Staging Beyond Terms: Prospects and Challenges

Jun Inoue
National Institute of Advanced Industrial Science and Technology, Japan
jun.inoue@aist.go.jp

Oleg Kiselyov
Tohoku University, Japan
oleg@okmij.org

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

Abstract

Staging is a program generation paradigm with a clean, well-investigated semantics which statically ensures that the generated code is always well-typed and well-scoped. Staging is often used for specializing programs to the known properties or parts of data to improve efficiency, but so far it has been limited to generating terms. This short paper describes our ongoing work on extending staging, with its strong safety guarantees, to generation of non-terms, focusing on ML-style modules. The purpose is to map out the promises and challenges, then to pose a question to solicit the community's expertise in evaluating how essential our extensions are for the purpose of applying staging beyond the realm of terms.

We demonstrate our extensions' use in specializing functor applications to eliminate its (currently large) overhead in OCaml. We explain the challenges that those extensions bring in and identify a promising line of attack. Unexpectedly, however, it turns out that we can avoid module generation altogether by representing modules, possibly containing abstract types, as polymorphic records. With the help of first-class modules, module specialization reduces to ordinary term specialization, which can be done with conventional staging. The extent to which this hack generalizes is unclear. Thus we have a question to the community: is there a compelling use case for module generation? With these insights and questions, we offer a starting point for a long-term program in the next stage of staging research.

Categories and Subject Descriptors D.3.2 [*Language Constructs and Features*]: Applicative (functional) languages; D.3.3 [*Language Constructs and Features*]: Modules, packages

General Terms Languages

Keywords Multi-stage programming, First-class modules, Type systems, Type optimization, Functional programming

1. Introduction and Motivation

Program generation has been successful in performance-critical areas such as high-performance computing [11, 20] and hardware circuit design [6]. Its main benefit is "abstraction without guilt", where a program written at a high level of abstraction generates low-level, efficient code, eliminating the cost of abstraction. Staging is an especially well-developed program generation framework, which gives programmers complete control over which computations happen statically (during the program-generation phase) and which happen dynamically (during the execution phase of the generated code). Staging has well-behaved semantics [4] with hygiene (i.e. generated code obeys lexical scoping) and well-studied type safety properties [5, 18]. These properties make staging ideal both as a tool for writing generators and as a theoretical foundation for other code generation systems. For instance, staging has been used as a basis for modular, type-safe macros [3, 17] and compiler optimization passes [12].

However, these firm foundations of staging have so far extended only to *term-level* staging, which manipulates only terms. In practice, most languages come with non-term constructs, such as types, type declarations, modules, and other forms of declarations. Generation of such non-term constructs are widespread, and it currently suffers from a lack of strong foundations. For example, the Mirage configuration tool [7] generates modules by concatenating strings, which is extremely error-prone. Generating class definitions is pervasive in C++ template programming, while a number of Haskell libraries like the popular `lens` package use TemplateHaskell [16] to generate type class instance declarations. In all of these uses, type checking (or in some of them, even syntax checking) is deferred until generation is complete, leading to latent discovery of bugs and difficulties tracing errors back to the parts of the source code that caused them.

Thus, the problem of giving clean, type-safe semantics to program generation that goes beyond term generation deserves attention as the next stage of staging research. As a first step toward this goal, we are studying the type-safety of module generation for MetaOCaml. Derived from OCaml, MetaOCaml is an implementation of staging that is among the most faithful to the theory. As modules contain type declarations and more or less all of the constructs not covered by conventional staging, studying the safety and semantics of these extensions amounts to studying all sorts of generation that are currently missing in staging.

Generating modules 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 predicate. 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. In effect, module generation enables type generation.

As is customary for short papers, this article is aimed at provoking discussions and posing questions. The specific purpose of this paper is twofold. Firstly, we explain the preliminary design of the extensions, mapping out the challenges in making it work and identifying a plausible line of attack for ensuring its type safety. Secondly, we pose questions about its potential applications to type specialization.

More precisely, after reviewing staging as it currently works in MetaOCaml (Section 2), we demonstrate the proposed extensions in a guilt-free specialization of the MakeSet functor (Section 3). We summarize the safety challenges arising from the extension and describe how Rossberg's F-ing translation technique [14] could guide us to a solution (Section 4). At first glance, the extensions we propose seem to be essential for type generation, but surprisingly, we find that concrete use cases for type specialization or manipulation turn out not to need the extension. Thus we arrive at the question: Does module generation add anything new for the purpose of type manipulation? (Section 5)

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 (Table 1), to OCaml. Brackets $\langle e \rangle$ generate a code value, i.e. the syntax tree of the expression e. Escape $\sim e$ must occur inside brackets and exempts the expression e from the brackets. This e must return a code value, a value of the form $\langle e' \rangle$, which is spliced into the surrounding code, i.e. e' replaces $\sim e$. Run, written like $!\,e$, is a library function that compiles and executes the code value produced by evaluating e.

Construct	Syntax	Typing
Brackets	$\langle e \rangle$	$\langle e : \tau \rangle : \tau\ \mathsf{code}$
Escape	$\sim e$	$(\sim(e : \tau\ \mathsf{code})) : \tau$
Run	$!\,e$	$!\,(e : \tau\ \mathsf{code}) : \tau$

Table 1. Staging annotations.

The following example exercises all three constructs.

```
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→intcode →intcode =
  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 (* gives (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*...* 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.

MetaOCaml is hygienic, i.e. the generated code obeys lexical scoping. For example, consider

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

The ⟨y⟩ in the invocation f ⟨y⟩ refers to the binding y = 1, but when f is called, the code ⟨y⟩ is substituted into the scope of another binding for the name y, namely y = 0. MetaOCaml renames the y's so as not to confuse these two bindings. That is, the ⟨y⟩ keeps referring to y = 1, the lexically enclosing binding. The resulting code in this case is ⟨**let** y1 = 1 **in** **let** y0 = 0 **in** y1⟩ rather than ⟨**let** y = 1 **in** **let** y = 0 **in** y⟩.

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 [18]. Static type safety means that the generated code is always well-typed if the generator type checks. By contrast, for example, C++ templates do not report type errors until templates are instantiated and compiled, often leading to latent discovery of bugs.

Type safety and hygiene 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. It is thus of utmost importance to maintain these advantages as we extend staging to cover module generation.

3. Module Generation

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 the MakeSet functor (Figure 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 = intlet eq = (=) end)
```

Figure 1. Set functor example.

The module type SET is a signature for sets equipped with a membership predicate member. The functor MakeSet creates a SET module, and 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, which is implemented as a module with the signature EQ. The module IntSet is an instance of SET, with element type being int and the comparison operation the equality on int.

This natural implementation of IntSet has a performance problem. In IntSet, the repeated calls to the 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 the term Elt.eq but also 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 module generation: the safety implications of manipulating code values containing types are unclear. Though System F extended by a richer type system and staging has been studied [2], the fact that Elt.t is declared within the inlined structure takes the challenge to a new level. We will examine the challenges in Section 4, but let us first argue that expending the effort to meet those challenges is worthwhile.

```
module type EQ' = sig
  type code t
  val eq : t code → t code → bool code
end
module MakeSetGen(Elt:EQ') :
   (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 code t = ⟨int⟩
         let eq x y = ⟨(∼x: int) = (∼y:int)⟩ end)
```

Figure 2. Staged set functor.

Let us hypothesize that MetaOCaml has been extended to allow the code in Figure 2. The staged functor MakeSetGen takes a module Elt : EQ' that carries a type code, a new construct introduced by the keyword **type** code. Conceptually, this is code of a type expression. The MakeSetGen functor splices the type code and the accompanying operation eq into a code of module, then runs the result to get a compiled module object. The type code 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 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.

Two things are different from conventional staging in Figure 2. Firstly, modules appear under brackets and the code type, and the resulting code can be compiled via the ! function. Secondly, a plain (i.e. non-code) module can contain a type code field, in this case Elt.t, which is spliced into code contexts where a type expression is expected. A type code serves as the runtime representation of the type, which is necessary to splice them at runtime. As we will examine in more detail in the next section, attempts to avoid introducing this new type code concept lead to dead ends.

These extensions harbor difficult questions for semantics, hygiene, and type safety. However, these problems deserve to be tackled, as a good balance between expressivity and avoidance of indirection is something compiler writers have traditionally struggled to provide. OCaml calls eq indirectly for even the simple code in Figure 1,[1] but the pervasive indirection helps to implement first-class modules. SML, by contrast, has a deliberately second-class module system, but this allows clever compilers like MLton to support a defunctorization pass that eliminates all functors at compile time [19]. 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 [9]. Outside the ML family, object-oriented languages also routinely access objects' members indirectly through references instead of inlining them, 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.

4. Challenges

We proposed two extensions to staging in MetaOCaml: allowing modules under brackets and letting modules carry type code that can be spliced into larger type expressions inside generated code. As simple and useful as they are, these extensions also raise difficult questions for semantics and type safety that we found to be much subtler than initially anticipated. In this section, we summarize the design rationale behind the extensions, along with the challenges that must be addressed for making type-safe type-level staging a reality. We then propose a possible approach to meeting those challenges.

In Figure 2, we typed the input as a plain (i.e. non-code) module Elt : EQ', and then declared that Elt.t is a new kind of member, type code. There are two other ways to type the input module that may seem superficially just as viable, but both turn out to be dead ends. The first option is to type the input as Elt : EQ' code, so that Elt.t is automatically code without the need for a new type code construct. However, with this design, not only would Elt.t be code of type, but also Elt.eq would be (Elt.t code → Elt.t code → bool code) code, instead of the desired Elt.t code → Elt.t code → bool code. This means that the splice ∼(Elt.eq ⟨x⟩ ⟨y⟩) doesn't work, which was the main motivation for specializing to int in the first place.

The second approach is to type the input as Elt : EQ' but without introducing a new construct **type** code. Instead, the type splice **type** elt = ∼(Elt.t) could be replaced by **type** elt = Elt.t. Just as we can refer to the top-level function = inside code, we allow referring to the type Elt.t inside code. But this scheme wouldn't work: there's no mechanism by which we can generate the appropriate code when evaluating the code-of-module expression ⟨**struct** ... **type** elt = Elt.t ... ⟩, because when the generator runs, the plain module Elt : EQ' would have lost all information about the type Elt.t. In (Meta)OCaml and in most statically typed languages, types are erased in the compiled code, so at runtime we have no way of reconstructing the type expression to which Elt.t is bound. If we want type information at run-time, we have to explicitly introduce a run-time representation for it – and that's exactly what **type** code is.

The **type** code construct does have a difficulty of its own. Because **type** code is not a genuine type, it makes no sense to classify values using **type** code. For example, we cannot allow

```
module F(M:...) = struct
  type code t = ⟨∼(M.t) ∗ int⟩
  let x : t = (0,0)
end
```

[1] Confirmed by inspecting the assembly code emitted by ocamlopt 4.01.0 and 4.02.1.

```
type α eq = α→ α → bool
let make_set (type a) : a eq →
(module SET with type elt = a) =
fun eq →
  (module struct
     type elt  = a
     type set  = elt list
     let singleton  x = [x]
     let rec member x = function
       | []  → false
       | y:: ys → eq x y || member x ys
   end)
```

Figure 3. Set functor example from Figure 1, but without the functor.

```
type α eq_code = α code → α code → bool code
let gen_set (type a) : a eq_code →
(module SET with type elt = a) =
fun eq →
  (module struct
    type elt  = a
    type set  = elt list
    let singleton  x = [x]
    let member =
      ! ⟨let rec member x = function
           | []  → false
           | y:: ys → ~(eq ⟨x⟩ ⟨y⟩) ||
                        member x ys in member⟩
   end)
```

Figure 4. Set functor example, without module generation.

to be well-typed because there's no way to check statically that the fst of x has type M.t. Instead, what we do allow is to classify code values with type code. Syntactically, a type code should be used inside code only. Thus, when t is type code, type expressions like t code or (t ∗ t list) code are allowed, but not t list or t ∗ t list, with no code surrounding them.

As these discussions show, there are clearly great difficulties in devising a static type system for the proposed extensions. In fact, devising a coherent semantics is already a challenge. Nonetheless, we feel these extensions are justified because we have a promising line of attack on these problems. The main ingredient is Rossberg et al.'s "F-ing translation", which explains (second-class) modules in terms of existentials in System $F\omega$ [15]. By applying this translation to staging with modules, we expect to reduce its type safety and semantics issues to those of system $F\omega$ with staging. System $F\omega$ with staging, in turn, has been investigated already in the form of Concoqtion [2]. By finessing the extensions so that after the F-ing translation we are left with permissible uses of staging in staged $F\omega$, we expect to establish a sound, clean semantics for our extensions.

5. Unexpected Solution

We have seen that staging the functor application in Figure 1 has many unanswered challenges. It seems there is no hope even to contemplate the specialization of SET in the current MetaOCaml. And yet we've hit upon an unexpected solution, using first-class modules and ordinary (term-level) staging. Moreover, the solution does not mention any module expressions (informally, any **struct**) in brackets and so it works in the current MetaOCaml. This section describes the solution and §7 reflects on its ramifications.

The key idea is replacing the MakeSet : EQ →SET functor with the ordinary function

```
make_set : α eq → (module SET with type elt = α)
```

as shown in Figure 3. We can specialize make_set to the statically known equality predicate α eq in the same way we specialized power n x to the statically known n. The result is gen_set shown in Figure 4.

In general, we represent the functor F in

```
module type Ssig = sig
  type t
  val m1: s1  val m2: s2 ...
end
module F: functor(S:Ssig) → struct ....  end
```

(where the types s1, s2, etc, may contain the abstract type t) with a function f

```
type α srec  = {m1: s1; m2: s2; ...}
val f : α srec → (module struct ....  end)
```

(where likewise the types s1, s2, etc. may mention the type parameter α, which takes place of t in Ssig). In other words, a module of type Ssig with an abstract type t is represented as a polymorphic record α srec. Informally, the existential, typically used to encode abstract types, is replaced with the universal.

What is most surprising is that the basis for this idea, of encoding abstract types by universals, can be found in the well-known paper by Mitchell and Plotkin [10] that helped to cement the direct opposite view of abstract types as existentials. The paper as a whole argues how existentials are the proper way of representing abstract types, but in section 3.8 it mentions the alternative encoding by universals (in the form of polymorphic records), crediting it to Reynolds. It is this alternative that we have just outlined.

Mitchell and Plotkin argue that Reynolds' encoding falls short of properly encoding the abstype construction they advocate. One of their arguments, that abstype is syntactically more expressive, does not apply in the case abstract types are represented as modules (which Mitchell and Plotkin mention themselves, in their discussion of the condition (AB.3)). The other argument, that Reynolds' encoding cannot be used if we want to choose an appropriate representation of the structure at runtime, does not apply to our case of module specialization. We explicitly do not want to defer the selection of the module representation to runtime: that is why we are generating specialized code in the first place. Thus Reynolds' encoding of modules with abstract types as polymorphic records covers all the applicable cases. The encoding clearly generalizes to modules with more than one abstract type.

Coming back to staging, the main feature of our solution in Figure 4 is that there are no type splices. All the types are explicit when type-checking the generator (at stage -1, so to speak). The code for the generated member function is ensured type-correct and does not need to be type-checked at later stages. There does not seem to be any compelling need to generate any splice types. All our attempts to come up with a realistic example of module generation that cannot be emulated with the techniques just outlined have so far come up empty. Thus, we have a question to the community: is there an example where module generation could make a difference, for the purpose of type manipulation?

6. Related Work

In Standard ML, which does not support recursive and first-class modules, all functor applications can be done at compile-time. In fact, the MLton compiler does exactly that. Compile-time functor applications generally cannot be done in OCaml, not only because of recursive, first-class and local modules but also because such applications do not work well with separate compilation. There are however preprocessors for OCaml that do some functor applications. The preprocessors do the best-effort functor applications and offer little user control. Mainly, they do not at all ensure any type or scope safety of the resulting code.

As explained before, 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: we apply Rossberg et al.'s F-ing translation [15] to reduce the system to system $F\omega$ with staging, and then check the results against a subset of Concoqtion [2]. Rossberg has also recently presented work on 1ML, which extends the F-ing translation to first-class modules [14]. 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 his work: 1ML can provide a means to establish type safety for staging with modules, while staging could recover guilt-free functors in 1ML.

An important use of module generation is type manipulation. This ability comes naturally with dependent types, and their goals and mechanisms overlap with ours. However, type-level staging via modules 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 [8], 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 [21] and Concoqtion [2] 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. [13] have successfully applied data structure optimizations based on a higher-level formulation of staging. Their goals appear 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. [1], 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.

7. Discussion and 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, bringing in the problem of representing and accessing types. These are complex questions, suitable for a grand challenge in the next stage of staging research.

However, once we started working on examples of module generation, it turns out that we can implement them already, in the existing MetaOCaml, or in MetaOCaml with small, easy extensions. Thus we have come against a deeper challenge: what are the *compelling* examples of module generation? If we want to assure the safety of the generated code at the generator time, aren't we compelled to expose all the types for the generator? One of the most important roles of types is to ensure abstraction. However, when we generate code, we can ensure abstraction boundaries during the code generation, in the generator. Therefore, there is nothing left to enforce in the generated code. We do not need abstract types to enforce abstraction in the generated code. Perhaps we do not even need modules in the generated code.

If we generate a typed language, we do need types since they guide the process of code generation (for example, checking of exhaustiveness of pattern-match and generating the default failure clause). Types are also needed for memory layout of data, etc. But those types, relevant for type generation, can be simple, and manifest types. Whereas the generator may deal with some abstract type `t`, the generated code may have `int`.

Thus, we have a question to the community: is there an example where module generation could make a difference, for the purpose of type manipulation?

References

[1] J. Chapman, P.-E. Dagand, C. McBride, and P. Morris. The gentle art of levitation. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 3–14, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. .

[2] S. Fogarty, E. Pašalić, J. Siek, and W. Taha. Concoqtion: Indexed types now! In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 112–121, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-620-2. .

[3] J. Gillenwater, G. Malecha, C. Salama, A. Y. Zhu, W. Taha, J. Grundy, and J. O'Leary. Synthesizable high level hardware descriptions: using statically typed two-level languages to guarantee verilog synthesizability. In *PEPM '08: Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 41–50, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-977-7. .

[4] J. Inoue and W. Taha. Reasoning about multi-stage programs. In *Proceedings of the 21st European Conference on Programming Languages and Systems*, ESOP'12, pages 357–376, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-28868-5. .

[5] Y. Kameyama, O. Kiselyov, and C.-c. Shan. Combinators for impure yet hygienic code generation. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14, pages 3–14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2619-3. .

[6] O. Kiselyov, K. N. Swadi, and W. Taha. A methodology for generating verified combinatorial circuits. In *EMSOFT '04: Proceedings of the 4th ACM International Conference on Embedded Software*, pages 249–258, New York, NY, USA, 2004. ACM. ISBN 1-58113-860-1. .

[7] A. Madhavapeddy, T. Gazagnaire, D. Scott, and R. Mortier. Metaprogramming with ML modules in the MirageOS. `http://sites.google.com/site/mlworkshoppe/Gazagnaire-abstract.pdf`, September 2014. ML Family Workshop 2014.

[8] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[9] Y. Minsky. Discussions on the caml-list mailing list, June 2007. `http://caml.inria.fr/pub/ml-archives/caml-list/2007/06/8517ef8cab9b778b7ded013f6a59c051.en.html`, last viewed January 2015.

[10] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3): 470–502, July 1988.

[11] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[12] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM*, 55(6):121–130, 2012. .

[13] T. Rompf, A. K. Sujeeth, N. Amin, K. J. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Lan-*

guages, POPL '13, pages 497–510, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1832-7. .

[14] A. Rossberg. 1ML – core and modules united (F-ing first-class modules). In *Proc. 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 35–47, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3669-7. .

[15] A. Rossberg, C. V. Russo, and D. Dreyer. F-ing modules. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '10, pages 89–102, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-891-9. .

[16] T. Sheard and S. P. Jones. Template meta-programming for Haskell. *SIGPLAN Not.*, 37(12):60–75, Dec. 2002. ISSN 0362-1340. .

[17] W. Taha and P. Johann. Staged notational definitions. In *GPCE '03: Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*, pages 97–116. Springer-Verlag New York, Inc., 2003.

[18] W. Taha and M. F. Nielsen. Environment classifiers. In *30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, pages 26–37, New York, NY, USA, 2003. ACM. ISBN 1-58113-628-5. .

[19] S. Weeks. Whole-program compilation in MLton. Online slides on MLton's official website, September 2006. `http://mlton.org/References.attachments/060916-mlton.pdf`, last viewed January 2015.

[20] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-89791-984-X.

[21] H. Xi. Applied Type System (extended abstract). In *post-workshop Proceedings of TYPES 2003*, pages 394–408. Springer-Verlag LNCS 3085, 2004.