The Next Stage of Staging

Jun Inoue¹, Oleg Kiselyov², Yukiyoshi Kameyama³

¹jun.lambda@gmail.com
²Tohoku University
³University of Tsukuba

Abstract This position paper argues for type-level metaprogramming, wherein types and type declarations are generated in addition to program terms. Term-level metaprogramming, which allows manipulating expressions only, has been extensively studied in the form of staging, which ensures static type safety with a clean semantics with hygiene (lexical scoping). However, the corresponding development is absent for type manipulation. We propose extensions to staging to cover ML-style module generation and show the possibilities they open up for type specialization and overhead-free parametrization of data types equipped with operations. We outline the challenges our proposed extensions pose for semantics and type safety, hence offering a starting point for a long-term program in the next stage of staging research. The key observation is that type declarations do not obey scoping rules as variables do, and that in metaprogramming, types are naturally prone to escaping the lexical environment in which they were declared. This sets next-stage staging apart from dependent types, whose benefits and implementation mechanisms overlap with our proposal, but which does not deal with type-declaration generation. Furthermore, it leads to an interesting connection between staging and the logic of definitions, adding to the study’s theoretical significance.

1 Introduction and Motivation

Metaprogramming, specifically program generation, has been successful in performance-critical areas such as high-performance computing [22, 14] and hardware circuit design [10]. Program generation enables “abstraction without guilt”, where a program specification written at a high level of abstraction generates a low-level, efficient program, eliminating the cost of abstraction. A particularly well-developed program-generation framework is staging, which gives programmers complete control over which computations happen statically (during the program-generation phase) and dynamically (during the execution phase of the generated code). Staging has two important advantages:

- Well-studied type safety properties [20, 8].
- Well-behaved semantics [6] with hygiene (i.e. generated code obeys lexical scoping).

These properties make staging ideal not only for writing generators directly but also for conferring similar benefits to other code generation systems. For instance, staging has been used as a basis for modular, type-safe macros [19, 5] and compiler optimization passes [15].

These assurances have so far been limited to term-level staging, which manipulates only terms and not types or type declarations. However, generating type-level objects – types, type declarations, and modules (which include type declarations) – can significantly extend the reach of program generation. A prominent example is guilt-free parametrization of data types equipped
with operations, such as the set type equipped with a membership test operation. First-class ML-style modules and functors allow unrestricted parametrization and composition of such types, but they have runtime overhead. Staging with modules can eliminate this cost of functors, just as traditional staging removes the overhead of term-level functions under the slogan “abstraction without guilt” by specializing and inlining them. Existing systems like TemplateHaskell or C++ templates can achieve similar effects, but they do not guarantee type safety of the generated code (until the elaboration time), and the semantics is not precisely defined or is overly complicated. Bringing the safety assurances of staging will have important benefits.

The purpose of this position paper is to show that the pursuit of type safety and clean semantics with hygiene for type-level staging is both useful and interesting, thus deserving attention as the next stage of staging research. We illustrate its promises along with challenges and connections to other fields, suggesting how we can proceed with this research program. In particular, generating type-level objects, especially type declarations, poses challenges for type safety. As we explain later, type declarations do not have scopes in the way variables do, and generating them creates ways to violate hygiene. This distinction is reminiscent of the difference in behavior between constants and variables in the logic of definitions [12], suggesting a strong link between the two fields.

1.1 Contributions

We propose extensions to the staging facilities in MetaOCaml (reviewed in section 2) to allow type and module generation, which can be used to implement guilt-free specialization of types with operations (section 3). We explain the challenges involved in laying down a clean, type-safe semantics for code generation with modules, as well as the relevance of this line of investigation to the logic of definitions (section 4).

We stress that we are not exclusively interested in MetaOCaml or ML-style modules. Extending higher-level code generation interfaces like tagless-final encodings [2, 15] or automatic partial evaluation [7] to the type level will likely be just as profitable, and specializing types are also relevant in object-oriented languages. We are open to any other approach that can bring the same benefits while solving the challenges; however, MetaOCaml’s bracket-escape-run formulation is both simple and explicit, making semantic subtleties easier to uncover, discuss, and hopefully solve. First-class modules are also representative of the maneuvers required in type-level metaprogramming, making both types and associated terms first-class. Therefore, we believe MetaOCaml with modules to be a good model case for type-level metaprogramming, much in the way ML has been for issues on static type safety and type inference.

2 Background: MetaOCaml Primer

In this section, we briefly review staging as it currently works in MetaOCaml, along with safety properties that we’d like to preserve in its extension to modules. We assume the reader is familiar with OCaml, including its first-class module system. MetaOCaml adds three primitive constructs, called staging annotations, to OCaml. Brackets ⟨e⟩ delay the expression e. Escape ∼ e must occur inside brackets and exempts the expression e from the delay. This e must return a code value, a value of the form ⟨e’⟩, which is spliced into the surrounding code, i.e e’ replaces ∼ e. Run !. e is a library function that compiles and executes the code value produced by evaluating e.

The following example exercises all three constructs.

```ocaml
let rec power : int → int → int = fun n x →
  if n = 1 then x
  else x * power (n-1) x
let rec power_gen : int → int → code → int code = fun n x →
  if n = 1 then x
  else (∼x * (∼(power_gen (n-1) x)))
```
let power_st : int → int → int = fun n →
! (fun x → (∼ (power_gen n ⟨x⟩)))
let pow3 = power_st 3 (* evaluates to (fun x → x*x*x) *)

The power function computes \(x^n\). It works for all (positive) values of \(n\), but it wastes time on a function call and a branch before every multiplication. The power_gen function generates code with these overheads removed, producing straight-line code of the form \(x*x*x*\cdots * x\) given an \(n\) and an \(⟨x⟩\). The power_st function takes only an \(n\) and then generates and compiles a function whose body is the straight-line multiplication generated by power_gen. Notice that the multiplication operator \(*\) is a binary function defined outside of brackets (as a builtin) but is used inside brackets. This feature, where definitions in the generating code carry over to the generated code, is called cross-stage persistence (CSP).

Currently, MetaOCaml’s staging constructs are limited to term-level expressions that avoid interacting with modules. The expression enclosed in brackets must not contain a module expression or bind a local module. Consequently, escapes cannot occur within such expressions. Thanks to these restrictions, the effects-free subset of MetaOCaml enjoys static type safety and hygiene [20]. Static type safety means that the generated code is always well-typed if the generator type checks. By contrast, C++ templates, for instance, does not report type errors until templates are instantiated and compiled, often leading to latent discovery of bugs. Hygiene means that generated code obeys lexical scoping. For example,

let f x = ⟨let y = 0 in ∼ x⟩  in  ⟨let y = 1 in ∼ (f ⟨y⟩)⟩

returns ⟨let y1 = 1 in let y0 = 0 in y1⟩ and not ⟨let y = 1 in let y = 0 in y⟩. The ⟨y⟩ in the source code refers to let y = 1, because that is the lexically enclosing occurrence, and this link to let y = 1 is preserved when the open code ⟨y⟩ is moved into the scope of another binder for the y.

Type safety and hygiene in the presence of effects is somewhat murkier due to the problem known as scope extrusion, but methods for detecting it has been put forth both in the implementation and in the theoretical literature. In scope extrusion, variables in the generated code are floated out of their scope (whereas the foregoing example floated ⟨y⟩ into a different scope). For example, in the following program:

let smuggle = ref ⟨0⟩ in let _ = (fun x → ∼ (smuggle := ⟨x⟩; ⟨x⟩)) in !. smuggle (* returns ⟨x⟩ *)

the variable x originally referred to fun x but has moved out of that scope and become unbound. The resulting code ⟨x⟩ is meaningless and cannot be compiled or executed. MetaOCaml currently detects this problem at generation time, as soon as the context (fun x → ∼ ⟨⟩) has been popped off of the stack (i.e. during the execution of the second line in the example above, but before the third line begins executing). In the literature, several type systems have been proposed to detect this problem before ever running the generator, e.g. [8].

Both of these features are indispensable for safe, modular program generation. Type safety eliminates a large class of bugs, while hygiene makes open terms like ⟨y⟩ safe to carry around in different contexts without worrying about name clashes. As mentioned in the introduction, the purpose of this paper is to illustrate the utility of extending staging to cover modules and to map out the issues surrounding type safety and hygiene in that new context.

3 Module Specialization

In this section, we show a natural generalization of staging that extends the paradigm “abstraction without guilt” to functor applications. The motivating example is a simple implementation of sets (fig. 1). We show only the bare minimum API needed to make our point.
module type EQ = sig type t val eq : t → t → bool end
module type SET = sig
type set (* the type of the set *)
type elt (* the type of its elements *)
val member : elt → set → bool end
module MakeSet(Elt:EQ) : SET with type elt = Elt.t = struct
  type elt = Elt.t
  type set = elt list
  let rec member : elt → set → bool = fun x → function
  | [] → false
  | h:: t → Elt.eq x h || member x t end
module IntSet = MakeSet(struct type t = int let eq = (= ) end)

Figure 1. Set functor example.

The functor MakeSet is an example of a parametrized data type that takes type parameters carrying operations. In this example, the element type must come equipped with a comparison operation. In IntSet, the repeated calls to this comparison operation, eq, involve indirections through the runtime representation of Elt, which forces (= ) to be called through a computed jump (in fact, through the FFI). This indirection is quite a waste. In principle, a single integer comparison instruction should suffice.

To eliminate this overhead and make the set abstraction guilt-free, we’d like to leverage staging to inline Elt. However, Set is quite unlike the power function in that we want to specialize not just to the term Elt. eq but also to the type Elt.t – in other words, we want to inline a type application. This is the critical reason why MetaOCaml has so far stayed clear of staging modules: the safety implications of manipulating code values containing types are unclear. However, as we argue in this paper, expending the effort to support such an extension is worthwhile.

Let us hypothesize that MetaOCaml has been extended to allow the code in fig. 2. The staged

module type EQ’ = sig type t val eq : t code → t code → bool code end
module MakeSetGen(Elt:EQ’ code) : (SET with type elt = Elt.t) code =
  ⟨ struct
    type elt = ∼(Elt.t)
    type set = elt list
    let rec member : elt → set → bool = fun x → function
    | [] → false
    | h:: t → ∼(Elt.eq ⟨ x ⟩ ⟨ h ⟩ ) || member x t end⟩
module IntSet = !. MakeSetGen(
  ⟨ struct type t = int let eq x y = ((x:int) = (y:int)) end ⟩)

Figure 2. Staged set functor.

functor MakeSetGen takes a code of module Elt: EQ’ code and splices its components into a new code of module, then runs the result to get a compiled module object. The type Elt.t is spliced (or inlined) into the returned module, while the comparison function Elt.eq is modified to map code values to code values, just as power from section 2 was modified into power_gen. The module type is thus changed from EQ to EQ’. The staged functor is then invoked and the result is run to create a compiled module IntSet, which looks like:

struct
  type elt = int
type set = elt list

let rec member : elt → set → bool = fun x → function
  | [] → false
  | h:: t → (x:int ) = (h:int )) ) || member x t end)
end

The overhead of calling into a separate module has been eliminated, and the comparison now takes a single machine instruction.

The reader may have noticed typing issues in fig. 2. The input Elt is a code of module and not a module of code, so by a naïve interpretation, accesses like Elt.t or Elt.eq should be type errors. In general, aggregate types like tuples, records, and modules can be manipulated in staging as either delayed aggregates (e.g. (int * int) code) or non-delayed aggregates of delayed values (e.g. (int code) * (int code)). Non-delayed aggregates are almost always desirable because accesses to their components have runtime overhead that we’d like to eliminate at generation time, and indeed that’s the prevailing practice in conventional staging. However, with modules this trick is less desirable because module components can be inter-dependent. For example, the type of Elt.eq is Elt.t code → Elt.t code → bool code, which is nonsensical if the module Elt is out of scope. Thus, Elt.eq is not a free-standing component: it cannot be extracted and used independently of the rest of the structure. We will discuss the implications of this issue in more detail in section 4.2, but for the time being let us assume that a delayed module’s components can be accessed like Elt.t, despite the delay.

Typing issues aside, two things are different from conventional staging in fig. 2. Firstly, modules appear under brackets and the code type, and the resulting code can be compiled via the ! function. Secondly, a type (Elt.t) is spliced into code where a type expression is expected. These extensions clearly harbor difficult questions for semantics, hygiene, and type safety. Yet, these problems deserve to be tackled, as balancing expressivity with avoidance of indirection is something compiler writers have traditionally struggled to provide. OCaml calls eq indirectly for even the simple code in fig. 1, but the pervasive indirection helps to implement first-class modules. SML, by contrast, has a deliberately second-class module system, but this allows, for example, the MLton compiler to support a defunctorization pass that eliminates all functors at compile time [21]. Formal studies on how much this matters in practice are scarce, but reportedly engineers at Jane Street – an industrial heavy user of OCaml – have had to avoid the use of OCaml functors for performance reasons [13]. Outside the ML family, object-oriented languages also routinely access objects’ members indirectly, which helps to keep those members first-class, at a certain cost to performance.

A well-designed staged language should enable programmers to overcome this dilemma between expressivity and performance, not just for term-level computation but also computations involving types, as demonstrated above with the set functor. In the next section, we will summarize what problems arise with the extensions, along with potential lines of attack.

4 Challenges

We have advocated two extensions to staging in MetaOCaml: allowing modules under brackets, and escaping (or splicing) type expressions. As simple and useful as they are, these extensions also raise difficult questions for semantics and type safety. In this section, we summarize the challenges that must be addressed for making type-safe type-level staging a reality, discussing possible approaches. There are three major issues: the meaning of hygiene when splicing binders, typing types, and scope extrusion of types. Let us consider each of these in turn.

1 Confirmed by inspecting the assembly code emitted by ocamlopt 4.01.0 and 4.02.1.
4.1 Splicing Binders

Splicing types like \(\text{struct type } t = \sim(M.t)\) is a very different beast from term-level escapes because the spliced type \(M.t\) could introduce constructors and field names. For example, if \(M.t\) is declared as \(\text{type } t = \text{Foo} | \text{Bar}\), then the result of the splice is \(\text{struct type } t = \text{Foo} | \text{Bar end}\), binding two names \text{Foo} and \text{Bar} that did not exist at the splice site in the source program. In other words, type splices can introduce new bindings at the escape – a phenomenon that never happens with conventional escapes. Binder-splicing also arises in module splicing, a subsidiary feature which we have not discussed due to its ad-hoc nature. For example, after generating the \text{IntSet} module in section 3, if we have to qualify all accesses to member functions of this module, we have a new source of indirection. To have guaranteed indirection-free access, we may want a minor enhancement to the extension for splicing modules into modules:

\[
\text{module UserProgram = struct}
\sim(\text{IntSet}) \text{ (* Splice IntSet’s bindings into UserProgram *)}
\text{let } f x = \text{ (* Use IntSet’s functions here *) end}
\]

The splice works essentially like \text{include}, but without introducing indirections.

The question here is this: what does hygiene mean for splicing binders? Should the names bound by splicing \text{IntSet} be renamed, so that they are local to the host module \text{UserProgram}? Or should they be exported from \text{UserProgram} unless masked by a signature? These questions never arose in conventional staging because the names generated invariably had limited scope. By contrast, bindings in modules are visible globally (in qualified form). The first challenge in laying down a coherent semantics for staging with modules is to sort out what needs to be renamed and how, when binders are spliced. The simplest solution is to not rename anything, which makes some sense because names exported from modules are typically not subject to \(\alpha\) conversion. Determining how desirable that solution is requires more extensive experimentation, however.

4.2 Representing and Accessing Types

A type splice like \(\sim(\text{Elt}\cdot t)\) requires \text{Elt} to carry a runtime representation for the type expression \(\text{Elt}\cdot t\). Fortunately, MetaOCaml’s \text{code type} (implemented by ASTs under the hoods) can readily serve as this representation. For example, given the invocation

\[
\text{MakeSetGen (\langle \text{struct type } t = \text{int let } eq = \ldots \text{ end} \rangle)}
\]

in fig. 2, we can consider the splice \(\sim(\text{Elt}\cdot t)\) in the body of \text{MakeSetGen} as reducing to \(\sim(\text{int})\) then to \text{int}. The code-of-module \text{Elt} will store \(\langle \text{int} \rangle\) as a representation of \(\text{Elt}\cdot t\), just like it stores term-level code values. However, this treatment of type splices creates a tension with module-component accesses. Strictly speaking, for accesses like \(\text{Elt}\cdot t\) to be well-typed, we need \text{Elt} to be a non-delayed module. However, if the module is not delayed, then OCaml will not keep a runtime representation of the type \(\text{Elt}\cdot t\), as per its usual treatment of modules.

Note that the need for a runtime representation of type expressions necessitates type splices, as opposed to CSP on types. A generator like the following, where the incoming module is not code of module but a present-stage module living outside of brackets, and where \(\text{Elt}\cdot t\) is injected into the generated module via CSP, doesn’t work:

\[
\text{module type EQ’ = sig type } t \text{ val eq : } t \text{ code } \rightarrow t \text{ code } \rightarrow \text{bool code end}
\]
\[
\text{module MakeSetGen(Elt:EQ’) : (SET with type elt = Elt.t) code =}
\]
\[
\text{!. (struct type elt = Elt.t ... end)}
\]

If we want the CSP’ed \(\text{Elt}\cdot t\) to be replaced by the definition of \(t\) (like \text{int}), then we have to be able to reconstruct at runtime the type expression to which \(\text{Elt}\cdot t\) is bound. At the least, this reconstruction requires modules to carry runtime information about the types declared within
it. Not only would this require a major change to the runtime of the base language (OCaml), it would also add some overhead of its own, defeating our purpose of providing guilt-free functors.

Elt.eq poses a similar challenge. If the module is delayed, it is most natural for Elt.eq to have type \((\text{int code} \to \text{int code} \to \text{bool code})\) code, but its usage in MakeSetGen requires \(\text{int code} \to \text{int code} \to \text{bool code}\). If we declared \(\text{eq : int} \to \text{int} \to \text{bool}\), we could avoid the type error at Elt.eq:

```ocaml
module type EQ = sig type t val eq : t \to t \to bool end
module MakeSetGen(Elt:EQ code) : SET with type elt = Elt.t =
  struct
    type elt = ~(Elt.t)
    let rec member : elt \to set \to bool = fun x \to function
    | [] \to false
    | h: t \to ~(Elt.eq) x h || member x t end
```

But this generator is unsatisfactory: Elt.eq is now directly spliced, with no inlining of its body. In other words, Elt.eq needs to be a dynamic component that performs the inlining. Having the whole module be static is therefore undesirable.

Thus, a module parameter to a code generator has two conflicting requirements. The types in the module force the module to be delayed, while (some) functions in the module prefer a non-delayed module. These requirements constitute the second challenge for type-level staging: what is the right interface for a module that mixes static and dynamic elements? How should we access its components? There are several potential solutions, each with pros and cons.

One approach is to commit to manipulating module of code throughout code generation, and to provide ways to have explicit type representations inside modules. For example, we can allow

```ocaml
struct (type t = int) let eq : t code \to t code \to bool code end
```

so the module itself is not delayed, but its type component is, as well as parts of the body of eq. This design is elegant in that it fits well into the broader theme of delayed aggregates vs. non-delayed aggregates with delayed components. It completes the facility for staging modules with a means to explicitly delay the type component. However, it’s unclear if we can safely mention delayed types within non-delayed values like eq. Because it provides fine-grain control over delaying type-containing modules, we expect its disruption to the type system to be comparably subtle and challenging. Moreover, as mentioned before, the component eq has a dependence on t, so it cannot be freely extracted from the module, to be manipulated independently. Whether or not we can enforce safe usage without disrupting free access to modules that do not involve staging remains to be investigated.

Another approach is to allow M.foo to be valid for code-of-module M in addition to module-of-code. This approach is conceptually simple, but it is difficult to ensure that the result is well-defined. For instance, if M is a functor application like \(\langle F(X) \rangle\), what should M.foo access? Clearly we do not want to execute the functor application just for this access. Perhaps we can distinguish between code of the form \(\langle \text{struct} \ ... \ \rangle\) which starts with the struct keyword from those that do not. By introducing modules we are already forced to introduce a distinction between statements (such as type declarations) and expressions (like fun x \to ... ), so we may as well also distinguish static occurrences of struct from other module expressions. This approach does not address the issue with Elt.eq having type \((\text{int code} \to \text{int code} \to \text{bool code})\) code. One workaround, which does not require any further extensions, is to run Elt.eq in the generator, like

```ocaml
module MakeSet(Elt:EQ code) : SET with type elt = Elt.t =
  struct
    type elt = ~(Elt.t)
    let rec member : elt \to set \to bool = fun x \to function
```
It remains to be seen how well this approach generalizes.

4.3 Scope Extrusion of Types

Even in the absence of staging, types in local modules are prone to scope extrusion.

```
| []    → false
| h:: t → ~(!. Elt . eq ⟨x⟩ ⟨(h)⟩ ) || member x t end)
```

OCaml’s type checker rejects this code. The type of the whole `let module` expression is `M.t → M.t`, but this is nonsensical because `M` is unbound outside of the body of `let module` – the type `M.t` extrudes from its scope. OCaml accepts a `let module` if all occurrences of `M.t` in the return type is hidden, like:

```
let module M : sig type t end = (struct type t = int end) in fun (x : M.t) → x
```

The overall type no longer mentions `M.t`, so OCaml accepts this code. Unfortunately, type splicing makes this workaround more questionable.

```
let module M : (sig type t end) code = ⟨(module struct type t = int end) in
(module struct type t = ~(M.t) end : Type)⟩
```

When the generated code is run, we must somehow evaluate the code, which refers to `M.t`, in an environment where `M` has gone out of scope. In this sense, `M.t` has extruded from its scope. Like in the preceding example, `M.t` does not appear in the return type of the `let module`, so the type system cannot reject this code without a major overhaul.

The hidden reference to `M.t` was not a problem in the absence of staging because types would be erased completely during compilation. It was not a problem if the returned value contained runtime values of type `M.t`, because that type information will have become irrelevant by the time the program runs. With staging, however, code generation could call for a second round of compilation, so the type environment must somehow be preserved in the code. While this idea entails an implementation challenge, it is also not obvious how this behavior can be modeled in a type safety proof of the resulting system. Perhaps local module bindings can be stacked onto the evaluation context, similar to how call-by-need λ calculi maintain a heap of lazy bindings [1]. It would also be interesting to see if MetaOCaml’s dynamic scope extrusion check can be extended to detect extrusion of types [9].

4.4 Connection to Logic

It is worth pointing out an intriguing connection between scope extrusion of types in modules, scope extrusion in term-level staging, and eigenvariables (a.k.a. local constants) in logic. Extrusion is a manifestation of the fact that types and future-stage variables\(^2\) behave as constants rather than variables. In logic, an eigenvariable is (despite its name) a constant introduced during the proof of a universal statement. For example, \(\forall x.\exists y.x \neq y\) can be proved by extending the logic with a fresh constant `c`, the eigenvariable, proving \(\exists y.c \neq y\) in the extended logic, and then transporting the proved theorem back into the original logic. This approach is different from proving the open formula \(\exists y.x \neq y\), which consists of proving all closed instances without changing the logic.

In logic, a variable is a placeholder for substitution, designating a hole in the formula waiting to be filled with a concrete term. It is a syntactic entity, not associated with any fixed object in the universe of discourse, so it makes no sense to ask within the logic if two variables `x` and `y` are distinct. One can only ask if the values substituted for them are distinct, so the formula `x \neq y` cannot be settled true nor false without a valuation for the variables. By contrast, a constant is

\(^2\)A variable is future-stage iff its binder is delayed by brackets.
a fixed object in the universe of discourse – it is a semantic entity. A formula $c \neq d$ for constants $c$ and $d$ has an intrinsic truth value, independent of any valuation.

Programming languages have a similar distinction. Variables are just syntactic entities, so they have intrinsic scope. Any construct that seems to move a variable out of its scope, like

```ocaml
let c = ref 0 in
let f x = c := x
```

actually only moves the value assigned to that variable. In this code, the variable $x$ itself is never stored in $c$ – the value substituted for $x$ is. As $x$ is merely a syntactic device that shows data flow in the body of $f$, the runtime which implements the data flow is never concerned with $x$. Thus, extrusion of $x$ is impossible, for it vanishes when the compiler translates syntax to meaning.

Constants, however, are semantic entities with a life outside of the syntax, so they have no a priori reason to obey any syntactic scoping discipline. Constants move to wherever computation takes them – if the code above said $c := 5$, then the 5 really would be stored in $c$.

Future-stage variables are like eigenvariables. Just as a proof of $\forall x. \phi x$ starts by extending (using the meta-logic) the object logic with a fresh constant $c$ and embarking on a proof of $\phi c$, evaluation of $\langle \text{fun } x \rightarrow \sim(f \langle x \rangle) \rangle$ starts by extending (in the meta-language) the object language with a fresh constant $\langle x \rangle$ by allocating a new runtime object in the heap and embarking on the evaluation of $f \langle x \rangle$. This is why, unlike ordinary variables, future-stage variables can extrude its scope. It has no obligation to respect any syntactic scoping rules, so given the wrong programming constructs it will indeed disobey the desired scoping.

Types in local modules are similar. Like constants, types have definite meanings that are sensible in any context. Hence given the wrong constructs, it will extrude any syntactic boundary set by scoping rules. In this sense, both kinds of extrusion are closely tied to the distinction between constants and variables. Miller and Tiu’s logic of definitions [12], which looked closely at the behavior of local constants, might therefore provide useful clues about scope extrusion problems. Conversely, scope extrusion considerations may shed new light on local constants and eigenvariables in logic. It will be interesting to see how the potential synergy unfolds.

5 Related Works

The main reason we feel justified with our extensions to MetaOCaml presented in this paper is that we have a promising line of attack at establishing type safety. On the one hand, Rossberg et al. gave the F-ing translation, which explains (second-class) modules in terms of existentials in system $F\omega$ [18]. This translation should reduce type safety of staging with modules to that of system $F\omega$ with staging, which has been investigated with Concoqtion [4]. Of course, we have to finesse the extensions to ensure that after the F-ing translation we are left with permissible uses of staging in staged $F\omega$, but this is nonetheless a promising lead and a useful design guide.

Rossberg has also recently presented preliminary work on 1ML, which extends the F-ing translation to first-class modules [17]. Incidentally, however, 1ML incurs performance penalties for functor abstraction. In fact, functors become regular functions. We believe that staging could be a good complement to this work: 1ML can provide a means to establish type safety for staging with modules, while staging could recover guilt-free functors in 1ML.

Type manipulation comes naturally with dependent types, and their goals and mechanisms overlap with ours. However, type-level staging has a strict phase separation between the type and term levels, which is essential for providing abstraction without guilt. Agda, Coq, and other systems similar to Martin-Löf’s intuitionistic type theory [11], by contrast, allow computations on types and terms to be freely mixed, which is essential for using them as logics but can make performance harder to predict. ATS [23] and Concoqtion [4] provide type-level programming with phase separation from the term-level and are closer to our work. Module generation, however, has not been addressed in these languages.
Rompf et al. [16] have successfully applied data structure optimizations based on a higher-level formulation of staging. Their goals appears to be less ambitious than ours for the systematic, type-safe manipulation of types. Perhaps our research could find use in extending theirs, while we could hope to learn from their more practical experiences. A systematic encoding of universes in dependent type theory is proposed by Chapman et al. [3], which is also suitable for type generation and type-level programming. One thing that has not been investigated in depth there is nominal type distinctions. They do not distinguish types with the same implementation, so for example, the set implementation in section 3 cannot be distinguished from lists. With modules, however, abstracted types are considered distinct from all other types. Investigating type-level metaprogramming via modules may therefore help achieve nominal separation of data in frameworks like that of Chapman et al.

6 Conclusion

We have shown the promises and challenges for type-level metaprogramming via staging with modules. We have advocated the following extensions to the conventional bracket-escape-run formulation of staging: generation of code of modules and splicing types into modules. These extensions introduce splicing of binders, the issue of representing and accessing types, and a new kind of scope extrusion for types. As we have shown, extending staging to modules has applications that are not handled well with conventional, term-level staging. Furthermore, the issue of scope extrusion has deep connections to logic, which would be interesting to clarify in its own right. Considering all these facts, we believe that pinning down a type-safe, hygienic semantics for type-level metaprogramming, specifically staging with modules, is suitable for a grand challenge in the next stage of staging research.

Acknowledgments. We thank the anonymous PPL reviewers for their comment which helped to improve the presentation of this paper.

References


