresh_err` is rejected as ill-staged: `id` is imported at level −1, but used at level 0 inside the quotation. But in general, how can we ensure that a runtime computation will not depend on compile-time values?

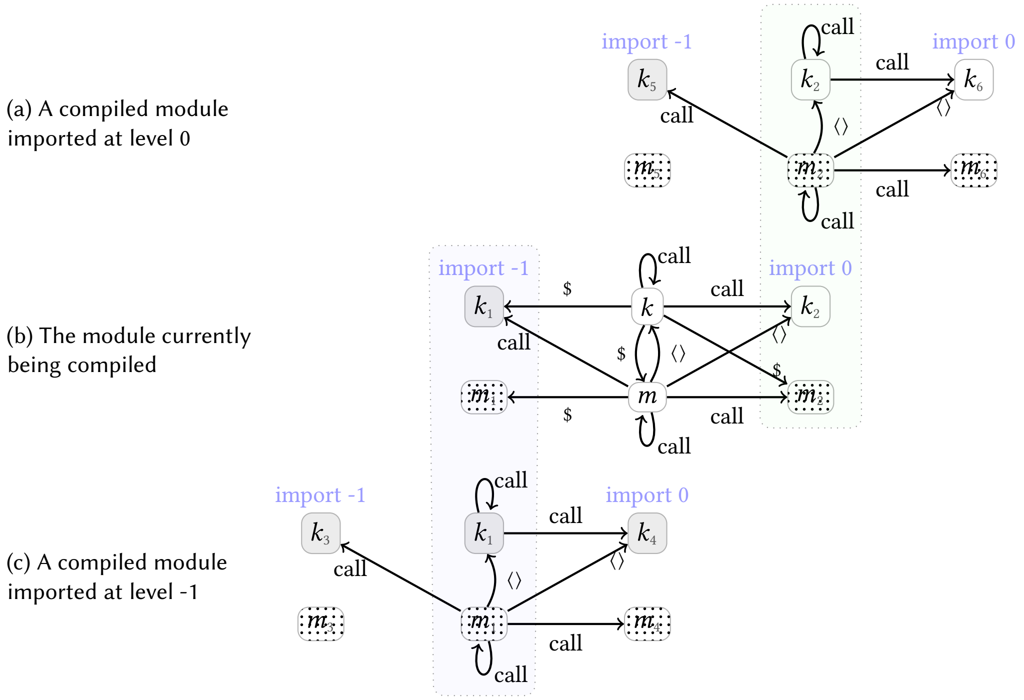
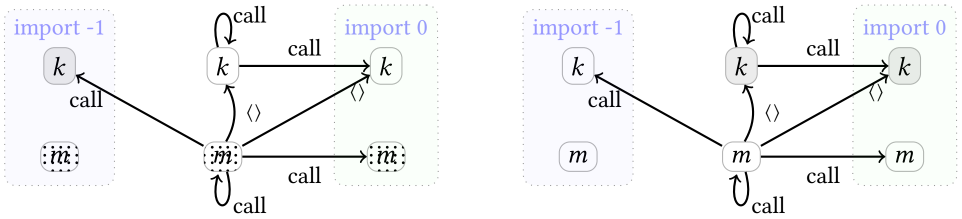


Fig. 3. Tower of imports: The call relation between definitions across module towers. The module (b) in the middle is the module currently being compiled, and it imports two modules, in green and purple respectively, with their call relations given in (a) and (c). This figure highlights 3 modules, but actually involves 7 modules: module (b), module (a) and its two imported modules at level 0 and -1 respectively, and module (c) and its two imported modules at level 0 and -1 respectively. Definitions in gray are evaluated during compilation. In practice, the tower can be arbitrarily high.



(a) Visiting a module will visit modules imported at level -1 and 0, and invoke the module imported at level -1.

(b) Invoking a module will evaluate k, and also invoke the module imported at level 0.

Fig. 4. Visiting and invoking: if the macro m is colored, it means the module needs to be visited; if the let definition k is colored, it means the module needs to be invoked.

Our solution: phase distinction of heaps. To answer this question, we formalize (integer) references and compile-time heaps in our calculi, and prove that well-staged expressions never need compile-time heaps after compilation (§4.5). As such, the compile-time heap can be safely discarded after compilation, which implies separate compilation of separate modules. This is the first time, to our best knowledge, such a result is formally established for compile-time computations.

The `id` and `fresh_err` example demonstrates the intuition behind this result: if a reference is created at compile-time, it can be called at compile-time, but its well-staged form can never be *captured* using quotations. Note that references may still appear inside quotations, such as `$<<!ST.id>>`, but as the top-level splice gets evaluated at compile time, this is equivalent to `!ST.id`.

Tower of modules. Side effects can happen across multiple modules. Suppose that we have another module, `Program`, that imports and uses `apply`:

```
(* program.ml *)
module A = Apply
let program = $(... A.apply ()...)  (* builds a large program *)
```

Before evaluating the call to `apply` in the splice, we must ensure that `Term` has been evaluated and thus `id` has been initialized. This kind of dependency can be arbitrarily deep: for example, importing `Program` in another module would also make it necessary to find and evaluate `Term`. For this reason, Flatt [2002] considers a *tower of modules.*

Fig. 3 presents a tower of modules, focusing on three modules: module (b) in the middle is the module currently being compiled, and it imports module (a) at level `-1` and module (c) at level `0`. As both (a) and (c) have been compiled, their call relations have no splice edges to indicate absence of top-level splices. And as they can themselves import modules, the tower can be arbitrarily high. Returning to our example, the definition `program` corresponds to a k in module (b), and `apply` is m_2 from module (a), and `fresh` and `id` are k_5 imported in module (a).

Now the goal is to find what definitions need to be evaluated during compilation. As all macros are values after compilation, macros themselves do not need to be evaluated, but they may require other definitions to be evaluated. All definitions colored in gray are let definitions of negative levels that need to be evaluated during compilation of module (b). The intuition behind evaluating those let definitions is as follows. Let definitions of negative levels can be spliced and thus evaluated at compile-time. We thus need to ensure that they have been evaluated before they can be spliced, so that all references have been initialized properly in the compile-time heap, just like `id` from `Term`.

As the tower of modules can be arbitrarily high, finding and evaluating those let definitions of negative levels are defined using two definitions, called *visiting* and *invoking* [Flatt 2002].

Visiting and invoking in MacoCaml. Fig. 4 shows the two processes. Visiting a module evaluates its compile-time computations, and invoking a module evaluates its runtime computations. We represent visiting by dotting macro definitions, and invoking by coloring let definitions gray.

Since macros are values, visiting a module (Fig. 4a) involves invoking modules imported at level `-1` and recursively visiting modules imported at level `-1` and `0`. Invoking a module (Fig. 4b) evaluates let definitions and recursively invokes modules imported at level `0`.

When compiling a module (module (b) in Fig. 3), we start by visiting as well as invoking modules imported at `-1` (corresponding to the coloring in module (c)), and also visiting modules imported at 0 (corresponding to the coloring in module (a)). For our example, the `Program` module visits `Apply` (m_2), which then in turn invokes `Term` and evaluates `id` (k_5).

Incorporating visiting and invoking into our framework introduces several differences to Flatt's design. The fact that macros in our system are always values after compilation makes visiting simpler in our system, since there is no need to evaluate macros during visiting. Moreover, as top-level splices have been evaluated for compiled modules, the value of m in the purple background of Fig. 4a will not be needed after compilation. As such, it is a design choice whether the m needs to be visited. In our system, we choose to visit m for its effects. Furthermore, we prove that phase distinction of heaps continues to hold in the presence of module towers, where a compile-time heap is threaded through visiting and invoking.

Compile-time and runtime phase distinction. We further prove that compile-time only computations do not interfere with runtime evaluation: when evaluating a compiled module, we can erase all macros and modules imported at compile-time. With such a result, we establish a full phase distinction between compile time and runtime.

Summary. We briefly summarize the MacoCaml notions introduced so far. Bindings are separated into let definitions at level 0 and macro definitions at level -1. Quotations and splices respectively increase and decrease the level of expressions, and can thus be used to cross phases in definitions. Modules can be imported at runtime or compile-time; in the latter case levels of imported definitions are shifted by -1. As imported modules can themselves import other modules, visiting and invoking traverse the tower of modules, evaluating definitions during module imports. Side-effects require care with compile-time heaps; we prove that compile-time heaps can be discarded after compilation. As we can see, phases, staging, and modules work together in MacoCaml. We have implemented MacoCaml in the OCaml compiler (§5) and compare our design to other systems in §6.

2.5 Larger Example: Print Formatted Data

Having introduced the key features of MacoCaml, in this section we present a more substantial example program. §5 describes larger-scale examples.

Consider defining a C-like `printf`-style function that takes a format and a sequence of arguments, and returns the formatted output. We start by encoding the format using a datatype:

```
(* fmt.ml *)
type (_, _) fmt = Int : (int → 'a, 'a) fmt
                | Lit : string → ('a, 'a) fmt
                | Cat : ('a, 'b) fmt * ('b, 'c) fmt → ('a, 'c) fmt
let (%) x y = Cat (x, y)
```

`fmt` is defined as a *generalized algebraic datatype* (GADT) taking two type parameters. Intuitively, in a format `('a, 'b) fmt`, `'a` prepends a number of `int →` to `b`, reflecting the number of integer arguments required by the format. There are three cases. The format either asks for an extra integer argument (**Int**), or is given a string (`Lit`), or concatenates two formats (`Cat`). The datatype can be easily extended to take more sorts of inputs. The function `%` is simply an infix alias for `Cat`.

For example, the following format corresponds to the C-style printf format string `"(%d, %d)"` that consumes a pair of integers.

```
let pair = Lit "(" % Int % Lit ", " % Int % Lit ")"  (* (int -> int -> 'b, 'b) fmt *)
```

One way to implement `printf` is through a continuation-passing-style (CPS) auxiliary function

```
(* printf.ml *)
open Fmt
let rec printk : type a b. (string → b) → (a, b) fmt → a
  = fun k fmt → match fmt with
      Int → fun s → k (string_of_int s)
    | Lit s → k s
    | Cat (l, r) → printk (fun x → printk (fun y → k (x ^ y)) r) l

let printf : type a. (a, string) fmt → a = fun fmt → printk (fun x → x) fmt
```

The `printk` function takes a continuation `k` of type `string → b` and a format `(a, b) fmt`, and constructs a function that takes as many integer arguments as the format specifies. If the format is **Int**, `printk` returns a function that takes an integer `s` and passes its string representation to `k`. If the format is `Lit s`, `s` is passed directly to `k`. For `Cat (l, r)`, `printk` processes `l`, binding its result to `x`, processes `r` binding its result to `y`, and finally catenates (`^`) the two strings, passing the result to `k`.

The definition gives us the desired behavior:

$$\begin{tikzcd}
\begin{array}{c}\textit{maco}\\(\S3)\end{array} & \begin{array}{c}\sigma_1 \mid \Omega \mid \Gamma \vdash \mathcal{M} : \Delta \rightsquigarrow \mathcal{M} \mid \sigma_2\\(\text{Fig. 7})\end{array} \arrow[r, shift right] & \begin{array}{c}\sigma_1 \mid \Omega \mid \Gamma \vdash \mathcal{S} : \phi \rightsquigarrow \mathcal{S} \mid \sigma_2\\(\text{Fig. 7})\end{array} \arrow[l, shift right] \arrow[r] \arrow[dl, bend left, "\begin{array}{c}(\textsc{st-importC})\\(\textsc{st-importR})\end{array}"'] \arrow[d] & \begin{array}{c}\sigma_1 \mid \Omega \mid \Gamma \vdash^{n}_{\star} e : \tau \rightsquigarrow e \mid \sigma_2\\(\text{Fig. 8})\end{array} \arrow[d, "(\textsc{codeGen})"] \\
\begin{array}{c}\textit{maco}_{\textit{core}}\\(\S4)\end{array} & \begin{array}{c}\sigma \mid \Gamma \vdash \mathcal{M} : \Delta\\ \sigma \mid \Gamma \vdash \mathcal{S} : \phi\\ \sigma \mid \Gamma \vdash^{n} e : \tau\\(\text{Fig. 11})\end{array} & \begin{array}{c}\sigma_1 \mid \mathcal{M}_1 \overset{\textit{visit}}{\rightsquigarrow} \mathcal{M}_2 \mid \sigma_2\\ \sigma_1 \mid \mathcal{S}_1 \overset{\textit{visit}}{\rightsquigarrow} \mathcal{S}_2 \mid \sigma_2\\ \sigma_1 \mid \mathcal{M}_1 \overset{\textit{invoke}}{\rightsquigarrow} \mathcal{M}_2 \mid \sigma_2\\(\text{Fig. 9})\end{array} \arrow[r] & \begin{array}{c}\sigma_1 \mid \Omega \mid \mathcal{M}_1 \longrightarrow \mathcal{M}_2 \mid \sigma_2\\ \sigma_1 \mid \Omega \mid \mathcal{S}_1 \longrightarrow \mathcal{S}_2 \mid \sigma_2\\ \sigma_1 \mid \Omega \mid e_1 \xrightarrow{n} e_2 \mid \sigma_2\\(\text{Fig. 12})\end{array}
\end{tikzcd}$$

Fig. 5. Key judgments of modules, structures, and expressions with their dependencies in the paper

```
let three_and_four = printf pair 3 4  (* "(3, 4)" *)
```

Defining `printk` as an interpreter of formats is inefficient, so it is useful to eliminate the interpretation overhead with macros. Here are macro definitions corresponding to `printk` and `printf`:

```
(* mprintf.ml *)
open ~Fmt
macro rec mprintk : type a b. (string expr → b expr) → (a, b) fmt → a expr
  = fun k fmt → match fmt with
      Int → << fun s → $(k <<string_of_int s>>) >>
    | Lit s → k (Expr.of_string s)
    | Cat (l, r) → mprintk (fun x → mprintk (fun y → k << $x ^ $y >>) r) l

macro mprintf : type a. (a, string) fmt → a expr = fun fmt → mprintk (fun x → x) fmt
```

Passing a format to `mprintf` generates code that takes exactly the right number of arguments and does not create or inspect the constructors of `fmt`:

```
let mpair = $(mprintf pair)
   (* fun s1 -> fun s2 -> "(" ^ string_of_int s1 ^ ", " ^ string_of_int s2 ^ ")" *)

let mthree_and_four = mpair 3 4  (* "(3, 4)" *)
```

3 A MACRO CALCULUS WITH STAGING AND MODULES

In this section, we present our source calculus *maco*, which forms the foundation for MacoCaml. Since the source can import compiled modules and enforce compile-time evaluation, the source judgments may depend on the core judgments to be defined in §4. Fig. 5 presents the key judgments and their dependencies in the paper. The reader is advised to refer back to this figure while reading the rest of the article, as what it depicts will gradually come to make sense.

Syntax. Fig. 6 presents the syntax of *maco*. Modules $\mathcal{M}$ include structures (**struct** $\mathcal{S}$ **end**), module variables (M), and qualified module variables ($p.M$). Structures $\mathcal{S}$ are either empty (•), or contain a sequence of items: source modules (**module** $M : \Delta = \mathcal{M}$), types (**type** $t = \tau$), let definitions (**def** $k = e$), and macros (**def**$^{\downarrow}$ $m = \lambda x : \tau.\ e$) that are always functions. We write **def** and **def**$^{\downarrow}$, instead of **let** and **macro**, to emphasize that macros are essentially definitions living at a shifted level (-1). Structure items also include explicit constructs for importing modules at level 0 (**import** $M : \Delta = \mathcal{M};$) or -1 (**import**$^{\downarrow}$ $M : \Delta = \mathcal{M};$). For clarity, imports in our system are qualified imports. Here $\mathcal{M}$ denotes a module in the core calculus, since it is only possible to import modules that have been compiled to the core. We will describe the core calculus in detail in §4. Here, we remark that the core calculus is essentially the source calculus after compile-time evaluation. Consequently, the core calculus does not have top-level splices, which will be evaluated during type-checking, but

module	$\mathcal{M}$	::=	$\textbf{struct}\ \mathcal{S}\ \textbf{end} \mid M \mid p.M$
structure item	$\mathcal{S}$	::=	$\bullet \mid \textbf{module}\ M : \Delta = \mathcal{M};\mathcal{S} \mid \textbf{type}\ t = \tau;\mathcal{S}$
		$\mid$	$\textbf{def}\ k = e;\mathcal{S} \mid \textbf{def}^{\downarrow}\ m = \lambda x : \tau.\ e;\mathcal{S}$
		$\mid$	$\textbf{import}\ M : \Delta = \mathcal{M};\mathcal{S} \mid \textbf{import}^{\downarrow}\ M : \Delta = \mathcal{M};\mathcal{S}$
path	p	::=	$M \mid p.M$
expression	e	::=	$i \mid \mathsf{unit} \mid x \mid \lambda x : \tau.\ e \mid e_1\ e_2 \mid k \mid p.k \mid m \mid p.m$
		$\mid$	$\mathsf{ref}\ e \mid !e \mid e_1 := e_2 \mid \langle e \rangle \mid \$e$
module type	Δ	::=	$\textbf{sig}\ \phi\ \textbf{end}$
structure type	ϕ	::=	$\bullet \mid M : (\Delta, n);\phi \mid t = \tau;\phi \mid k : \tau;\phi \mid m : \tau;\phi$
type	τ	::=	$\mathsf{Int} \mid \mathsf{Unit} \mid \tau_1 \longrightarrow \tau_2 \mid \mathsf{Ref}\ \tau \mid \mathsf{Code}\ \tau \mid t \mid p.t$
context	Γ	::=	$\bullet \mid \Gamma, M : (\Delta, n) \mid \Gamma, t = \tau \mid \Gamma, k : \tau \mid \Gamma, m : \tau \mid \Gamma, x : (\tau, n)$
heap	σ	::=	$\bullet \mid \sigma, l \mapsto v$
evaluation context	Ω	::=	$\bullet \mid \Omega; M = \mathcal{M} \mid \Omega; k = v \mid \Omega; m = v$

Fig. 6. Syntax in the source calculus *maco*. For clarity, we use colors to distinguish source and core, and syntax in blue and light blue denotes and refers to definitions in the core calculus.

additionally includes *locations*, which are values of evaluating references. A path p is a sequence of module variables that can be used to locate a definition inside (nested) modules.

Expressions e include literals i, unit unit, variables x, functions $\lambda x : \tau.\ e$, applications $e_1\ e_2$, definitions k and qualified definitions $p.k$, and macros m and qualified macros $p.m$. For simplicity, the calculus has only integer references[5], with reference creation ref e, access $!e$, and assignment $e_1 := e_2$, Lastly, $\langle e \rangle$ quotes an expression and $\$e$ splices an expression.

Types. Module types Δ include structure types $\textbf{sig}\ \phi\ \textbf{end}$, and structure types ϕ keep track of module types $M : (\Delta, n)$, with a level n to indicate the level the module is available at, type definitions $(t = \tau; \phi)$ to propagate type equivalence, and **def** and $\textbf{def}^{\downarrow}$ types ($k : \tau$ and $m : \tau$). Types τ include the integer type Int, the unit type Unit, functions $\tau_1 \longrightarrow \tau_2$, references Ref τ, code fragment Code τ, type variables t and type variables from a path $p.t$.

Contexts. The context Γ maps definitions, as well as local variable x, to their types and levels[6]. Throughout this paper, we assume all definitions have distinct names, so there is no shadowing.

Heaps σ and evaluation contexts Ω refer to definitions in the core calculus. σ maps a location l to a value v; while Ω maps modules, definitions, and macros to their definitions and values. Both contexts are needed in the source calculus for compile-time computations.

3.1 Typing Modules and Structures

The rules for typing modules and structure items are given in Fig. 7. The reader is advised to ignore σ, Ω, and the elaboration part ($\rightsquigarrow$) until §3.3.

The elaboration judgment $\sigma_1 \mid \Omega \mid \Gamma \vdash \mathcal{M} : \Delta \rightsquigarrow \mathcal{M} \mid \sigma_2$ reads: under the heap σ_1, the evaluation context Ω, and the type context Γ, the module $\mathcal{M}$ has type Δ, elaborates to a core module $\mathcal{M}$, updating the heap to σ_2. Rule M-STRUCT simply uses the typing rule for structure items. In rule M-MVAR, we get a module variable from the type context. Rule M-PMVAR type-checks a module variable from a path. In both cases, the module variable has level 0, as in programs modules themselves are always at level 0, in contrast to their names in paths, e.g., $M.k$ where M can be of level -1.

[5] References of code can cause *scope extrusion*; see §5.2 and §6 for more discussion.

[6] An alternative representation of the syntax is to use a uniform variable (e.g. x) for all definitions (e.g., macros are simply $x : (\tau, 0)$). We make the syntactic distinction for clarity.

$\boxed{\sigma_1 \mid \Omega \mid \Gamma \vdash \mathcal{M} : \Delta \leadsto \mathcal{M} \mid \sigma_2}$ *(Typing module)*

M-STRUCT
$$\frac{\sigma_1 \mid \Omega \mid \Gamma \vdash \mathcal{S} : \phi \leadsto \mathcal{S} \mid \sigma_2}{\sigma_1 \mid \Omega \mid \Gamma \vdash \mathbf{struct}\,\mathcal{S}\,\mathbf{end} : \mathbf{sig}\,\phi\,\mathbf{end} \leadsto \mathbf{struct}\,\mathcal{S}\,\mathbf{end} \mid \sigma_2}$$

M-MVAR
$$\frac{M : (\Delta, 0) \in \Gamma}{\sigma \mid \Omega \mid \Gamma \vdash M : \Delta \leadsto M \mid \sigma}$$

M-PMVAR
$$\frac{\Gamma \vdash^0 p : \mathbf{sig}\,\phi\,\mathbf{end} \qquad M : (\Delta, 0) \in \phi}{\sigma \mid \Omega \mid \Gamma \vdash p.M : \lceil \Delta \rceil^p \leadsto p.M \mid \sigma}$$

$\boxed{\Gamma \vdash^n p : \Delta}$ *(Typing path)*

P-MVAR
$$\frac{M : (\Delta, n) \in \Gamma}{\Gamma \vdash^n M : \Delta}$$

P-PMVAR
$$\frac{\Gamma \vdash^{n_1} p : \mathbf{sig}\,\phi\,\mathbf{end} \qquad M : (\Delta, n_2) \in \phi}{\Gamma \vdash^{n_1+n_2} p.M : \lceil \Delta \rceil^p}$$

$\boxed{\sigma_1 \mid \Omega \mid \Gamma \vdash \mathcal{S} : \phi \leadsto \mathcal{S} \mid \sigma_2}$ *(Typing structure item)*

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

ST-TYPE
$$\frac{\sigma_1 \mid \Omega \mid \Gamma, t = \tau \vdash \mathcal{S} : \phi \leadsto \mathcal{S} \mid \sigma_2}{\sigma_1 \mid \Omega \mid \Gamma \vdash (\mathbf{type}\, t = \tau; \mathcal{S}) : (t = \tau; \phi) \leadsto (\mathbf{type}\, t = \tau; \mathcal{S}) \mid \sigma_2}$$

ST-DEF
$$\frac{\sigma_1 \mid \Omega \mid \Gamma \vdash^0_c e : \tau \leadsto e \mid \sigma_2 \qquad \sigma_2 \mid \Omega \mid \Gamma, k : \tau \vdash \mathcal{S} : \phi \leadsto \mathcal{S} \mid \sigma_3}{\sigma_1 \mid \Omega \mid \Gamma \vdash (\mathbf{def}\, k = e; \mathcal{S}) : (k : \tau; \phi) \leadsto (\mathbf{def}\, k = e; \mathcal{S}) \mid \sigma_3}$$

ST-MACRO
$$\frac{\sigma_1 \mid \Omega \mid \Gamma \vdash^{-1}_c \lambda x : \tau.\, e : \tau \leadsto v \mid \sigma_2 \qquad \sigma_3 \mid \Omega; m = v \mid \Gamma, m : \tau \vdash \mathcal{S} : \phi \leadsto \mathcal{S} \mid \sigma_4}{\sigma_1 \mid \Omega \mid \Gamma \vdash (\mathbf{def}^{\downarrow}\, m = \lambda x : \tau.\, e; \mathcal{S}) : (m : \tau; \phi) \leadsto (\mathbf{def}^{\downarrow}\, m = v; \mathcal{S}) \mid \sigma_4}$$

ST-MODULE
$$\frac{\sigma_1 \mid \Omega \mid \Gamma \vdash \mathcal{M} : \Delta \leadsto \mathcal{M} \mid \sigma_2 \qquad \sigma_2 \mid \Omega; M = \mathcal{M} \mid \Gamma, M : (\Delta, 0) \vdash \mathcal{S} : \phi \leadsto \mathcal{S} \mid \sigma_3}{\sigma_1 \mid \Omega \mid \Gamma \vdash (\mathbf{module}\, M : \Delta = \mathcal{M}; \mathcal{S}) : (M : (\Delta, 0); \phi) \leadsto (\mathbf{module}\, M : \Delta = \mathcal{M}; \mathcal{S}) \mid \sigma_3}$$

ST-IMPORTR
$$\frac{\bullet \mid \bullet \vdash \mathcal{M} : \Delta \qquad \sigma_1 \mid \mathcal{M} \overset{visit}{\leadsto} \mathcal{M}_1 \mid \sigma_2 \qquad \sigma_2 \mid \Omega; M = \mathcal{M}_1 \mid \Gamma, M : (\Delta, 0) \vdash \mathcal{S} : \phi \leadsto \mathcal{S} \mid \sigma_3}{\sigma_1 \mid \Omega \mid \Gamma \vdash (\mathbf{import}\, M : \Delta = \mathcal{M}; \mathcal{S}) : (M : (\Delta, 0); \phi) \leadsto (\mathbf{import}\, M : \Delta = \mathcal{M}; \mathcal{S}) \mid \sigma_3}$$

ST-IMPORTC
$$\frac{\begin{array}{c}\bullet \mid \bullet \vdash \mathcal{M} : \Delta \qquad \sigma_1 \mid \mathcal{M} \overset{visit}{\leadsto} \mathcal{M}_1 \mid \sigma_2 \qquad \sigma_2 \mid \mathcal{M}_1 \overset{invoke}{\leadsto} \mathcal{M}_2 \mid \sigma_3 \\ \sigma_3 \mid \Omega; M = \mathcal{M}_2 \mid \Gamma, M : (\Delta, -1) \vdash \mathcal{S} : \phi \leadsto \mathcal{S} \mid \sigma_4\end{array}}{\sigma_1 \mid \Omega \mid \Gamma \vdash (\mathbf{import}^{\downarrow}\, M : \Delta = \mathcal{M}; \mathcal{S}) : (M : (\Delta, -1); \phi) \leadsto (\mathbf{import}^{\downarrow}\, M : \Delta = \mathcal{M}; \mathcal{S}) \mid \sigma_4}$$

Fig. 7. Typing modules and structures in *maco*

The path typing judgment $\Gamma \vdash^n p : \Delta$ says that under the type context Γ, the path p has type Δ with level n. The types and levels of modules variables are obtained from the context (rule P-MVAR). For a nested path $p.M$ (rule P-PMVAR), we first get the type ϕ and level n_1 of p, and get from ϕ the type (Δ, n_2) for the module. The return type is $\lceil \Delta \rceil^p$ with level $n_1 + n_2$. The notation $\lceil\ \rceil^p$ prefixes p to all variables that are defined in p. For example, if M_1 defines $t = \mathsf{Int}$ and a nested module $M_2 : (\mathbf{sig}\, k : t\, \mathbf{end})$, then $M_1.M_2$ has type $\lceil \mathbf{sig}\, k : t\, \mathbf{end} \rceil^{M_1} = \mathbf{sig}\, k : M_1.t\, \mathbf{end}$.

The judgment $\sigma_1 \mid \Omega \mid \Gamma \vdash \mathcal{S} : \phi \leadsto \mathcal{S} \mid \sigma_2$ type-checks a structure item. Rules ST-EMPTY and ST-TYPE are straightforward. For definitions $k = e$ (rule ST-DEF), we type-check e at level 0 (expression typing is explained in the next section) and put $k : \tau$ in the context Γ for type-checking $\mathcal{S}$.

Rule ST-MACRO type-checks macros $m = e$. We type-check the macro definition at level -1 and add its type $m : \tau$ to the context Γ for checking $\mathcal{S}$.

Rules ST-MODULE, ST-IMPORTR, and ST-IMPORTC type-check modules and imports. When we define or **import** a module, the module is at level 0. In contrast, when we **import**$^{\downarrow}$ a module, the

$$\boxed{\sigma_1 \mid \Omega \mid \Gamma \vDash^{n}_{\star} e : \tau \rightsquigarrow e \mid \sigma_2} \qquad \boxed{\text{compiler mode} \quad \star ::= c \mid s \mid q} \qquad \textit{(Typing expression)}$$

LIT
$$\frac{}{\sigma \mid \Omega \mid \Gamma \vDash^{n}_{\star} i : \mathsf{Int} \rightsquigarrow i \mid \sigma}$$

UNIT
$$\frac{}{\sigma \mid \Omega \mid \Gamma \vDash^{n}_{\star} \mathsf{unit} : \mathsf{Unit} \rightsquigarrow \mathsf{unit} \mid \sigma}$$

VAR
$$\frac{x : (\tau, n) \in \Gamma}{\sigma \mid \Omega \mid \Gamma \vDash^{n}_{\star} x : \tau \rightsquigarrow x \mid \sigma}$$

KVAR
$$\frac{k : \tau \in \Gamma}{\sigma \mid \Omega \mid \Gamma \vDash^{0}_{\star} k : \tau \rightsquigarrow k \mid \sigma}$$

MACRO
$$\frac{m : \tau \in \Gamma}{\sigma \mid \Omega \mid \Gamma \vDash^{-1}_{\star} m : \tau \rightsquigarrow m \mid \sigma}$$

PKVAR
$$\frac{\Gamma \vdash^{n} p : \mathbf{sig}\ \phi\ \mathbf{end} \qquad k : \tau \in \phi}{\sigma \mid \Omega \mid \Gamma \vDash^{n}_{\star} p.k : \lceil \tau \rceil^{p} \rightsquigarrow p.k \mid \sigma}$$

PMACRO
$$\frac{\Gamma \vdash^{n} p : \mathbf{sig}\ \phi\ \mathbf{end} \qquad m : \tau \in \phi}{\sigma \mid \Omega \mid \Gamma \vDash^{n-1}_{\star} p.m : \lceil \tau \rceil^{p} \rightsquigarrow p.m \mid \sigma}$$

ABS
$$\frac{\Gamma \vdash \tau_1 \qquad \sigma_1 \mid \Omega \mid \Gamma, x : (\tau_1, n) \vDash^{n}_{\star} e : \tau_2 \rightsquigarrow e \mid \sigma_2}{\sigma_1 \mid \Omega \mid \Gamma \vDash^{n}_{\star} \lambda x : \tau_1 .\ e : \tau_1 \to \tau_2 \rightsquigarrow \lambda x : \tau .\ e \mid \sigma_2}$$

APP
$$\frac{\sigma_1 \mid \Omega \mid \Gamma \vDash^{n}_{\star} e_1 : \tau_1 \to \tau_2 \rightsquigarrow e_1 \mid \sigma_2 \qquad \sigma_2 \mid \Omega \mid \Gamma \vDash^{n}_{\star} e_2 : \tau_1 \rightsquigarrow e_2 \mid \sigma_3}{\sigma \mid \Omega \mid \Gamma \vDash^{n}_{\star} e_1\, e_2 : \tau_2 \rightsquigarrow e_1\, e_2 \mid \sigma_3}$$

REF
$$\frac{\sigma_1 \mid \Omega \mid \Gamma \vDash^{n}_{\star} e : \mathsf{Int} \rightsquigarrow e \mid \sigma_2}{\sigma_1 \mid \Omega \mid \Gamma \vDash^{n}_{\star} \mathsf{ref}\ e : \mathsf{Ref\ Int} \rightsquigarrow \mathsf{ref}\ e \mid \sigma_2}$$

GET
$$\frac{\sigma_1 \mid \Omega \mid \Gamma \vDash^{n}_{\star} e : \mathsf{Ref\ Int} \rightsquigarrow e \mid \sigma_2}{\sigma_1 \mid \Omega \mid \Gamma \vDash^{n}_{\star} !e : \mathsf{Int} \rightsquigarrow !e \mid \sigma_2}$$

SET
$$\frac{\begin{array}{c}\sigma_1 \mid \Omega \mid \Gamma \vDash^{n}_{\star} e_1 : \mathsf{Ref\ Int} \rightsquigarrow e_1 \mid \sigma_2 \\ \sigma_2 \mid \Omega \mid \Gamma \vDash^{n}_{\star} e_2 : \mathsf{Int} \rightsquigarrow e_2 \mid \sigma_3\end{array}}{\sigma_1 \mid \Omega \mid \Gamma \vDash^{n}_{\star} e_1 := e_2 : \mathsf{Unit} \rightsquigarrow e_1 := e_2 \mid \sigma_3}$$

EQ
$$\frac{\begin{array}{c}\sigma_1 \mid \Omega \mid \Gamma \vDash^{n}_{\star} e : \tau_1 \rightsquigarrow e \mid \sigma_2 \\ \Gamma \vdash \tau_1 \approx \tau_2\end{array}}{\sigma_1 \mid \Omega \mid \Gamma \vDash^{n}_{\star} e : \tau_2 \rightsquigarrow e \mid \sigma_2}$$

QUOTE
$$\frac{\sigma_1 \mid \Omega \mid \Gamma \vDash^{n+1}_{q} e : \tau \rightsquigarrow e \mid \sigma_2}{\sigma_1 \mid \Omega \mid \Gamma \vDash^{n}_{c \vee s} \langle e \rangle : \mathsf{Code}\ \tau \rightsquigarrow \langle e \rangle \mid \sigma_2}$$

SPLICE
$$\frac{\sigma_1 \mid \Omega \mid \Gamma \vDash^{n-1}_{s} e : \mathsf{Code}\ \tau \rightsquigarrow e \mid \sigma_2}{\sigma_1 \mid \Omega \mid \Gamma \vDash^{n}_{q} \$e : \tau \rightsquigarrow \$e \mid \sigma_2}$$

CODEGEN
$$\frac{\sigma_1 \mid \Omega \mid \Gamma \vDash^{n-1}_{s} e : \mathsf{Code}\ \tau \rightsquigarrow e \mid \sigma_2 \qquad \sigma_2 \mid \Omega \mid e \xrightarrow{0}{}^{*} \langle v^1 \rangle \mid \sigma_3}{\sigma_1 \mid \Omega \mid \Gamma \vDash^{n}_{c} \$e : \tau \rightsquigarrow v^1 \mid \sigma_3}$$

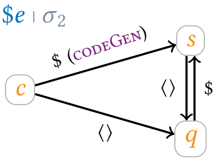


Fig. 8. Typing expressions with compile-time code generation in *maco*

module is at level -1. Importing a module also requires the module to be well-typed in the core under an empty heap ($\bullet \mid \bullet \vdash \mathcal{M} : \Delta$), making *compilation independence* [Culpepper et al. 2007] obvious that compilation of a module does not depend on side effects that occurred during the compilation of imported modules.

3.2 Typing Expressions

Fig. 8 presents the typing rules for expressions. The judgment $\sigma_1 \mid \Omega \mid \Gamma \vDash^{n}_{\star} e : \tau \rightsquigarrow e \mid \sigma_2$ reads: under the heap σ_1, the evaluation context Ω, and the type context Γ, the expression e has type τ under level n and *mode* $\star$, elaborates to e, and updates the heap to σ_2. The definition of compiler modes appears at the top of the figure. The mode is only significant for staging annotations, and is explained with the staging rules.

The first three rules are straightforward. Rule KVAR says that a definition variable k is well-typed only at level 0. Similarly, rule MACRO says that a macro m is well-typed only at level -1. Definitions from paths can have lower levels (rules PKVAR and PMACRO). Thus we first get the level n of the path p. Then $p.k$ is well-typed at level n, and $p.m$ is well-typed at level $n - 1$. In both cases, the return type is $\lceil \tau \rceil^{p}$.

Rule ABS introduces the binder x at level n, so later x can be used only at level n. The notation $\Gamma \vdash \tau$ checks that all type variables in τ are bound. Rule APP is self-explanatory. Rules REF, GET, and SET are standard typing rules for creating, getting and setting a reference. Rule EQ says that if e has

$$\boxed{\sigma_1 \mid \mathcal{M}_1 \overset{visit}{\leadsto} \mathcal{M}_2 \mid \sigma_2} \qquad \textit{(Visiting)}$$

V-STRUCT

$$\frac{\sigma_1 \mid \mathcal{S} \overset{visit}{\leadsto} \mathcal{S}' \mid \sigma_2}{\sigma_1 \mid \textbf{struct}\ \mathcal{S}\ \textbf{end} \overset{visit}{\leadsto} \textbf{struct}\ \mathcal{S}'\ \textbf{end} \mid \sigma_2}$$

$$\boxed{\sigma_1 \mid \mathcal{M}_1 \overset{invoke}{\leadsto} \mathcal{M}_2 \mid \sigma_2} \qquad \textit{(Invoking)}$$

I-STRUCT

$$\frac{\sigma_1 \mid \bullet \mid \mathcal{M} \longrightarrow^* \mathcal{M}_v \mid \sigma_2}{\sigma_1 \mid \mathcal{M} \overset{invoke}{\leadsto} \mathcal{M}_v \mid \sigma_2}$$

$$\boxed{\sigma_1 \mid \mathcal{S}_1 \overset{visit}{\leadsto} \mathcal{S}_2 \mid \sigma_2} \qquad \textit{(Visiting structure)}$$

V-IMPORTC

$$\frac{\sigma_1 \mid \mathcal{M} \overset{visit}{\leadsto} \mathcal{M}' \mid \sigma_2 \quad \sigma_2 \mid \mathcal{M}' \overset{invoke}{\leadsto} \mathcal{M}'' \mid \sigma_3 \\ \sigma_3 \mid \mathcal{S} \overset{visit}{\leadsto} \mathcal{S}' \mid \sigma_4}{\begin{array}{c}\sigma_1 \mid \textbf{import}^{\downarrow}\ M : \Delta = \mathcal{M}; \mathcal{S} \\ \overset{visit}{\leadsto}\ \textbf{import}^{\downarrow}\ M : \Delta = \mathcal{M}''; \mathcal{S}' \mid \sigma_4\end{array}}$$

V-IMPORTR

$$\frac{\sigma_1 \mid \mathcal{M} \overset{visit}{\leadsto} \mathcal{M}' \mid \sigma_2 \\ \sigma_2 \mid \mathcal{S} \overset{visit}{\leadsto} \mathcal{S}' \mid \sigma_3}{\begin{array}{c}\sigma_1 \mid \textbf{import}\ M : \Delta = \mathcal{M}; \mathcal{S} \\ \overset{visit}{\leadsto}\ \textbf{import}\ M : \Delta = \mathcal{M}'; \mathcal{S}' \mid \sigma_3\end{array}}$$

Fig. 9. Key rules of visiting and invoking

type τ_1, and τ_1 is *equivalent* ($\approx$) to τ_2, then e can also be typed at τ_2. The type equivalence judgment is standard [Leroy 1994] and is put in the appendix. As an example, if one module M defines t = Int and $k : t$, then $1 + M.k$ is well-typed, as $M.k : M.t$ (pkvar), $M.t \approx$ Int, and thus $M.k$: Int (eq).

Typing staging annotations. The final three rules type-check staging annotations. Rule quote says that if $\langle e \rangle$ has type τ at level $n + 1$, then $\langle e \rangle$ has type Code τ at level n. Dually, rules splice and codeGen say that if e has type Code τ at level $n - 1$, then $\$e$ has type τ at level n; the two rules apply under different *compiler modes*.

The *compiler mode* $\star$ manages compile-time code generation, and is similar to the *typing states* in Template Haskell [Sheard and Jones 2002]. The transition between the three modes is given at the bottom of Fig. 8. They work as follows. As described in §2.2, compile-time evaluation happens inside *top-level* splices, i.e. splices that are not inside quotations. When we first enter into the judgment for typing expressions and macros, as in rules st-def and st-macro during structure typing, the compiler is in mode c. If the compiler then encounters a splice, rule codeGen applies. The rule switches to mode s to type-check the spliced body, and then forces evaluating the elaboration result (§3.3). With compiler modes, we ensure that compile-time code generation happens only inside top-level splices in rule codeGen. In contrast, if the compiler encounters a quotation, then it goes to mode q (rule quote). Rules splice and quote switch back and forth between mode s and q, but the compiler can never go back to c.[7]

Notably, our rule codeGen generalizes that of Sheard and Jones [2002]: our rule applies at any level, reflecting the fact that top-level splices can occur in let definitions at level 0 (rule st-def) as well as in macros at level -1 (rule st-macro), while Sheard and Jones [2002] required (and only needed) the level to be at 0.

3.3 Elaboration and Compile-time Evaluation

As we will see, the source syntax is a subset of the core syntax. The elaboration part ($\leadsto$) in Fig. 7 and 8 mostly does nothing except for two kinds of compile-time computation:

(1) Visiting and invoking imported modules in rules st-importR and st-importC;

[7] It is easy to extend the model transition system to allow nested quotations, but without nested splices they are not that useful. The key challenge with nested splices is that ideally we would like expressions like $(\$(\$e_1))$ to be evaluated before $(\$e_2)$, which is difficult to model during typing. Typed Template Haskell (TTH) [Xie et al. 2022] solves this issue by lifting splices to the top-level following levels, so e_1 will be put before e_2. However, compile-time evaluation is not formalized in TTH beyond lifting splices to the top-level. It would be interesting to integrate splice lifting in our system in the future. Other languages including Template Haskell [Sheard and Jones 2002] and Scala [Stucki et al. 2018] disallow nested splices.

$$\begin{array}{lcl} \Omega & = & m = \lambda x : \mathsf{Ref\ Int}.\ \mathsf{if}\ (!x) = 0\ \mathsf{then}\ \langle -1\rangle\ \mathsf{else}\ \langle 1\rangle \\ \Gamma & = & m : \mathsf{Ref\ Int} \longrightarrow \mathsf{Code\ Int} \end{array}$$

$$\cfrac{\cfrac{\cdots \qquad \cfrac{\cfrac{\cfrac{\cdots}{\bullet \mid \Omega \mid \Gamma, x : (\mathsf{Ref\ Int}, -1) \vdash^{-1}_{s} x := 1; m\,x : \mathsf{Code\ Int} \rightsquigarrow x := 1; m\,x \mid \bullet}\ \textsc{app}}{\bullet \mid \Omega \mid \Gamma, x : (\mathsf{Ref\ Int}, -1) \vdash^{0}_{q} \$(x := 1; m\,x) : \mathsf{Int} \rightsquigarrow \$(x := 1; m\,x) \mid \bullet \quad ①}\ \textsc{splice}}{\bullet \mid \Omega \mid \Gamma, x : (\mathsf{Ref\ Int}, -1) \vdash^{-1}_{s} \langle \$(x := 1; m\,x)\rangle : \mathsf{Code\ Int} \rightsquigarrow \langle \$(x := 1; m\,x)\rangle \mid \bullet}\ \textsc{quote}}{\bullet \mid \Omega \mid \Gamma \vdash^{-1}_{s} x \leftarrow \mathsf{ref}\ 0; \langle \$(x := 1; m\,x)\rangle : \mathsf{Code\ Int} \rightsquigarrow x \leftarrow \mathsf{ref}\ 0; \langle \$(x := 1; m\,x)\rangle \mid \bullet}\ \textsc{app} \qquad \begin{array}{c} \bullet \mid \Omega \mid x \leftarrow \mathsf{ref}\ 0; \langle \$(x := 1; m\,x)\rangle \xrightarrow{0}{}^{*} \langle 1\rangle \mid l \mapsto 1 \\ \vdots \end{array}}{\bullet \mid \Omega \mid \Gamma \vdash^{0}_{c} \$(x \leftarrow \mathsf{ref}\ 0; \langle \$(x := 1; m\,x)\rangle) : \mathsf{Int} \rightsquigarrow 1 \mid l \mapsto 1 \quad ②}\ \textsc{codeGen}$$

Fig. 10. Example: typing with compile-time code generation in *maco*. For clarity, we use the following syntactic sugar: (1) $x \leftarrow e_1; e_2$ for $(\lambda x : \_.\ e_2)\, e_1$, and (2) $e_1; e_2$ for $(\lambda\_ : \_.\ e_2)\, e_1$. We assume if-expressions and the equality operator = on integers. We omit some uninteresting details for space reasons.

(2) Compile-time code generation in rule codeGen.

Heaps and evaluation contexts. To support compile-time computations, typing judgments take as input a heap σ and an evaluation context Ω. Heaps keep track of references, whose values are stored inside *locations*. The typing judgment takes in a heap σ_1 and updates it to σ_2.

The evaluation context Ω stores definitions needed for evaluation, and is extended in four rules: rule st-macro when typing macros, and rules st-module, st-importR, and st-importC when typing (imported) modules. Notably, in source typing the evaluation context stores only *compile-time* definitions, hence we ignore definitions of k in rule st-def as the definitions will not be needed during compile-time evaluation.

Visiting and invoking modules. As explained in §2.4, when we import a module at level 0, we need to *visit* the module (rule st-importR), and when we import a module at level -1, we need to *visit* as well as *invoke* the module (rule st-importC). Intuitively, visiting and invoking ensure that compile-time computations from imported modules are evaluated for their side effects (e.g., all compile-time references are initialized). Note that in both cases, the elaborated module is the original module, as visiting and invoking do not affect runtime computations.

We present the key rules for visiting and invoking in Fig. 9; for space reasons, the complete rules are in the appendix. We have two judgments for visiting and invoking a module respectively:

- rule v-struct: Visiting a module will traverse its structure to find imported modules, and visit and invoke **import**$^{\downarrow}$ed modules (rule v-importC) and visit **import**ed modules (rule v-importR).
- rule i-struct: Invoking a module will simply *evaluate* it; rules for evaluating a module are defined in the core calculus (§4).

Compile-time code generation. Rule codeGen is where compile-time code generation happens inside top-level splices. After elaborating the spliced expression to e, the rule evaluates e into a value $\langle v^1\rangle$. *Level-annotated values* (v^1) are introduced in §4.1; for now, it suffices to know that $\langle v^1\rangle$ is a quotation value that cannot be further reduced. The rule then removes the quotation and inserts v^1 as the elaboration result.

3.4 Example

Fig. 10 presents an example for typing and compile-time code generation. The expression

$$\$(x \leftarrow \mathsf{ref}\ 0; \langle \$(x := 1; m\,x)\rangle)$$

is a top-level splice. Inside the splice, we first create a reference x. Then inside a quotation and another (non-top-level) splice, we set x to 1, and call the macro m with x.

$$
\begin{array}{llcl}
\text{module} & \mathcal{M} & ::= & \textbf{struct}\ \mathcal{S}\ \textbf{end} \mid M \mid p.M \\
\text{structure item} & \mathcal{S} & ::= & \bullet \mid \textbf{module}\ M : \Delta = \mathcal{M}; \mathcal{S} \mid \textbf{type}\ t = \tau; \mathcal{S} \\
 & & \mid & \textbf{def}\ k = e^0; \mathcal{S} \mid \textbf{def}^{\downarrow}\ m = v^0; \mathcal{S} \\
 & & \mid & \textbf{import}\ M : \Delta = \mathcal{M}; \mathcal{S} \mid \textbf{import}^{\downarrow}\ M : \Delta = \mathcal{M}; \mathcal{S} \\
\text{path} & p & ::= & M \mid p.M \\
\text{expression} & e & ::= & i \mid \mathsf{unit} \mid x \mid \lambda x : \tau.\ e \mid e_1\ e_2 \mid k \mid p.k \mid m \mid p.m \\
 & & \mid & \mathsf{ref}\ e \mid\ !e \mid e_1 := e_2 \mid \langle e \rangle \mid \$e \mid l
\end{array}
$$

$$
\boxed{\sigma \mid \Gamma \vdash \mathcal{M} : \Delta} \quad \boxed{\Gamma \vdash^{n} p : \Delta} \quad \boxed{\sigma \mid \Gamma \vdash \mathcal{S} : \phi} \quad \boxed{\sigma \mid \Gamma \Vdash^{n} e : \tau} \qquad \textit{(Typing expression)}
$$

$$
\textsc{C-Lit}\ \frac{}{\sigma \mid \Gamma \Vdash^{n} i : \mathsf{Int}}
\qquad
\textsc{C-Unit}\ \frac{}{\sigma \mid \Gamma \Vdash^{n} \mathsf{unit} : \mathsf{Unit}}
\qquad
\textsc{C-Var}\ \frac{x : (\tau, n) \in \Gamma}{\sigma \mid \Gamma \Vdash^{n} x : \tau}
\qquad
\textsc{C-Abs}\ \frac{\Gamma \vdash \tau_1 \qquad \sigma \mid \Gamma, x : (\tau_1, n) \Vdash^{n} e : \tau_2}{\sigma \mid \Gamma \Vdash^{n} \lambda x : \tau_1.\ e : \tau_1 \longrightarrow \tau_2}
$$

$$
\textsc{C-App}\ \frac{\sigma \mid \Gamma \Vdash^{n} e_1 : \tau_1 \longrightarrow \tau_2 \qquad \sigma \mid \Gamma \Vdash^{n} e_2 : \tau_1}{\sigma \mid \Gamma \Vdash^{n} e_1\ e_2 : \tau_2}
\qquad
\textsc{C-KVar}\ \frac{k : \tau \in \Gamma}{\sigma \mid \Gamma \Vdash^{0} k : \tau}
\qquad
\textsc{C-Macro}\ \frac{m : \tau \in \Gamma}{\sigma \mid \Gamma \Vdash^{-1} m : \tau}
$$

$$
\textsc{C-PKVar}\ \frac{\Gamma \vdash^{n} p : \textbf{sig}\ \phi\ \textbf{end} \qquad k : \tau \in \phi}{\sigma \mid \Gamma \Vdash^{n} p.k : \lceil \tau \rceil^{p}}
\qquad
\textsc{C-PMacro}\ \frac{\Gamma \vdash^{n} p : \textbf{sig}\ \phi\ \textbf{end} \qquad m : \tau \in \phi}{\sigma \mid \Gamma \Vdash^{n-1} p.m : \lceil \tau \rceil^{p}}
\qquad
\textsc{C-Ref}\ \frac{\sigma \mid \Gamma \Vdash^{n} e : \mathsf{Int}}{\sigma \mid \Gamma \Vdash^{n} \mathsf{ref}\ e : \mathsf{Ref\ Int}}
$$

$$
\textsc{C-Assign}\ \frac{\sigma \mid \Gamma \Vdash^{n} e_1 : \mathsf{Ref\ Int} \qquad \sigma \mid \Gamma \Vdash^{n} e_2 : \mathsf{Int}}{\sigma \mid \Gamma \Vdash^{n} e_1 := e_2 : \mathsf{Unit}}
\qquad
\textsc{C-Deref}\ \frac{\sigma \mid \Gamma \Vdash^{n} e : \mathsf{Ref\ Int}}{\sigma \mid \Gamma \Vdash^{n} !e : \mathsf{Int}}
\qquad
\textsc{C-Quote}\ \frac{\sigma \mid \Gamma \Vdash^{n+1} e : \tau}{\sigma \mid \Gamma \Vdash^{n} \langle e \rangle : \mathsf{Code}\ \tau}
$$

$$
\textsc{C-Splice}\ \frac{\sigma \mid \Gamma \Vdash^{n-1} e : \mathsf{Code}\ \tau}{\sigma \mid \Gamma \Vdash^{n} \$e : \tau}
\qquad
\textsc{C-Eq}\ \frac{\sigma \mid \Gamma \Vdash^{n} e : \tau_1 \qquad \Gamma \vdash \tau_1 \approx \tau_2}{\sigma \mid \Gamma \Vdash^{n} e : \tau_2}
\qquad
\textsc{C-Loc}\ \frac{l \in \sigma}{\sigma \mid \Gamma \Vdash^{n} l : \mathsf{Ref\ Int}}
$$

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

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

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

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

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

Fig. 11. Syntax and typing in the core calculus $maco_{core}$

The example demonstrates several interesting aspects. First, it shows how the level and the mode change during typing. Second, at ①, we apply rule SPLICE rather than rule CODEGEN, as the splice is *not* a top-level splice. Applying rule CODEGEN here, in fact, would go wrong — if rule CODEGEN is applied, we would have no way to evaluate $x := 1$ since x has not been initiated yet! Lastly, at ②, we evaluate the elaborated expression inside the splice. The evaluation result $\langle 1 \rangle$ suggests that the expression $x := 1; m\ x$ generated at ① has also been evaluated, but this time with x properly initialized. We will see how evaluation works in the core calculus in the next section.

4 COMPILATION TARGET WITH DYNAMIC SEMANTICS

In this section, we present $maco_{core}$, which is the compilation target for *maco*.

4.1 Syntax and Typing

Fig. 11 presents the syntax, typing rules, and level-annotated expressions and values for $\mathit{maco}_{\mathit{core}}$.

Syntax. The syntax in the core is mostly the same as the source. In structure items $\mathcal{S}$, the definitions **def** and $\textbf{def}^{\downarrow}$ have an extra syntactic condition e^0 and v^0 (explained below), which ensures that *definitions have no top-level splices* after compilation. Modules and imported modules are now both of form $\mathcal{M}$, but we still distinguish them as imported modules need to be type-checked under an empty context for separate compilation. Expressions e include an additional construct, *locations* l, which are the values of references. We omit the definition for types (Δ, ϕ, τ) and contexts (Γ, σ, Ω), which are exactly the same as in the source calculus (Fig. 6).

Typing. Typing rules in the core has no compile-time evaluation, and thus judgments have no evaluation context, output heap, compile mode, or elaboration result. The input heap σ is used to type-check locations (rule c-loc), where a location l is well-typed only if it is bound in the heap. We omit the typing rules for modules, paths, and structure items, as they are otherwise the same as the corresponding typing part in the source calculus (Fig. 7).

The judgment $\sigma \mid \Gamma \vdash^{n} e : \tau$ reads: under the heap σ and the type context Γ, the expression e has type τ at level n. Most rules are self-explanatory. Rule c-loc, as mentioned above, type-checks locations. As references are always of integers, locations have type Ref Int.

Level-annotated expressions and values. We disallow top-level splices in the core calculus, enforcing that restriction by means of *level-annotated expressions* [Calcagno et al. 2003b; Taha et al. 1998] The notion e^n means that e is an expression at level n, where $n \geq 0$. Importantly, a splice $\$e$ is an expression at only positive levels $n + 1$. Thus, e^0 in the structure syntax ensures that expressions do not have top-level splices. In other words, there are no negative level-annotated expressions.

The level-annotated *values* v^n, where v is a subset of expressions e, means that v is a value at level n. Values v^0 include literals i, units unit, locations l, and lambdas $\lambda x : \tau.\ e$ and quotations $\langle e \rangle$ only if e^0, i.e., e is an expression at level 0 that has no top-level splices. This suggests that evaluation can happen *inside* lambdas and quotations, as we will see. Values 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 evaluate at level $n + 1$, and thus it can only have up to n nested splices, which is exactly expressions of e^n. For example, if $\$e$ is an expression at level 1, then it is a value at level 2.

Notations: we often write simply v for v^0. We write partially annotated expression (and values), such as ${e_1}^n\ e_2$ to mean that e_1 is an expression at level n but there is no level restriction on e_2.

4.2 Dynamic Semantics

Fig. 12 presents the dynamic semantics for the core calculus.

A module value $\mathcal{M}_v$ contains a structure value $\mathcal{S}_v$. A structure is a value $\mathcal{S}_v$ when its **module**s, **def**s, $\textbf{def}^{\downarrow}$s, and **import**ed modules are values. Note that $\textbf{import}^{\downarrow}$ed modules can be any $\mathcal{M}$, as they are compile-time only computations that do not evaluate in the dynamic semantics.

The judgment $\sigma_1 \mid \Omega \mid \mathcal{M}_1 \longrightarrow \mathcal{M}_2 \mid \sigma_2$ reads: under heap σ_1 and evaluation context Ω, the module $\mathcal{M}_1$ evaluates to $\mathcal{M}_2$, updating the heap to σ_2. Rule ev-m-struct evaluates the structure. Rules ev-m-mvar and ev-m-pmvar get the module definition from the context. We use the notation $p.M = \mathcal{M} \in \Omega$ to mean that we can get the definition of M inside Ω following the path p. Recall that the notation $\lceil\ \rceil^p$ prefixes p to all variables that are defined in p.

The judgment $\sigma_1 \mid \Omega \mid \mathcal{S}_1 \longrightarrow \mathcal{S}_2 \mid \sigma_2$ evaluates structures. For **def** $k = e$, we first evaluate e to a value (rule ev-st-def1), and then adds $k = v$ in the evaluation context to evaluate the rest of the structure (rule ev-st-def2). Type definitions are ignored in the dynamic semantics (rule ev-st-type). Macros are *not* added to the evaluation context (rule ev-st-macro), making it explicit that macros are compile-time only computations. For space reasons, we omit the evaluation rules for modules.

$$
\begin{array}{llll}
\text{module value} & \mathcal{M}_v & ::= & \bullet \mid \textbf{struct}\ \mathcal{S}_v\ \textbf{end} \\
\text{structure value} & \mathcal{S}_v & ::= & \bullet \mid \textbf{module}\ M : \Delta = \mathcal{M}_v; \mathcal{S}_v \mid \textbf{type}\ t = \tau; \mathcal{S}_v \mid \textbf{def}\ k = v; \mathcal{S}_v \\
 & & \mid & \textbf{def}^{\downarrow}\ m = v; \mathcal{S}_v \mid \textbf{import}\ M : \Delta = \mathcal{M}_v; \mathcal{S}_v \mid \textbf{import}^{\downarrow}\ M : \Delta = \mathcal{M}; \mathcal{S}_v
\end{array}
$$

$$\boxed{\sigma_1 \mid \Omega \mid \mathcal{M}_1 \longrightarrow \mathcal{M}_2 \mid \sigma_2} \qquad \textit{(Evaluating module)}$$

$$\textsc{ev-m-struct}\ \frac{\sigma_1 \mid \Omega \mid \mathcal{S} \longrightarrow \mathcal{S}' \mid \sigma_2}{\sigma_1 \mid \Omega \mid \textbf{struct}\ \mathcal{S}\ \textbf{end} \longrightarrow \textbf{struct}\ \mathcal{S}'\ \textbf{end} \mid \sigma_2} \qquad \textsc{ev-m-mvar}\ \frac{M = \mathcal{M} \in \Omega}{\sigma \mid \Omega \mid M \longrightarrow \mathcal{M} \mid \sigma} \qquad \textsc{ev-m-pmvar}\ \frac{p.M = \mathcal{M} \in \Omega}{\sigma \mid \Omega \mid p.M \longrightarrow \lceil \mathcal{M} \rceil^{p} \mid \sigma}$$

$$\boxed{\sigma_1 \mid \Omega \mid \mathcal{S}_1 \longrightarrow \mathcal{S}_2 \mid \sigma_2} \qquad \textit{(Evaluating structure item (excerpt))}$$

$$\textsc{ev-st-def1}\ \frac{\sigma_1 \mid \Omega \mid e \xrightarrow{0} e' \mid \sigma_2}{\sigma_1 \mid \Omega \mid \textbf{def}\ k = e; \mathcal{S} \longrightarrow \textbf{def}\ k = e'; \mathcal{S} \mid \sigma_2} \qquad \textsc{ev-st-def2}\ \frac{\sigma_1 \mid \Omega; k = v \mid \mathcal{S} \longrightarrow \mathcal{S}' \mid \sigma_2}{\sigma_1 \mid \Omega \mid \textbf{def}\ k = v; \mathcal{S} \longrightarrow \textbf{def}\ k = v; \mathcal{S}' \mid \sigma_2}$$

$$\textsc{ev-st-type}\ \frac{\sigma_1 \mid \Omega \mid \mathcal{S} \longrightarrow \mathcal{S}' \mid \sigma_2}{\sigma_1 \mid \Omega \mid \textbf{type}\ t = \tau; \mathcal{S} \longrightarrow \textbf{type}\ t = \tau; \mathcal{S}' \mid \sigma_2} \qquad \textsc{ev-st-macro}\ \frac{\sigma_1 \mid \Omega \mid \mathcal{S} \longrightarrow \mathcal{S}' \mid \sigma_2}{\sigma_1 \mid \Omega \mid \textbf{def}^{\downarrow}\ m = v; \mathcal{S} \longrightarrow \textbf{def}^{\downarrow}\ m = v; \mathcal{S}' \mid \sigma_2}$$

$$\boxed{\sigma_1 \mid \Omega \mid e_1 \xrightarrow{n} e_2 \mid \sigma_2\ (n {\geq} 0)} \qquad \textit{(Evaluating expression)}$$

$$\textsc{ev-app1}\ \frac{\sigma_1 \mid \Omega \mid e_1 \xrightarrow{n} e_1' \mid \sigma_2}{\sigma_1 \mid \Omega \mid e_1\ e_2 \xrightarrow{n} e_1'\ e_2 \mid \sigma_2} \qquad \textsc{ev-app2}\ \frac{\sigma_1 \mid \Omega \mid e \xrightarrow{n} e' \mid \sigma_2}{\sigma_1 \mid \Omega \mid v^n\ e \xrightarrow{n} v^n\ e' \mid \sigma_2} \qquad \textsc{ev-abs}\ \frac{\sigma \mid \Omega \mid e \xrightarrow{n+1} e' \mid \sigma}{\sigma \mid \Omega \mid \lambda x : \tau.\ e \xrightarrow{n+1} \lambda x : \tau.\ e' \mid \sigma}$$

$$\textsc{ev-beta}\ \frac{}{\sigma \mid \Omega \mid (\lambda x : \tau.\ e)\ v \xrightarrow{0} e[x \mapsto v] \mid \sigma} \qquad \textsc{ev-deref1}\ \frac{\sigma_1 \mid \Omega \mid e \xrightarrow{n} e' \mid \sigma_2}{\sigma_1 \mid \Omega \mid\ !e \xrightarrow{n}\ !e' \mid \sigma_2} \qquad \textsc{ev-deref}\ \frac{l \mapsto v \in \sigma}{\sigma \mid \Omega \mid\ !l \xrightarrow{0} v \mid \sigma}$$

$$\textsc{ev-ref1}\ \frac{\sigma_1 \mid \Omega \mid e \xrightarrow{n} e' \mid \sigma_2}{\sigma_1 \mid \Omega \mid \textbf{ref}\ e \xrightarrow{n} \textbf{ref}\ e' \mid \sigma_2} \qquad \textsc{ev-ref}\ \frac{l \notin \sigma}{\sigma \mid \Omega \mid \textbf{ref}\ v \xrightarrow{0} l \mid \sigma, l \mapsto v} \qquad \textsc{ev-assign1}\ \frac{\sigma_1 \mid \Omega \mid e_1 \xrightarrow{n} e_1' \mid \sigma_2}{\sigma_1 \mid \Omega \mid e_1 := e_2 \xrightarrow{n} e_1' := e_2 \mid \sigma_2}$$

$$\textsc{ev-assign2}\ \frac{\sigma_1 \mid \Omega \mid e \xrightarrow{n} e' \mid \sigma_2}{\sigma_1 \mid \Omega \mid v^n := e \xrightarrow{n} v^n := e' \mid \sigma_2} \qquad \textsc{ev-assign}\ \frac{l \in \sigma}{\sigma \mid \Omega \mid l := v \xrightarrow{0} \textbf{unit} \mid \sigma[l \mapsto v]}$$

$$\textsc{ev-quote}\ \frac{\sigma_1 \mid \Omega \mid e_1 \xrightarrow{n+1} e_2 \mid \sigma_2}{\sigma_1 \mid \Omega \mid \langle e_1 \rangle \xrightarrow{n} \langle e_2 \rangle \mid \sigma_2} \qquad \textsc{ev-splice}\ \frac{\sigma_1 \mid \Omega \mid e_1 \xrightarrow{n} e_2 \mid \sigma_2}{\sigma_1 \mid \Omega \mid \$e_1 \xrightarrow{n+1} \$e_2 \mid \sigma_2} \qquad \textsc{ev-spliceCode}\ \frac{}{\sigma \mid \Omega \mid \$\langle v^1 \rangle \xrightarrow{1} v^1 \mid \sigma}$$

$$\textsc{ev-kvar}\ \frac{k = v\ \in\ \Omega}{\sigma \mid \Omega \mid k \xrightarrow{0} v \mid \sigma} \qquad \textsc{ev-pkvar}\ \frac{p.k = v\ \in\ \Omega}{\sigma \mid \Omega \mid p.k \xrightarrow{0} \lceil v \rceil^{p} \mid \sigma} \qquad \textsc{ev-macro}\ \frac{m = v\ \in\ \Omega}{\sigma \mid \Omega \mid m \xrightarrow{0} v \mid \sigma} \qquad \textsc{ev-pmacro}\ \frac{p.m = v\ \in\ \Omega}{\sigma \mid \Omega \mid p.m \xrightarrow{0} \lceil v \rceil^{p} \mid \sigma}$$

Fig. 12. Dynamic semantics in the core calculus $maco_{core}$

At a high-level, just like **def**, we evaluate **module** and **import**ed modules to values and add them to the evaluation context; and just like macros, we ignore **import**$^{\downarrow}$ed modules.

The judgment $\sigma_1 \mid \Omega \mid e_1 \xrightarrow{n} e_2 \mid \sigma_2$ reads: under heap σ_1 and evaluation context Ω, evaluating e_1 at level n results to e_2 and updates the heap to σ_2. The rules are used for both compile-time evaluation (as in rule codeGen) and runtime. The judgment is level-indexed: intuitively, the rule searches for expressions at level 0 to evaluate, adjusting the level when evaluating inside quotations and splices. For an application $e_1\ e_2$ at level n, we first evaluate e_1 (rule ev-app1) until it becomes a value v^n, and we evaluate e_2 (rule ev-app2). We evaluate a lambda $\lambda x : \tau.\ e$ by evaluating its body, so splices inside the body can get evaluated (rule ev-abs); the level is positive as there are no splices at negative levels. Beta-reduction only happens at level 0 (rule ev-beta). Evaluation for references

$$\begin{array}{l} \Omega \quad = \quad m = \lambda x : \text{Ref Int. if } (!x) = 0 \text{ then } \langle -1 \rangle \text{ else } \langle 1 \rangle \\ \bullet \mid \Omega \mid x \leftarrow \text{ref } 0; \langle \$(x := 1; m\,x) \rangle \\ \xrightarrow{0} x \leftarrow l; \langle \$(x := 1; m\,x) \rangle \mid l \mapsto 0 \\ \xrightarrow{0} \langle \$(l := 1; m\,l) \rangle \mid l \mapsto 0 \;\; \textcircled{1} \\ \xrightarrow{0} \langle \$(\text{unit}; m\,l) \rangle \mid l \mapsto 1 \;\; \textcircled{2} \\ \xrightarrow{0} \langle \$(m\,l) \rangle \mid l \mapsto 1 \\ \xrightarrow{0} \langle \$\langle 1 \rangle \rangle \mid l \mapsto 1 \;\; \textcircled{3} \\ \xrightarrow{0} \langle 1 \rangle \mid l \mapsto 1 \;\; \textcircled{4} \;\; (\textsc{ev-spliceCode}) \end{array}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{l \mapsto 0 \mid \Omega \mid l := 1 \xrightarrow{0} \text{unit} \mid l \mapsto 1}\;\textsc{ev-assign}}{l \mapsto 0 \mid \Omega \mid l := 1; m\,l \xrightarrow{0} \text{unit}; m\,l \mid l \mapsto 1}\;\textsc{ev-app2}}{l \mapsto 0 \mid \Omega \mid \$(l := 1; m\,l) \xrightarrow{1} \$(\text{unit}; m\,l) \mid l \mapsto 1}\;\textsc{ev-splice}}{\textcircled{1}\; l \mapsto 0 \mid \Omega \mid \langle \$(l := 1; m\,l) \rangle \xrightarrow{0} \langle \$(\text{unit}; m\,l) \rangle \mid l \mapsto 1 \;\textcircled{2}}\;\textsc{ev-quote}$$

Fig. 13. Example: evaluation steps for $\bullet \mid \Omega \mid x \leftarrow \text{ref } 0; \langle \$(x := 1; m\,x) \rangle \xrightarrow{0}{}^{*} \langle 1 \rangle \mid l \mapsto 1$ in Fig. 10 (left). For concision, we write $\sigma_1 \mid \Omega \mid e \xrightarrow{n} \cdots \xrightarrow{n} e \mid \sigma$ for a series of evaluation steps. Derivation from ① to ② (right).

$$\boxed{\sigma \mid \Gamma \vDash^{\mathsf{c}} \Omega} \qquad \textit{(compile-time)} \qquad\qquad \boxed{\sigma \mid \Gamma \vDash^{\mathsf{r}} \Omega} \qquad \textit{(runtime)}$$

$$\dfrac{\sigma \mid \Gamma \vDash^{\mathsf{c}} \Omega}{\sigma \mid \Gamma, k : \tau \vDash^{\mathsf{c}} \Omega} \qquad \dfrac{\sigma \mid \Gamma \vDash^{\mathsf{c}} \Omega \qquad \sigma \mid \Gamma \vdash^{-1} v : \tau}{\sigma \mid \Gamma, m : \tau \vDash^{\mathsf{c}} \Omega; m = v} \qquad \dfrac{\sigma \mid \Gamma \vDash^{\mathsf{r}} \Omega \qquad \sigma \mid \Gamma \vdash^{0} v : \tau}{\sigma \mid \Gamma, k : \tau \vDash^{\mathsf{r}} \Omega; k = v} \qquad \dfrac{\sigma \mid \Gamma \vDash^{\mathsf{r}} \Omega}{\sigma \mid \Gamma, m : \tau \vDash^{\mathsf{r}} \Omega}$$

Fig. 14. Well-formedness of evaluation contexts (excerpt). Modules in the context are checked similarly.

is similar to applications, and we have rules for reference creation (rules EV-REF1 and EV-REF), gets (rules EV-DEREF1 and EV-DEREF), and sets (rules EV-ASSIGN1, EV-ASSIGN2, and EV-ASSIGN).

Quotations (rule EV-QUOTE) and splices (rule EV-SPLICE) increment and decrement the evaluation level respectively. Rule EV-SPLICECODE is where we cancel out a pair of quotation and splice: when v^1 is a value at level 1, splicing the quotation $\langle v^1 \rangle$ at level 1 removes the quotation and steps to v^1.

The last four rules evaluate definitions at level 0 to their values in the evaluation context. Rules EV-MACRO and EV-PMACRO for macros should be used only for compile-time evaluation.

4.3 Example

Fig. 13 presents the evaluation steps for the compile-time evaluation happened in Fig. 10. The derivation from ① to ② is given on the right of the figure, demonstrating how we search for expressions to evaluate inside quotations and splices and how heaps are updated. The last step from ③ to ④ cancels out a pair of quotation and splice with rule SPLICECODE.

4.4 Type Soundness

We discuss type soundness for the core calculus *maco*$_{core}$. First, we need notions of contexts being *well-formed.* Γ ok means that all types in the type context are well-formed with all type variables bound. σ ok checks all values in the heap are integers. An evaluation context Ω is well-typed, with respect to a context Γ and a heap σ, if all definitions in it has the type specified by Γ. Interestingly, as the reader may have noticed, our calculi distinguish between *compile-time* and *runtime evaluation contexts*. Specifically, when typing the source calculus, we add macros (rule ST-MACRO) but not definitions (rule ST-DEF) to the evaluation context, while we do the opposite when evaluating modules: we add definitions (rule EV-ST-DEF2) but not macros (rule EV-ST-MACRO). Therefore, evaluation contexts has two well-formedness judgments $\sigma \mid \Gamma \vDash^{\mathsf{c}} \Omega$ and $\sigma \mid \Gamma \vDash^{\mathsf{r}} \Omega$, as given in Fig. 14, used for compile-time and runtime computations, respectively.

Preservation holds for both compile-time and runtime evaluation. For space reasons, we show theorems only for expressions, but theorems 4.1, 4.2, and 4.4 also hold for modules and structures.

Theorem 4.1 (Preservation). *Given* Γ ok, *and* σ ok, *and* $\sigma \mid \Gamma \vDash^{\mathsf{r}} \Omega$ *or* $\sigma \mid \Gamma \vDash^{\mathsf{c}} \Omega$, *if* $\sigma \mid \Gamma \vDash^{n'} e : \tau$ *and* $\sigma \mid \Omega \mid e \xrightarrow{n} e' \mid \sigma'$, *then* $\sigma' \mid \Gamma \vDash^{n'} e' : \tau$.

The theorem does not relate the typing level n' to the evaluation level n, as it takes typing and the evaluation step as given, and proves that every possible step of evaluation preserves typing.

The progress theorem is subtler. For example, evaluating modules (rule ev-st-macro) does not add macros to the evaluation context, so we must ensure runtime evaluation never encounters macros (rule ev-macro) to avoid evaluation getting stuck. Similarly, compile-time evaluation should never encounter rule ev-kvar. But given an expression, how do we know whether we are evaluating at compile-time or runtime? Rule codeGen gives us a hint: the elaborated expression e is typed at level $n-1$, which is either -1 or -2 for top-level splices in let definitions and macros, but evaluated at level 0 — thus, we can determine the evaluation phase by comparing the typing level n' with the evaluation level n: a smaller typing level indicates compile-time evaluation; otherwise, the phase is runtime. The following theorem establishes progress, where the notation $\hat{\Gamma}$ checks that Γ does not contain local variables (i.e. x).

Theorem 4.2 (Progress). *Given* $\hat{\Gamma}$ ok*, and* σ ok*, and* $(n' < n \land \sigma \mid \Gamma \Vdash^{c} \Omega)$ *or* $(n' \geq n \land \sigma \mid \Gamma \Vdash^{r} \Omega)$*, 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* $\sigma \mid \Omega \mid e \xrightarrow{n} e' \mid \sigma'$.

Combining preservation and progress yields type soundness. Below, we show soundness for runtime where $n' = n = 0$, and for compile time where $n' = -1$ (or any negative level) and $n = 0$.

Theorem 4.3 (Soundness). *Given* $\hat{\Gamma}$ ok*, and* σ ok,

- *Runtime: if* $(\sigma \mid \Gamma \Vdash^{r} \Omega)$ *and* $\sigma \mid \Gamma \Vdash^{0} e : \tau$ *where* e^{0}*, then either* e *is a value* v^{0}*, or there exist* e' *and* σ' *such that* $\sigma \mid \Omega \mid e \xrightarrow{0} e' \mid \sigma'$ *and* $\sigma' \mid \Gamma \Vdash^{0} e' : \tau$*;*
- *Compile time: if* $(\sigma \mid \Gamma \Vdash^{c} \Omega)$ *and* $\sigma \mid \Gamma \vdash^{-1} e : \tau$ *where* e^{0}*, then either* e *is a value* v^{0}*, or there exist* e' *and* σ' *such that* $\sigma \mid \Omega \mid e \xrightarrow{0} e' \mid \sigma'$ *and* $\sigma' \mid \Gamma \vdash^{-1} e' : \tau$.

4.5 Elaboration Soundness and Phase Distinction

We discuss additional properties of our calculi.

Elaboration soundness and phase distinction of heaps. First, we want to show that elaboration preserves typing. A natural question then is: if $\sigma \mid \Omega \mid \Gamma \Vdash^{n}_{\star} e : \tau \rightsquigarrow e \mid \sigma'$, under which heap, σ or σ', should we type-check e? The answer is: *neither.* In practice, we may compile a program in one environment, and then run the compiled program in another environment, where compile-time information is no longer available. It is therefore important that e does not refer to values from the compile-time heap. Indeed, in the following theorem, we prove that e is well-typed under an *empty* heap in the core calculus. Namely, heaps have a clear compile-time and runtime phase distinction, suggesting that we can safely discard the compile-time heap after compilation.

Theorem 4.4 (Elaboration Soundness). *Given* Γ ok*,* σ ok*, and* $\sigma \mid \Gamma \Vdash^{c} \Omega$*, if* $\sigma \mid \Omega \mid \Gamma \Vdash^{n}_{\star} e : \tau \rightsquigarrow e \mid \sigma'$*, then* $\bullet \mid \Gamma \Vdash^{n} e : \tau$*. Moreover, if* $\star = q$*, then* e^{1}*, else* e^{0}*.*

The intuition behind the empty heap is as follows. In rule codeGen, suppose e is typed at level -1 and then evaluated at level 0. Any locations created at evaluation level 0 are at typing level -1. Because of preservation, the result $\langle v^{1} \rangle$ is also of typing level -1, and thus v^{1} has typing level 0, and so v^{1} will not be able to capture locations from typing level -1.

The theorem also says that e is an expression at level 1 under mode q and at level 0 otherwise. Thus in rule codeGen since e is typed under s, we have e^{0}. Therefore compile-time type soundness (Theorem 4.3) applies to the evaluation derivation in rule codeGen.

Phase distinction. The distinction between compile-time and runtime heaps have already made it evident that compile-time only computations are not needed for runtime evaluation. The following theorem makes the phase distinction further explicit: the notation $[\![\ ]\!]$ erases all macros (including those inside modules) and modules imported at compile-time (**import**$^{\downarrow}$). The theorem says that if a module $\mathcal{M}$ evaluates to $\mathcal{M}'$, then after erasure, $[\![\mathcal{M}]\!]$ evaluates to $[\![\mathcal{M}']\!]$. Consequently, when evaluating a module we can safely erase all compile-time only computations.

Theorem 4.5 (Phase Distinction)**.** *Given* Γ ok, σ ok, *and* $\sigma \mid \Gamma \Vdash^{r} \Omega$, *if* $\sigma \mid \Gamma \vdash \mathcal{M} : \Delta$, *and* $\sigma \mid \Omega \mid \mathcal{M} \longrightarrow \mathcal{M}' \mid \sigma'$, *then* $\sigma \mid [\![\Omega]\!] \mid [\![\mathcal{M}]\!] \longrightarrow [\![\mathcal{M}']\!] \mid \sigma'$.

5 MACOCAML: IMPLEMENTATION

So far we have focused on the calculi that capture the essence of our design. We have incorporated our design into OCaml, and include the modified OCaml compiler in the supplementary material.

5.1 Compiler

The implementation is a substantial change that touches many parts of the compiler (top level, type checker, runtime, dynamic loading, parser, code generator, primitive types, standard library, etc.).

Syntax. For harmony with other OCaml features our syntax differs from the formalism in several ways. Rather than **def** $k = e$ and **def**$^{\downarrow}$ $m = e$ we write **let** `k = e` and **macro** `m = e`. In place of **import** M or **import**$^{\downarrow}$ M we project from modules using `M.k` at runtime or `~M.k` at compile time and make names available without qualification using **open** `M` and **open** `~M`. For the code type we write postfix `expr` rather than prefix Code.

Quotation. Following other multi-stage language implementations such as BER MetaOCaml [Kiselyov 2014], compilation translates typed quoted expressions into combinators that construct terms. Since type checking guarantees that generated code is well-typed, the combinators do not need to carry type information; they construct and compose untyped representations.

Splicing. The execution of the compile-time code that is inserted into top-level splices builds an array of intermediate code values (type `lambda array`). Splicing these values constructs a single large representation for each module that is compiled with OCaml's standard backend.

Compile-time evaluation. As in the formalism, compilation involves evaluating top-level splices. These are first translated to bytecode, then executed using the `Meta.reify_bytecode` function that OCaml's interactive top-level uses to execute phrases. Although the OCaml compiler supports compilation to both bytecode and native code, for simplicity and portability in our implementation compile-time code is currently always compiled and executed as bytecode. In the future we plan to support native code execution in compile-time by integrating a method to generate and execute native code dynamically as proposed by Fischbach and Meurer [2011].

Compilation. Since modules can only import compiled modules which type-check under an empty heap, it is obvious that compilation of a module does not depend on side effects that occurred during the compilation of imported modules (§3.1). When compiling a module, each imported module is visited (similarly, invoked) at most once at each level. For example, **module** `~ST1 = Term` followed by **module** `~ST2 = Term` will only visit `Term` once[8]. As another example, if both modules A and B import `Term` at compile-time, and another module C imports A at runtime and B at compile-time, then `Term` will be visited twice, once at level -1 (for A) and once at level -2 (for B). If module C imports both A and B at runtime, then `Term` is visited only once at level -1, and A and B share its heap state at level -1.

5.2 Language Extensions

The implementation of MacoCaml follows the key ideas of our design. Full integration in OCaml requires additional steps, which we touch on briefly here.

[8] This behavior is not modeled in the formalism, but it is easy to extend the formalism with an additional context that records identifiers of visited modules, so that visiting does not revisit a visited module, but directly sets, e.g., ~ST2 = ~ST1.

References of code. Our calculus *maco* restricts to integer references to study compile-time heaps. If references can store code, then a variable can escape its scope if code quoting it can be stored in references and spliced outside the corresponding binder; this issue is known as *scope extrusion*. MacoCaml follows other systems (e.g. Kiselyov [2014]) in detecting scope extrusion at splice time.

Module subtyping. In the formalism of *maco*, every module exports all the names that are defined in its body. In contrast, the richer module systems of full-scale ML family languages such as OCaml offer *module subtyping* [Mitchell and Harper 1988] that supports exporting only a subset of names. There is an interesting interaction between subtyping and quotation. In the example below, the signature of `M` exports a macro `public` that quotes an unexported function `secret`. When the result of `M.public` is spliced outside the module, its expansion includes a name that is not in scope:

```
module M : sig macro public : int expr → int expr end = struct
  let secret x = x * x
  macro public x = <<secret ($x + 1) >>
end
$(M.public << 3 >>) (* M.secret (3 + 1); but secret is not exported by M! *)
```

To deal with this, the compiler incorporates *path closures*, which systematically transform macros and signatures to ensure that names like `secret` that are hidden from user code by module boundaries are nonetheless accessible in elaborated programs. Essentially, the compiler closure-converts each macro that uses module-local definitions; the produced closure is a quoted module containing those definitions, whose path is then injected into the macro definition. As a result, the above program will generate `M.Closure1.secret (3+1)`, instead of `M.secret (3+1)`. As users cannot access closures, data abstraction is preserved for user programs. As macros are always function values, the insertion of an additional parameter to inject the path closure does not change their evaluation behaviour. We leave a formal treatment of path closures to future work.

Cross stage persistence. Cross-stage persistence (CSP) in multi-stage languages comes in three flavours: heap-based CSP, value-based CSP, and path-based CSP. In heap-based CSP, heap-allocated values are stored in code as reference to the heap. In compile-time staging systems such as MacoCaml and Template Haskell [Sheard and Jones 2002] this form of CSP is not permitted, since the compile-time heap where code is constructed is discarded after compilation. For path-based CSP, top-level identifiers can appear in code quotations and are stored as names. Our system naturally allows something similar: a macro can quote a top-level let identifier and, more generally, quotations at level n can quote identifiers at level $n + 1$. Path closures, discussed above, manage the tricky interactions with the hiding of top-level names.

Finally, in value-based CSP, simple immutable values are automatically converted to their code representations. For example, in MetaOCaml one might write `let x = "foo" in .< x >.` to produce a code value equivalent to `.<"foo">.`. In our system this example is rejected as ill-typed; the user must apply a type-specific function `lift_string : string → string expr` instead. In some systems, such as Template Haskell, these lift functions are inserted automatically using type classes, and we hope that the eventual integration of *modular implicits* [White et al. 2014] will allow a similar approach in MacoCaml.

5.3 Libraries

While we leave a systematic evaluation of MacoCaml to future work, we have ported two substantial existing libraries to MacoCaml to validate our implementation.

Stream fusion. Kiselyov et al. [2017] present a MetaOCaml library, *Strymonas*, for stream fusion with appealing performance improvements on OCaml microbenchmarks. The interface offers high-level combinators such as `map` and `fold`, and generates efficient loop-based code.

We have ported the 904 lines of code of *Strymonas* to MacoCaml, which generates *identical code* to the original library, thus inheriting the same performance improvements. Although *Strymonas* was designed for run-time staging, the port was straightforward, since MacoCaml's quotes and splices are similar to MetaOCaml's. The compilation time is also similar, at around 600ms for both the MetaOCaml implementation and the MacoCaml port. Besides syntactic updates, porting involved two changes. First, the original implementation has several uses of cross-stage persistence, which MacoCaml does not support; we changed these to explicit applications of MacoCaml's *lifting* functions (e.g. `Expr.of_int : int → int expr`). Second, where the original implementation executes quoted code either using MetaOCaml's primitive `run` function or by printing to a file followed by compilation, the MacoCaml port uses top-level splices. We expect porting other existing MetaOCaml libraries to be similarly straightforward.

Format. OCaml has a sophisticated formatting library that represents typed format trees using GADTs [Vaugon 2013] similarly to the example in §2.5. The library is distributed with OCaml, and the compiler includes a translation from format strings such as `"(%d, %d)"` to typed format trees, which are then interpreted at run-time by functions such as `printf` and `scanf`.

We have staged the core formatting library (2476 lines) using MacoCaml to interpret the format trees during compilation, eliminating run-time overhead. The port to MacoCaml was largely a matter of performing a manual binding-time analysis to ascertain which subexpressions in the program are statically known, then adding the `expr` type, quotations, splices and import annotations accordingly. The implementation uses most of the features described in the paper (147 quotes, 85 splices, 33 module annotations), but does not use compile-time effects, since the generation of specialized code from format strings in the original library is purely functional.

We have measured the compilation overhead of using MacoCaml's features, and found it to be modest: the ported format library compiles in 543ms, compared to 477ms for the original library.

Other libraries. We expect that other OCaml libraries will present similar opportunities for optimization using MacoCaml's staging constructs . Regular expression libraries (e.g. `re` [2023]) will benefit from compiling regexes during compilation, and from generating OCaml code without the overhead of table-based automata. Numerical libraries (e.g. `Owl` [Wang and Zhao 2022]) will be able to use macros to generate specialized code (e.g. applying the generative techniques in Carette and Kiselyov [2005]). Libraries that generate code in an untyped way (e.g. the `ctypes` foreign function library [Yallop et al. 2018]) will enjoy improved implementations using our typed code generation facilities. Generic programming libraries based on type representations (e.g. `lrt` [2023]) will be able to statically generate type-specialized functions using the techniques in Yallop [2017]. More generally, a library will benefit from MacoCaml's staging constructs whenever some aspect of the library can be specialized using information (e.g. a regex, a parameter, or a type) that is unavailable when the library is written, but available at the point where code using the library is compiled.

6 RELATED WORK

Staging and macros have a rich literature. Here we discuss works that are most relevant.

Staged programming. Staging has been used for both compile-time [Sheard and Jones 2002; Xie et al. 2022] and runtime code generation [Calcagno et al. 2003b; Kiselyov 2014; Taha et al. 1998], each with its own merits. Compile-time code generation, as in our system, has a clear and proven phase distinction between compile-time and runtime heaps, while dynamic code generation allows code to be specialized with respect to information that only becomes available at runtime.

Template Haskell (TH) [Sheard and Jones 2002] supports compile time code generation and our notion of compiler modes is inspired by TH. However, TH does not present formal operational semantics. Xie et al. [2022] formalized Typed Template Haskell (TTH) with operational semantics, but TTH is not yet implemented. Neither work supports side effects, compile-time heaps, or modules. TTH also does not explicitly formalize compile-time evaluation; instead, negative-leveled splices are lifted to top-level so that they are evaluated before the rest of the program, leaving evaluation phases unclear. Kovács [2022] takes a radically different approach, employing *two-level type theory* (2LTT) to model staged compilation in a dependently typed setting.

Staging for runtime code generation comes with a rather different methodology. Languages such as MetaML [Taha et al. 1998], MetaOCaml [Calcagno et al. 2003b], and BER MetaOCaml [Kiselyov 2014] support an additional run construct that evaluates code at runtime. There are no compile-time bindings, top-level splices, or compile-time evaluation. As code is generated at runtime, there is no clear phase distinction. Like the present paper, the MetaML and MetaOCaml work also defines calculi and establishes theorems, but their formalisms do not support features like side-effects, heaps, or modules. Our future plans include extending MacoCaml with runtime code generation.

Another line of work uses *modalities* to model staged computation. Davies and Pfenning [1996] introduce a language Mini-ML$^{\Box}$ based on the modal logic S4, that supports manipulation of closed terms. Davies [1996] shows that the application of the Curry-Howard correspondence to linear-time temporal logic produces a language $\lambda^{\bigcirc}$ supporting manipulation of open terms (similarly to MetaML, which was introduced shortly afterwards). Nanevski et al. [2008] generalize the work of Davies and Pfenning in a different direction, augmenting the $\Box$ type modality with the context of free variables that may appear in a term of that type. This *contextual modal type theory* provides an appealing basis for metaprogramming, but integrating it into a language like OCaml would be a significantly more disruptive change than adding a simple MetaML-style type constructor for code.

Macros and modules. The design of the module system in MacoCaml is directly built on top of Racket [Flatt 2002] (also see its extensions [Culpepper et al. 2007; Flatt et al. 2012]), which fits extremely well with our notion of macros and staging (§2.3). Flatt [2013] further allows shifting a module at a positive level. From the typing perspective, shifting a module at a positive level would work in our system, though it remains to see what implications that would have to code generation and runtime; for example, shifting a module at level +1 means macros from that module are now at level 0 and can be used at runtime.

However, there are important differences between the notion of macros in MacoCaml and those in Racket. First, Racket and MacoCaml manage macros differently. In particular, a macro in Racket can be viewed as a binding whose body is *defined* at level -1 but which itself is *bound* at level 0[9]. This approach allows a macro body to use definitions imported at compile-time, while the macro itself can be called and expanded in a normal program. The discrepancy between levels can cause surprising results. For example, consider the following example adapted from the Racket document[10]:

```
(module a racket                (module b racket
  (define button 0)               (require (for-syntax 'a)) (* compile-time import *)
  (define see-button #'button)    (define-syntax (m stx) see-button)
  (provide button see-button))    (m))        (* error: button: unbound identifier *)
```

Instead of returning 0, evaluating module b will raise an error about button being unbound. Let us understand what happened. Module a defines button and see-button as normal bindings at level 0, where see-button's value is the *syntax object* (#') for button.[11] The provide form exports the definitions. Module b imports module a at compile-time, so both button and see-button are at level -1. The macro m, as its body is defined at level -1, can see see-button at level -1 and will

[9] https://docs.racket-lang.org/reference/syntax-model.html#%28part._transformer-model%29

[10] https://docs.racket-lang.org/guide/phases.html#%28part._Phases_and_.Modules%29

[11] (#') captures a binding at all levels. Such flexibility is not needed for this example.

return the `#'button` syntax object, which refers to `button` at level -1. The use of `m` is at level 0, as the macro binding itself is at level 0. So both the macro definition and its use are well-leveled — but the program raises an error after macro expansion! The reason is that since macro `m` is used at level 0, it expands to `#'button` at level 0, but there is no `button` at level 0![12]

MacoCaml differs from Racket, giving a more consistent and refined view of compile-time computations: macros are defined and bound both at level -1, and by leveraging staging annotations we can distinguish between a macro such as `mpower` (§2.3), which is an expression at level -1, and splices of macros such as `$(mpower 5)`, which is a top-level splice at level 0 that triggers compile-time evaluation. Moreover, MacoCaml is fully typed, built on top of *maco* with proven soundness results, so a well-typed program never generates an ill-typed program.

On the other hand, macros in Flatt's system are more expressive as they may scrutinise abstract syntax. In a typed setting, such *analytic* macros require a sophisticated type system (such as *contextual types* used in Squid [Parreaux et al. 2018] or Moebius [Jang et al. 2022]) that exposes contexts in types rather than a simple MetaML-style system that is sufficient for our purely *generative* macros.

Macros and staging in MacroML. MacroML [Ganz et al. 2001] views macros as multi-stage programs, by translating macros to multi-stage programs in MetaML. The key idea is to model macro expansion as one step of elaboration under level 1 ($\vdash^{-1} e \rightsquigarrow e'$), which translates ordinary arguments into code and macro applications into splices, followed by one step of evaluating the quotation of the resulted program ($\langle e' \rangle \xrightarrow{0}{}^{*} e'_1$). Regular execution then calls the run construct (run $e'_1 \xrightarrow{0}{}^{*} e'_2$) to get the final result. As an example, the power macro from §2.1 can be expressed in MacroML as (using MacoCaml notations for quotations and splices throughout this section):

```
let mac mpower $n x =                                        (* e *)
  $(if n = 0 then <<1>> else <<x * (mpower (n-1) x)>>)
in mpower 5 2
```

where **mac** specifies a macro definition, `$n` means an early parameter, and `x` is an ordinary parameter. The program translates to:

```
$(letrec mpower n x =                                        (* e' *)
   (if n = 0 then <<1>> else <<$x * $(mpower (n-1) x)>>)
    in mpower 5 <<2>>)
```

The macro expansion step evaluates the quotation of the above program, generating:

```
<<2 * (2 * (2 * 2 * (2 * 1)))>>                              (* e1' *)
```

Executing the program then calls `run` on the above program, resulting in:

```
32                                                           (* e2' *)
```

MacoCaml differs from MacroML in several aspects. At a high level, rather than considering macros as staged programs, MacoCaml combines macros and staging in the same language, where macros are compile-time bindings and staging and macros can interact. Moreover, MacoCaml models compile-time evaluation using top-level splices and runtime evaluation via direct operational semantics. In contrast, evaluation in MacroML is, in a sense, simulating the behavior of top-level splices: as macro applications elaborate to splices, the elaboration result e' could contain top-level splices, and thus macro expansion evaluates $\langle e' \rangle$ rather than e'. This in turn forces regular execution to call `run` on the evaluation result (i.e. run e'_1) as the result of macro expansion e'_1 is unnecessarily wrapped inside a quotation. Lastly, extending MacroML with references would require proving

[12] See the document for possible fixes. However, our point here is that expanding a well-leveled macro can raise an error.

phase distinction by showing that the runtime heap from $\langle e' \rangle \xrightarrow{0}^* e'_1$ is not needed for regular execution run $e'_1 \xrightarrow{0}^* e'_2$.

Combining macros and staging in Scala. Stucki et al. [2018] described a design in Scala to combine macros and multi-stage programming. As they did not present formal semantics, we compare with their system based on the design description and example programs from their paper.

In their setting, macros are inline functions defined using top-level splices, which are only evaluated when the code is inlined. For example, the power macro from §2.1 is defined as follows:

```
def mpower'(n: Int, x:Expr[Int]) = if n = 0 then <<1>> else <<$(x) * $(mpower' (n-1) x)>>)
inline def mpower (inline n: Int, x:Int) = $(mpower' n <<x>> )        (* no evaluation *)
mpower (5, 2)  (* inlined to $(mpower' 5 <<2>>) *)
               (* then expanding to 2 * (2 * (2 * 2 * (2 * 1))) *)
```

where mpower' is a normal definition, and the macro mpower is defined as an inline function with a top-level splice. There are a few notable things. First, mpower' is defined at level 0, but used at level -1 in the definition of mpower. To make the call to mpower' well-staged, macros such as mpower are type-checked *"as if they were in a quoted context"* [Stucki et al. 2018, §3.3]. This can be implemented by type-checking mpower under level 1, similar to MacroML. Moreover, this means that programs using macros such as mpower (5, 2) also need to be *"thought of as a quoted program"* to make sure that after mpower expands, the call to mpower' is still well-staged. This is conceptually similar to MacroML where the elaboration result is evaluated inside a quotation. Furthermore, because mpower has a top-level splice, the parameter n seems ill-staged, as it is introduced at the definition level of mpower, but used in a lower level inside the splice. To make it work, the program marks n as inline, reflecting the fact that its value will be known during macro expansion. Furthermore, the top-level splice in the definition of mpower does not trigger evaluation until it is inlined when mpower is called.

MacoCaml differs from Scala in several aspects. First, by modeling macros as compile-time bindings, MacoCaml maintains well-stagedness without needing the inlining mechanism or considering programs in a quoted context. Moreover, while both MacoCaml and Scala consider top-level splices as compile-time evaluation, MacoCaml uses top-level splice at the macro expansion site instead of the macro definition site, thus treating top-level splices more consistently. However, we note that some of those differences arise from Scala's aim of supporting both compile-time and runtime code generation. For runtime code generation, it is important that mpower' is not a macro:

```
val power5 = << (x:Int) ⇒ $(mpower'(5, <<x>>)) >>.run
power5(2)  (* generating at runtime: 2 * (2 * (2 * 2 * (2 * 1))) *)
```

It is thus interesting as future work to extend MacoCaml with runtime code generation and compare the resulting system again with Scala.

Combining staging and references. Calcagno et al. [2003a] studied references in MetaML and handled *scope extrusion*, where a variable can escape its scope if code quoting it can be stored in a reference then spliced outside its binder. More recently, Kiselyov et al. [2016] introduced a type system that models context nesting using subtyping, allowing references to store open code values while preventing scope extrusion. *maco* restricts references to integers as they serve a different purpose: they make compile-time heaps explicit, thus making the language interesting for studying phase distinction. We remark that scope extrusion is orthogonal to our phase distinction result. That is, even in a system with scope extrusion, one may still prove phase distinction (Theorem 4.5), as the extruded part is still well-leveled (though not well-scoped) and any compile-time only (extruded or not) computations are not needed for runtime evaluation.

We believe that the type systems of Calcagno et al. and Kiselyov et al. can be integrated into our system to support records of code. Currently, our implementation (§5.2) follows MetaOCaml [Kiselyov 2014] in detecting scope extrusion at splice time.

Staging modules. While MacoCaml focuses on term-level code generation, there is a line of work [Inoue et al. 2016; Sato and Kameyama 2021; Sato et al. 2020] on generating modules for eliminating performance penalty from functor applications. In the future we would be interested to see how module generation could be incorporated in our system. In particular, we would like to study how the approach in [Sato and Kameyama 2021; Sato et al. 2020], which converts code of a module into a module of code, relates to path closures (§5.2) used in our system, which essentially require quoting and splicing of module paths.

Existing approaches to compile-time computation in OCaml. OCaml currently offers two approaches to compile-time computation. Historically, Camlp5 [de Rauglaudre 2007] (and the similar tool, Camlp4), have supported pre-processing of OCaml programs by transformation of concrete syntax trees. More recently, `ppx` [White 2013] supports transformation of OCaml programs by abstract syntax tree transformation. Although these tools have notable drawbacks — e.g. they do not guarantee hygiene or type-preservation — they have found many uses in practice: for example, `ppx_monad` provides Haskell-style "do" notation, `ppx_expect` provides facilities for embedding expect tests in programs, and `ppx_bitstring` provides bitstring pattern matching. We expect that these and other such libraries can be restructured to keep their current `ppx`-based interfaces and use MacoCaml's typed code representations internally, improving confidence in their correctness.

7 CONCLUSION

We have presented the design and implementation of MacoCaml. The MacoCaml system supports compile-time code optimization of a variety of OCaml programs, and its formalization *maco* can be used to support a novel combination of staging and macros for other languages.

In the future, we plan to extend MacoCaml with a number of features. First, we would like to allow nested quotations and splices. Nested splices will raise an interesting question: how to ensure that a nested splice is evaluated before a single splice, while evaluation is interleaved with typing? Second, we plan to enrich our module language with module subtyping as described in §5.2, as well as *functors*, where macros, like types, will be considered as compile-time components of modules, by following and extending the phase splitting transformation by Harper et al. [1989].

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments, and members of IFIP WG 2.11 for feedback on a presentation of this work.

This work is funded by the Isaac Newton Trust (grant no. G101121), by EPSRC (EP/K031724/2), by the Natural Sciences and Engineering Research Council of Canada, by Jane Street Capital, and by Ahrefs.

REFERENCES

David Abrahams and Aleksey Gurtovoy. 2004. *C++ template metaprogramming: concepts, tools, and techniques from Boost and beyond.* Pearson Education.

Eugene Burmako. 2013. Scala macros: Let our powers combine! On how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala.* 1–10.

Cristiano Calcagno, Eugenio Moggi, and Tim Sheard. 2003a. Closed types for a safe imperative MetaML. *Journal of functional programming* 13, 3 (2003), 545–571.

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

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

William Clinger and Jonathan Rees. 1991. Macros that work. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. 155–162.

Ryan Culpepper, Sam Tobin-Hochstadt, and Matthew Flatt. 2007. Advanced Macrology and the Implementation of Typed Scheme. In *Proceedings of the Eighth Workshop on Scheme and Functional Programming*.

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.

Marcell Fischbach and Benedikt Meurer. 2011. Towards a native toplevel for the OCaml language. *CoRR* abs/1110.1029 (2011). arXiv:1110.1029 http://arxiv.org/abs/1110.1029

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

Matthew Flatt. 2013. Submodules in Racket: You Want It When, Again?. In *Proceedings of the 12th International Conference on Generative Programming: Concepts and Experiences* (Indianapolis, Indiana, USA) *(GPCE '13)*. Association for Computing Machinery, New York, NY, USA, 13–22. https://doi.org/10.1145/2517208.2517211

Matthew Flatt, Ryan Culpepper, David Darais, and Robert Bruce Findler. 2012. Macros that Work Together - Compile-time bindings, partial expansion, and definition contexts. *J. Funct. Program.* 22, 2 (2012), 181–216. https://doi.org/10.1017/S0956796812000093

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.

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

Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream fusion, to completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 285–299. https://doi.org/10.1145/3009837

Oleg Kiselyov, Yukiyoshi Kameyama, and Yuto Sudo. 2016. Refined Environment Classifiers - Type- and Scope-Safe Code Generation with Mutable Cells. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 10017)*, Atsushi Igarashi (Ed.). 271–291. https://doi.org/10.1007/978-3-319-47958-3_15

Eugene Kohlbecker, Daniel P Friedman, Matthias Felleisen, and Bruce Duba. 1986. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*. 151–161.

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.

Lexifi. 2023. LRT: LexiFi runtime types. https://github.com/LexiFi/lrt Online; accessed March 2023.

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

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

Lionel Parreaux, Antoine Voizard, Amir Shaikhha, and Christoph E. Koch. 2018. Unifying analytic and statically-typed quasiquotes. *Proc. ACM Program. Lang.* 2, POPL (2018), 13:1–13:33. https://doi.org/10.1145/3158101

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

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

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

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.

Benoît Vaugon. 2013. A new implementation of OCaml formats based on GADTs. OCaml Users and Developers workshop.

Jerome Vouillon. 2023. Re: A regular expression library for OCaml. https://github.com/ocaml/ocaml-re Online; accessed March 2023.

Liang Wang and Jianxin Zhao. 2022. *Architecture of Advanced Numerical Analysis Systems: Designing a Scientific Computing System using OCaml.* Apress Berkeley, CA. ISBN 978-1-4842-8852-8.

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

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

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. 2018. A modular foreign function interface. *Sci. Comput. Program.* 164 (2018), 82–97. https://doi.org/10.1016/j.scico.2017.04.002