The Design and Implementation of BER MetaOCaml

System Description

Oleg Kiselyov

University of Tsukuba, Japan
`oleg@okmij.org`

Abstract. MetaOCaml is a superset of OCaml extending it with the data type for program code and operations for constructing and executing such typed code values. It has been used for compiling domain-specific languages and automating tedious and error-prone specializations of high-performance computational kernels. By statically ensuring that the generated code compiles and letting us quickly run it, MetaOCaml makes writing generators less daunting and more productive.
The current BER MetaOCaml is a complete re-implementation of the original MetaOCaml by Taha, Calcagno and collaborators. Besides the new organization, new algorithms, new code, BER MetaOCaml adds a scope extrusion check superseding environment classifiers. Attempting to build code values with unbound or mistakenly bound variables (liable to occur due to mutation or other effects) is now caught early, raising an exception with good diagnostics. The guarantee that the generated code always compiles becomes unconditional, no matter what effects were used in generating the code.
We describe BER MetaOCaml stressing the design decisions that made the new code modular and maintainable. We explain the implementation of the scope extrusion check.

1 Introduction

MetaOCaml is a conservative extension of OCaml for "writing programs that generate programs". MetaOCaml adds to OCaml the type of *code values* (denoting "program code", or future-stage computations), and two basic constructs to build them: quoting and splicing. The generated code can be printed, stored in a file – or compiled and linked-back to the running program, thus implementing run-time code optimization.

MetaOCaml has been successfully used for specializing numeric and dynamic programming algorithms; building FFT kernels, compilers for an image processing domain-specific language (DSL), OCaml server pages; and generating families of specialized basic linear algebra and Gaussian Elimination routines, and high-performance stencil computations [1–4].

MetaOCaml is distinguished from Camlp4 and other such macro-processors by: hygiene (maintaining lexical scope); generating assuredly well-typed code;

and the integration with higher-order functions, modules and other abstraction facilities of ML, hence promoting modularity and reuse of code generators. A well-typed BER MetaOCaml program produces the code that shall compile without type errors. We no longer have to puzzle out a compilation error in the generated code (which is typically large, obfuscated and with unhelpful variable names). We illustrate these features in §2.

The original MetaOCaml was developed by Walid Taha, Cristiano Calcagno and collaborators [5] as a dialect of OCaml. Therefore, MetaOCaml took the full advantage of the OCaml's back-end code generation, the standard and other libraries, the top-level, etc. Alas, the divergence between the two languages has made integrating OCaml's new features and improvements progressively more and more difficult. Eventually MetaOCaml could no longer be maintained without a major investment.

BER MetaOCaml [6] is a re-design and the complete re-implementation of MetaOCaml, with different algorithms and techniques. It aims at the most harmonious integration with OCaml and lowering the barrier for contribution. The compatibility with OCaml becomes relatively easy to maintain, bringing better tools, better diagnostics, new libraries and new features to code generators. Contributors of new ways of running the generated code (e.g., translating it to C or LLVM) no longer need to be familiar with the OCaml internals and keep recompiling the system. The new goals of modularity and maintainability called for new code organization and design decisions. BER MetaOCaml also took advantage of the large experience with MetaOCaml, which prompted the drastic change of retiring environment classifiers and introducing the scope extrusion check.

Despite polar design decisions and the different implementation (BER and the old MetaOCaml share no staging-related code apart from parsing and pretty-printing), BER MetaOCaml does run the old MetaOCaml user code with little or no change. The implementation differences between the two systems are summarized in Appendix A of the full paper[1]. Here is the brief comparison from the user point of view:

- Whereas the old MetaOCaml is formalized by λ^i_{let} [10], BER MetaOCaml implements the classifier-less version of that calculus[2]. The translation from λ^i_{let} to the latter only affects types (which can be inferred); the two calculi have the same dynamic semantics.
- BER MetaOCaml requires user-defined data types be declared in a separate file, see §4.
- BER MetaOCaml accepts programs that could not be typed before, see §5.2, making it easier to use modules to structure generators.

[1] `http://okmij.org/ftp/meta-programming/ber-design.pdf`

[2] We drop single-classifier annotations of λ^i_{let} and replace a sequence of classifiers with the natural number denoting its length. Strictly speaking, we also have to replace the terms open e and close e of λ^i_{let} with just e. However, these terms never show up explicitly in the user-written code, so their disappearance is unnoticeable.

– Scope extrusion (see §2.1 for illustration) during code generation was not detected before. BER MetaOCaml detects it early and raises an exception.

All in all, a type-annotation-free old MetaOCaml program using standard data types will be accepted by BER MetaOCaml as it was. It will produce the same result (or raise the scope-extrusion exception).

Although this paper is a system description of BER MetaOCaml, it highlights the guidance from theory (§5.3 in particular), or the regrettable lack of it. Staging in the presence of user-defined data types, described in §4, is a thorny problem that seems to have not been addressed in any of the staged calculi. Implementing BER MetaOCaml thus has suggested directions for further theoretical research.

Specifically, our contributions are as follows:

– the specific approach of adding staging to OCaml that minimizes the changes to the base system (significantly, compared to the old MetaOCaml), making it easier to contribute to and maintain MetaOCaml and to keep it consistent with new revisions of OCaml (see §3 and App. A);
– constructor restriction, §4: a new trade-off in supporting values of user-defined data types within the generated code. The restriction markedly simplifies the implementation. The experience with the restriction (first introduced in the January 2013 release) showed its burden to be light, justifying the trade-off;
– scope-extrusion check, §5: detecting scope extrusion promptly at the generator run-time, aborting the code generation with an informative error message pointing to problematic locations in the generator source code. The positive experience with the check (again first introduced in the January 2013 release) led to the retirement of environment classifiers in the current version. BER MetaOCaml guarantees that the successfully generated code is well-typed and well-scoped. The guarantee is now unconditional: it holds even if the generator performed arbitrary effects, including delimited control effects.

We start with a brief introduction to MetaOCaml and finish, §6, with related work. BER MetaOCaml is available from OPAM, among other places [6].

2 The taste of MetaOCaml

This section introduces MetaOCaml and describes its features on very simple examples. §2.1 continues with a more realistic case, also explaining the need for control effects when generating code – and the accompanying danger of producing ill-scoped code.

Our first example is very familiar and simple, letting us focus on the MetaOCaml features used in its implementation. It centers on computing the n-th element of the Fibonacci-like sequence with the user-defined first two elements:

```
let rec gib n x y = match n with
  | 0 → x | 1 → y
  | n → let z = x + y in gib (n−1) y z
```

This ordinary OCaml code can be entered into MetaOCaml as it is since MetaOCaml is source- (and binary-) compatible with OCaml. If we are to compute gib n many times for a fixed n, we want to *specialize* gib to that n, obtaining the code that will later receive x and y and efficiently compute the n-th element of the sequence. We re-write gib annotating expressions as computed 'now' (when n is given) or 'later' (when x and y are given):

```
let rec sgib n x y = match n with
  | 0 → x | 1 → y
  | n → ⟨let z = ∼x + ∼y in ∼(sgib (n−1) y ⟨z⟩ )⟩
⇝ val sgib : int → int code → int code → int code = <fun>

let sgib4 = ⟨fun x y → ∼(sgib 4 ⟨x⟩ ⟨y⟩ )⟩ ;;
⇝ val sgib4 : (int → int → int ) code = ⟨fun x_1 → fun y_2 →
    let z_3 = (x_1 + y_2) in let z_4 = (y_2 + z_3) in
    let z_5 = (z_3 + z_4) in z_5⟩

(!. sgib4) 1 1;;
⇝ − : int = 5
```

The two annotations, or staging constructs, are brackets ⟨e⟩ (in code, `.<e>.`) and escape ∼e (in code `.~e`). Brackets ⟨e⟩ 'quasi-quote' the expression e, annotating it as computed later, or at the *future stage*. Escape ∼e, which must occur within brackets, tells that e is computed now, at the *present stage*, but produces the code for later. That code is spliced into the containing bracket. The plus + appearing in brackets is not a symbol: it is the identifier bound to the OCaml function (infix operator) (+): int →int →int. A present-stage bound identifier referred to in the future stage is called cross-stage persistent (CSP). CSP is the third, less noticeable feature of MetaOCaml.

The inferred type of sgib (printed by the MetaOCaml top-level) tells its result is not an int: rather, it is int code – the type of expressions that compute an int. Hence, sgib is a code generator. Its type spells out which argument is received now, and which are later: the future-stage arguments have the code type. The type of a future-stage code is known now – letting us type-check future stage expressions and the code that generates them, assuring that the generated code is well-typed. For example, if we replace (+) in sgib with the floating-point addition (+ .) or omit an escape, we see a type error with an informative diagnostic.

The expression sgib4 shows how to actually apply sgib to produce the specialized code and how to obtain the int code values to pass as the last two arguments of sgib. The code value ⟨x⟩ represents an open code: the free variable "x". We may store such variables in reference cells and pass them as arguments and function results. MetaOCaml hence lets us manipulate (future-stage) variables *symbolically*. We can splice variables into larger future-stage expressions but we cannot compare or substitute them, learn their name, or examine the already generated code and take it apart. This pure generativity of MetaOCaml helps maintain hygiene: open code can be manipulated but the lexical scoping is still preserved[3].

The inferred type of sgib4 shows it as a code expression that will, when compiled, be a function on integers. Code, even of functions, can be printed, which is what the MetaOCaml top-level did. The prefix operator !. lets us *run* sgib4, that is, to compile it and link back to our program. The result can be used as an ordinary int→int→int function.

Generating code and then running it is specializing a frequently used function to some data obtained at run-time, e.g., from user input. In our example, !. sgib4 is such a version of gib n x y specialized to n= 4. The sgib4 code is straight-line and can be efficiently compiled and executed. The generated code can also be saved into a file. Since the generated code is ordinary OCaml, it can be compiled with ordinary OCaml compilers, even ocamlopt, and later linked into various *ordinary* OCaml applications. Thus, MetaOCaml can be used not only for run-time specialization, but also for offline generation of specialized library code, e.g., of BLAS and Linpack libraries.

Since hygiene and lexical scoping is one of the two main topics of the paper (see §5), we illustrate it on another example – demonstrating the crucial difference between brackets and Lisp quasi-quotation. The example is a one-line generator, producing the code shown underneath:

```
    ⟨fun x → ~(let body = ⟨x⟩ in ⟨fun x → ~body⟩ )⟩
 ⇝ ⟨fun x_1 → fun x_2 → x_1⟩
```

Re-written in Lisp, with anti- and un-quotations, it generates the code

```
    ‘( lambda (x) ,( let ((body ‘x)) ‘( lambda (x) , body)))
 ⇝ ‘( lambda (x) (lambda (x) x))
```

with two indistinguishable instances of x, which denotes a different function. MetaOCaml maintains the distinction between the variables that, although identically named, are bound at different places. A variable in MetaOCaml is not just a symbol. We return to this topic in §5.

We have thus seen the five features that MetaOCaml adds to OCaml: brackets and escapes, CSP, showing and running code values. We will now see a realistic example of their use.

2.1 Code motion

This section gives a glimpse of a realistic application of MetaOCaml, generating high-performance numerical kernels. We demonstrate generating matrix-matrix multiplication with a loop-invariant code motion, i.e., moving the code not depending on the loop index out of the loop. We will see the need for delimited control, the actual danger of generating ill-scoped code, and how BER MetaOCaml alerts of the danger before it becomes too late. We will hence see the

[3] At first blush, the inability to examine the generated code seems to preclude any optimizations. Nevertheless, generating optimal code is possible [3, 7, 8].

scope extrusion check, explained in depth in §5. For the lack of space, the running example is presented schematically and in a less general form: see the complete code[4] and [8] for the explanation of the overall approach.

To generate a variety of specialized kernels and optimize them easily, we introduce a minimalist linear-algebra DSL (demonstrating how the abstraction facilities of OCaml such as modules benefit code generation):

```
module type LINALG = sig
  type tdom
  type tdim   type tind   type tunit
  type tmatrix
  val ( ∗ )      : tdom →tdom →tdom
  val mat_dim  : tmatrix → tdim ∗ tdim
  val mat_get  : tmatrix → tind → tind → tdom
  val mat_incr : tmatrix → tind → tind →  tdom →tunit
  val loop     : tdim → (tind → tunit ) → tunit
end
```

The abstract type tdom is the type of scalars, with the operation to multiply them; tdim is the type of vector dimensions (zero-based) and tind is the type of the index; tunit is the unit type in our DSL. The operation mat_get accesses an element of a matrix, and mat_incr increments it. The DSL lets us write the multiplication of matrix a by matrix b with the result in c (which is assumed zero at the beginning) in the familiar form[5]:

```
module MMUL(S: LINALG) = struct open S
  let mmul a b c = loop (fst (mat_dim a)) @@ fun i →
    loop (fst (mat_dim b)) @@ fun k →
      loop (snd (mat_dim b)) @@ fun j →
        mat_incr c i j @@ mat_get a i k ∗ mat_get b k j
end
```

With different implementations of LINALG, we obtain either functions for matrix-matrix multiplication (in float, int or other domains), or the code for such functions. For example, the following instance of LINALG produces the familiar matrix multiplication code

```
module LAintcode = struct
  type tdom = int code   type tdim = int code ...
  type tmatrix = int array array code
  let ( ∗ ) = fun x y → ⟨∼x ∗ ∼y⟩
  let mat_get a i j = ⟨(∼a).( ∼i ).( ∼j )⟩
  let loop n body = ⟨for i= 0 to ∼n−1 do ∼(body ⟨i⟩) done⟩
end
```

[4] `http://okmij.org/ftp/meta-programming/tutorial/loop_motion.ml`
[5] The infix operator @@ is a low-precedence application, introduced in OCaml 4.01.

We can do better: in MMUL.mmul the expression mat_get a i k does not depend on the index j of the innermost loop, and can be moved out. We extend our DSL with the operation

```
module type LINALG_GENLET = sig include LINALG
 val genlet : tind → (unit → tdom) →tdom end
```

One may think of genlet k (**fun** () →e) as memoizing the value of e with key k in a 1-slot memo table. We re-write mmul and manually introduce this memoization optimization:

```
module MMULopt(S: LINALG_GENLET) = struct open S
  let mmul a b c = loop (fst (mat_dim a)) @@ fun i →
    loop (fst (mat_dim b)) @@ fun k →
      loop (snd (mat_dim b)) @@ fun j →
        mat_incr c i j @@ genlet k (fun () → mat_get a i k) ∗
                             genlet j (fun () → mat_get b k j )
end
```

We extend LAintcode by adding genlet, the new realization of tind and the new implementation of loop. In thus extended LAintcode_opt, genlet k (**fun** () → ⟨e⟩) evaluates to a future-stage variable ⟨t⟩ bound by **let** t = e **in** ... inserted at the beginning of the loop with the index k. LAintcode_opt has to rely [9] on delimited control effects, provided by the library delimcc. MMULopt(LAintcode_opt).mmul then generates the following code

```
⟨fun a_7 b_8 c_9 →
   for i_10 = 0 to (Array.length a_7) − 1 do
     for i_11 = 0 to (Array.length b_8) − 1 do
       let t_14 = a_7.(i_10 ).( i_11 ) in
       for i_12 = 0 to (Array.length b_8.(0)) − 1 do
         let t_13 = b_8.(i_11 ).( i_12 ) in
         c_9.( i_10 ).( i_12 ) ← c_9.( i_10 ).( i_12 ) + t_14 ∗ t_13
  done done done⟩
```

The expressions to access the elements of a and b are let-bound; a is accessed outside the innermost loop. The code motion is evident.

The operation genlet is powerful but dangerous. If we by mistake instead of genlet j (**fun** () →mat_get b k j) in MMULopt.mmul write genlet k (**fun** () → mat_get b k j), we pull the code generated by mat_get b k j out of the innermost loop as well. The old MetaOCaml then produces:

```
⟨fun a_7 b_8 c_9 →
   for i_10 = 0 to (Array.length a_7) − 1 do
     for i_11 = 0 to (Array.length b_8) − 1 do
       let t_13 = b_8.(i_11 ).( i_12 ) in
       let t_14 = a_7.(i_10 ).( i_11 ) in
       for i_12 = 0 to (Array.length b_8.(0)) − 1 do
         c_9.( i_10 ).( i_12 ) ← c_9.( i_10 ).( i_12 ) + t_14 ∗ t_13
  done done done⟩
```

Although the generated code is simple, it is already hard to see what is wrong with it: as typical, variables in the generated code have unhelpful names. If we look carefully at the let-binding t_13, we see that the variable i_12, the index of the innermost loop, escaped its binding, creating the so-called scope extrusion. The escaped variable is unbound in the generated code above. More dangerously, it may be accidentally captured by another binding. The generated code will then successfully compile; the resulting bug will be very difficult to find. Since the scope extrusion has not been detected, it is hard to determine what part of the generator did it by looking only at the final result.

In contrast, BER MetaOCaml detects the scope extrusion with a good diagnostic. For example, executing the generator with the mistaken genlet aborts the execution when b_8.(i_11).(i_12) has just been moved out of the innermost loop, with the error that identifies the expression containing the escaped variable (the matrix element access), the name of the variable and where it was supposed to be bound (in the loop header). No bad code is hence generated. The exception backtrace further helps find the mistake in the generator[6].

We have seen the benefit of effects in code generation, for loop-invariant code movement. The same technique can also do loop interchange and loop tiling. We have also seen the danger of generating ill-scoped code and MetaOCaml's detecting the scope extrusion as soon as it occurs. The section hopefully has given the taste of generator abstractions; see the poster [8] for an elaborated example of using the OCaml module system to state an algorithm in a clear way and then apply various optimizations.

3 Design of BER MetaOCaml

This section briefly overviews the design of BER MetaOCaml and outlines our approach to implementing staging. The following two sections will explain in depth two particularly subtle issues, user-defined types and the scope extrusion check. Our guiding principle is to make MetaOCaml easier to maintain and use by making its changes to the OCaml code base smaller and modular.

MetaOCaml has to modify OCaml to extend its syntax with staging annotations and its type checker with the notion of the present and future stages. Unlike the original MetaOCaml, BER MetaOCaml tries to minimize the modifications and hence makes different design decisions, see below and §4. Whereas the original MetaOCaml was a fork, BER MetaOCaml is maintained as a set of patches to OCaml plus a library. Such an organization reflects the separation between the MetaOCaml 'kernel' and 'user-level'. The kernel (patched OCaml) is responsible for building and type-checking code values. The user-level processes closed code values, e.g., prints or runs them. As with the kernel/user-level separation in an OS, adding a new way to run code (e.g., to compile to Javascript) is

[6] The real linear-algebra DSL will unlikely offer genlet to the end-user. Rather, genlet will be incorporated into mat_get, where it could compare loop indices, determine which one corresponds to an innerer loop, and insert let appropriately. The scope extrusion may well happen however during the development of the DSL.

like writing a regular library, which requires no patching or recompilation of the MetaOCaml system. The separation lessens the maintenance burden and makes it easier to contribute to MetaOCaml.

Here is an example of how BER MetaOCaml minimizes changes to OCaml. For the most part, type checking is invariant of the stage (bracketing) level, with a notable exception [10]. Identifiers bound by future-stage binding forms should be annotated with their stage level. The original MetaOCaml added a field val_level to the value_description record describing an identifier in the type checker. This change has lead to the cascade of patches at every place a new identifier is added to the type environment. A new OCaml version typically modifies the type checker quite heavily. Integrating all these modifications into MetaOCaml, accounting for the new field, is a hard job. It is avoidable however: we may associate identifiers with levels differently, by adding a new map to the environment that tells the level of each future-stage identifier. Any identifier not in the domain of that map is deemed present-stage. This alternative helped BER MetaOCaml significantly reduce the amount of changes to the OCaml type checker and make MetaOCaml more maintainable[7].

BER MetaOCaml follows the general staging implementation approach by Taha et al.[5]. After type checking, the code with brackets and escapes is post-processed to translate brackets and escapes into expressions that produce code values[8]. These expressions are built from primitive code generators, which produce a representation of code values; in MetaOCaml, it is OCaml's abstract syntax tree, called Parsetree. Other possible code representations (e.g., the intermediate language or the typed AST) are more difficult to compose. The post-processing of the type-checked code by and large implements the rules in [5, Figure 3]. (The translation of binding forms is new and described in §5.) For example, <succ 1> is translated to the pure OCaml expression (slightly abbreviated) build_apply [Pexp_ident "succ"; Pexp_constant (Const_int 1)] which will construct, at run-time, a Pexp_apply node of the Parsetree. Here, Pexp_ident and Pexp_constant are constructors of Parsetree.

With staging annotations eliminated after the translation, the original OCaml back-end (compiling to the intermediate language, optimizing, and generating the target code) can be used as it is. To run the generated code, we follow the pattern in the OCaml top-level, which also needs to compile and execute the (user-entered) code. Having given an overview of BER MetaOCaml, we describe in depth two of its features, in which BER MetaOCaml significantly differs from the original one.

[7] The first version of BER MetaOCaml modified 35 files in the OCaml distributions, which is 23 fewer files compared to the original MetaOCaml. The patch to the distribution was 59KB in size, reduced to 48KB in the current version.

[8] Doing such a translation before type checking is tantalizing because it can be done as a pre-processing step and requires no changes to OCaml. Alas, we will not be able to support let-polymorphism within brackets; also, the value restriction will preclude polymorphic code values like ⟨[]⟩.

4 Staging user-defined data types

We now illustrate the first of the two distinct features of BER MetaOCaml: the different handling of values of user-defined data types within brackets.

Algebraic data types and records are one of the salient features of OCaml, which, alas, have not been considered in staged calculi. The theory therefore gives no guidance on staging the code with constructors of user defined data types, such as the following:

```
type foo = Foo | Bar of int
 ⟨function Bar _ → Foo⟩
```

The generated program, which can be stored in a file, is **function** Bar _ →Foo. Compiling this file will fail since Foo and Bar are not defined. The problem is how to put a data type declaration into the generated code, which is syntactically an *expression* and hence cannot contain declarations.

The old MetaOCaml dealt with the problem by modifying the AST representing the generated code and adding a field for declarations (actually, the entire type environment) [5, §6.1]. Such a change sent ripples of modifications throughout the type checker, and was one of the main reasons for the divergence from OCaml, which contributed to MetaOCaml's demise.

We observe that there is no problem compiling the code such as **true**, raise Not_found, Some [1] and {Complex.re = 1.0; im = 2.0} – even though labels like re and data constructors like Some are likewise undefined within the compilation unit. However, the data types bool, option, list, Complex.t are either Pervasive or defined in the (separately compiled) standard library. External type declarations like those of Complex.t are found in the compiled interface complex.cmi, which can be looked up when the generated code is compiled. This observation leads to the *constructor restriction*: "all data constructors and record labels used within brackets must come from the types that are declared in separately compiled modules". The code at the beginning of the section is rejected by BER MetaOCaml. The type declaration foo must be moved into an interface file, separately compiled, and be available somewhere within the OCaml library search path – as if it were the standard library type.

Thanks to the constructor restriction, BER MetaOCaml evades the thorny problem of user-defined data types and eliminates the AST modifications by the original MetaOCaml, bringing BER MetaOCaml much closer to OCaml and making it significantly more maintainable.

We are researching the possibility to cleanly lift the constructor restriction. On the other hand, from the experience with BER MetaOCaml (for example, project [8] and the MetaOCaml tutorial at CUFP 2013) the restriction does not seem to be bothersome or hard to satisfy.

5 Detecting scope extrusion

MetaOCaml lets us manipulate open code. This section describes the complexities and trade-offs in making sure all free variables in such code will eventually be bound, by their intended binders. BER MetaOCaml reverses the choice of its predecessors and trades an incomplete type-level check for a comprehensive and more informative dynamic scope-extrusion check. A well-typed BER MetaOCaml program may attempt, when executed, to run an open code or construct ill-scoped code – the code with a free variable that 'escaped' its binder and hence will remain unbound or, worse, bound accidentally. BER MetaOCaml detects such attempts early, aborting the execution of the generator with an informative error message. If the code is successfully generated, it is guaranteed to be well-typed and well-scoped – no matter what effects have been used in its generation.

5.1 Scope-extrusion check in action

Manipulating open code is overshadowed by two dangers. First, the operation to run the code may be applied to the code still under construction:

```
⟨fun x y → ∼(let z = !. ⟨x+ 1⟩ in ⟨z⟩ ))⟩                    (1)
```

The old MetaOCaml rejects this code with the type error:[9]

```
  ⟨fun x y → ∼(let z = .! ⟨x+ 1⟩ in ⟨z⟩ ))⟩
                          ^^^^^^^
.! error : α not generalizable in (α, int ) code
```

BER MetaOCaml type checks this generator but its evaluation aborts with the run-time exception:

```
Exception: Failure
 "The code built at Characters 29−32:
  ⟨fun x y → ∼(let z = !. ⟨x+ 1⟩ in ⟨z⟩ ))⟩
                           ^^^
 is not closed: identifier x_1 bound at Characters 6−7:
  ⟨fun x y → ∼(let z = !. ⟨x+ 1⟩ in ⟨z⟩ ))⟩
       ^
is free".
```

The error message is more informative, explicitly telling the name of the free variable and pointing out, in the generator source code, the binder that should have bound it.

The second, far more common danger comes from effects: a piece of code with a free variable may be stored within the scope of its future-stage binder, to be retrieved from outside:

```
let r = ref ⟨0⟩ in
let _ = ⟨fun x → ∼(r := ⟨x+ 1⟩; ⟨x⟩ )⟩ in                    (2)
⟨fun y → ∼(! r)⟩ ;;
```

(A free variable can also be smuggled out of its binder by raising an exception containing open code, or through control effects, shown in §2.1). The old MetaOCaml accepts this generator and lets it run to completion, producing

[9] In the old MetaOCaml, the operation to run the code was a special form spelled `.!`. In BER MetaOCaml, it is the regular function and spelled `!.`, following the OCaml lexical convention for prefix operators.

⟨**fun** y_2 →(x_1 + 1)⟩, which can be further spliced-in. It is only when we attempt to execute the final code we get a run-time exception Unbound value x_1. If we save the code in a file for offline compilation, the error will be discovered only when we later compile this file.

Although BER MetaOCaml also accepts generator (2), it does *not* let it run to completion. The generator now produces nothing: it aborts with the informative exception:

```
Exception: Failure
 "Scope extrusion detected at Characters 96−111:
          ⟨fun y → ∼(! r)⟩ ;;
          ^^^^^^^^^^^^^^^
 for code built at Characters 67−70:
          let _ = ⟨fun x → ∼(r := ⟨x+ 1⟩; ⟨x⟩ )⟩ in
                                  ^^^
 for the identifier  x_5 bound at Characters 52−53:
          let _ = ⟨fun x → ∼(r := ⟨x+ 1⟩; ⟨x⟩ )⟩ in
                       ^".
```

5.2 The trade-offs of environment classifiers

Previous versions of MetaOCaml employed so-called environment classifiers [11] to prevent the first, quite rare, danger at compile-time: example (1) was rejected by the type checker. Environment classifiers do not help in detecting scope extrusion errors. BER MetaOCaml retired environment classifiers and introduced the dynamic scope extrusion check. The problem with example (1) now reported when running the generator rather than when type-checking it. The problem with example (2) is now reported, early and informatively.

Removing the type-level feature and introducing the dynamic check is the significant departure of BER MetaOCaml from the original system. We summarize this static-dynamic trade-off as follows.

accepting more good programs BER MetaOCaml accepts programs that did not type check previously, for example, ⟨**fun** x →!. x⟩. With environment classifiers, type-checking this code requires impredicative polymorphism. More practically relevant, the operation to run the code was a special form in the old MetaOCaml, with its special typing rules (akin to runST in Haskell). It was not first-class. In BER MetaOCaml, (!.) is the ordinary function. Removing environment classifiers simplified the type system. Previously, a code value ⟨1⟩ had the type (α,int) code where α is the classifier. When defining a new data type, we had to parameterize it by the classifier if the data type may contain code values. This extra type parameter caused not only cosmetic problems: it notably hindered the use of module system to structure generators. For example, LAintcode in §2.1 could not be an implementation of the signature LINALG. To accommodate code types and their classifiers, all abstract types in LINALG should have an extra type parameter, even though LINALG may have implementations without code values. When writing signatures we had to anticipate their implementations.

accepting more bad programs The old MetaOCaml rejected example (1) before running the generator. BER MetaOCaml detects the problem only at run-time. Extensive experience with MetaOCaml showed that the problematic code like (1) is exceedingly rare. Because of the special typing rules of .!, this operator was essentially usable only at the top level; one rarely sees it in subexpressions. Furthermore, by its very design an environment classifier represents just a staging level rather than an individual variable. Therefore, the type checker can tell that the code to run was open but it cannot tell the name of its free variable. Although BER MetaOCaml exception is raised at run-time, the error message refers to the free variable by its name, pointing out its binder.

detecting previously undetected error Environment classifiers do not help in detecting scope extrusion. A well-typed generator could produce ill-scoped code. In contrast, in BER MetaOCaml the generator stops as soon as the ill-scoped piece of code is about to be used in any way, spliced, run, or shown. It throws an exception with a fairly detailed and helpful error message, pointing out the variable that got away and the location of the extrusion, in terms of the source code of the *generator*. Since the error is an exception, the exception stack backtrace further describes exactly which part of the generator caused that variable leak. Previously, we would discover the problem, in the best case, only when compiling the generated code. It could be quite a challenge in figuring out which part of the generator is to blame.

implementation complexity On the whole, environment classifiers are easier to implement. Checking for scope extrusion is not as straightforward as it may seem, as described in the next section. It requires more code, which is however isolated in one MetaOCaml-specific module, mainly, outside the type checker.

run-time cost The scope extrusion check adds run-time overhead to code generation. As the next section mentions, micro-benchmarks and experience showed the overhead to be negligible.

The fact that the old MetaOCaml let a well-typed, effectful generator produce ill-scoped code must not be confused with an implementation bug. It was not a coding error. No practical approaches to statically prevent scope extrusion were known at the time. Even now, there are only hopeful candidates relying on fancy types. Environment classifiers are a remarkable achievement: they are relatively simple to implement, they fit within OCaml type checking and inference, and they statically preclude a class of scoping problems, illustrated by example (1). At the time of writing the original MetaOCaml, it was unclear which of the two scoping problems, example (1) or (2), turns out more common in practice. The decision to retire the classifiers was made in light of all the accumulated experience with MetaOCaml.

5.3 Implementing the scope-extrusion check

Detection of scope extrusion may appear straightforward: traverse the result of a generator looking for unbound identifiers. Instead of traversing, we may annotate each code value with a list of free variables therein; code generation combinators will combine annotations as they combine code values. The code to be run must be annotated with no free variables – reflecting the requirement only closed code be run. That requirement is necessary but not sufficient however. Detecting the scope extrusion by checking the result of the entire generator is too late: it is hard to determine which part of the generator caused the extrusion. Furthermore, scope extrusion does not necessarily lead to an unbound variable: the escaped variable may be accidentally captured by a stray binder. We need early detection of scope extrusion; first we need a precise criterion for it.

The staging theory lets us define scope extrusion and identify it early. The most suitable is not the λU-staged calculus by Taha et al. [5, Fig.1] with brackets and escapes but the 'single-stage target language' of code-generation combinators [5, Fig.2], which we call λ_{AST}. The latter underlies MetaOCaml and is proven to simulate the λU [5, Corollary 1]. The insight comes from translating the characteristic example ⟨**fun** x → ~(body ⟨x⟩)⟩ (where body is a variable bound somewhere in the environment) to code combinators, and evaluating it by the rules of λ_{AST}. The result, to be called efun, is presented in a sugared form compared to [5]:

```
build_fun_simple  "x"  (fun x → body (Var x))
```

where Var (and Lam below) are self-explanatory data constructors of the code representation data type, AST. Our sugared translation is a higher-order abstract syntax (HOAS) representation of the original future-stage function: the future-stage variable and the binder are translated to the present-stage variable and the binder, but of a code type. The big-step evaluation relation of λ_{AST} has the form N;e ⇝v where e is an expression, v is its value and N is a sequence of names ν (which are called symbols in [5] and denoted α). We obtain from [5, Fig.2] that N;efun ⇝Lam(ν,v) provided N,ν; (**fun** x →body (Var x)) ν ⇝v and ν is chosen to be not in N. During the evaluation of body (Var ν), Var ν is the code of the free variable, whose name however appears in the name environment N,ν. The body may store Var ν in a mutable cell. If it is retrieved after efun is evaluated, ν would no longer appear in the name environment current at that point. That is scope extrusion.

The final insight – which leads to the implementation and accommodates delimited control – is that the name environment N is the *dynamic* environment; ν created during the evaluation of build_fun_simple is in the *dynamic scope* of the latter. In other words, build_fun_simple dynamically binds the name of its free variable during the evaluation of its body.

Definition At any point during the evaluation, an occurrence of an open-code value with a free variable whose name is not dynamically bound is called *scope extrusion.* [10]

[10] Normally, dynamic scope cannot be reentered. Therefore, a scope extrusion that occurs at one point in the evaluation will persist through the end. Delimited control however can reenter once exited dynamic scope. Therefore, our definition could potentially raise false alarm. We have not observed such cases in practice.

This definition clarifies our intuitions. It lets us detect scope extrusion without waiting for the result of the generation. It also has a straightforward implementation, without representing N explicitly. The function build_fun_simple (the actual name of the code-generating combinator) creates a fresh variable name and dynamically binds it. Each code value carries the list (heap actually, for ease of merging) of its free variables. Every code-generating combinator verifies that every free variable of the argument code value is currently dynamically bound. A scope-extrusion exception is thrown otherwise. App. B gives further details.

Microbenchmarks (generating code with up to 120 free variables) and experience shows that the scope-extrusion check imposes a linear (in the number of free variables) and negligible cost.

We have described a dynamic, generation-time test, for scope extrusion. A variable that got away is detected as soon as its code is used in any way (spliced, printed, run). The check generates very helpful error messages with precise location information. The location refers to the generator code (rather than the generated code). The test works even in presence of delimited control.

6 Related work

Building code by quasi-quotation is the hallmark of Lisp (see [12] for overview). Any effects are permitted in code generation but the result is not even assured well-formed. Scheme macros support hygiene to some extent (see [13] for overview) but the generator is written in a restricted language of syntax transformers, which permits no effects.

Metaprogramming in Haskell is quite similar to that in Lisp. The original Template Haskell (TH) [14] provides anti- and un-quotation, generates declarations as well as expressions, and permits arbitrary IO effects in the generator. On the flip side, TH is unhygienic. The constructed code may well be ill-typed, and has to be type checked when spliced into the main program (in compile-time code generation) or run, using GHC API. Alas, type errors reported at that stage come with poor diagnostics and refer to the generated code rather than the generator. Furthermore, mistakenly bound variables escape detection. Recently, Haskell gained so-called typed template Haskell expressions TExp. Like MetaOCaml, they construct only expressions (rather than, say, declarations) and are typed checked as being constructed, hence ensuring the generated code is well-typed. TExp offer no run operation; to prevent scope extrusion, any effects during code generation are disallowed.

Code generation is part of partial evaluation (PE); hence a partial evaluator that handles effectful code and performs effects at specialization time has to contend with a possible scope extrusion. Since the user of PE has no direct control over the code generation or specialization, scope extrusion can be prevented by the careful design of PE [15]. Explicit staging annotations let the programmer directly control specialization, and take blame for scope extrusion. BER MetaOCaml places the blame early (before the code generation is finished) and precisely, within the source code of the generator.

Scala-Virtualized [16] successfully demonstrates an alternative to quasi-quotation: code-generating combinators. Normally, using them directly is inconvenient. The pervasive overloading of Scala however makes code generators look like ordinary expressions. For example, $1 + 2$ may mean either the addition of two numbers or building the code for it, depending on the type of that expression. Scala-Virtualized takes the overloading to extreme: everything is an (overloaded) method call, including conditionals, loops, pattern-matching, record declarations, type annotations and other special forms. DSL expressions may look like ordinary Scala code but produce various code representations, which can then be optimized and compiled to target code. Lightweight Modular Staging (LMS) [17] further provides code representations used in the Scala compiler itself. A DSL writer then gets for free the compiler optimizations like common-subexpression elimination, loop fusion, etc. The many DSLs built with LMS proved the approach successful.

Code-generating combinators however cannot easily express polymorphic let (see §3) and often polymorphic code. LMS was not used for DSL with polymorphism. In contrast, polymorphic let is common in the generated OCaml code. With regards to hygiene and scope extrusion, LMS takes the same pragmatic approach as Lisp.

7 Conclusions and further plans

We have presented BER MetaOCaml, a superset of OCaml for writing, conveniently and with ease of mind, programs that generate programs. BER MetaOCaml continues the tradition of the original MetaOCaml by Taha, Calcagno and collaborators, remaining largely compatible with it. There are many design and implementation differences under the hood. They are motivated by the desire to make it easier to maintain and contribute to MetaOCaml, to make it more convenient to use and to catch more errors, and earlier. The motivations are somewhat contradictory, and we had to make choices and test them through experience. We strove to report errors as informatively as possible.

BER MetaOCaml poses questions for the staging theory, of accounting for user-defined data types, objects, modules and GADTs. On the development agenda are adding more ways to 'run' code values, by translating them to C, Fortran, LLVM, Verilog and others. MetaOCaml can then be used for generating libraries of specialized C, etc. code.

Active development, new modular structure, new features of MetaOCaml will hopefully attract more users and contributors, and incite future research into type-safe meta-programming.

Acknowledgments. I am very grateful to Walid Taha for introducing me to MetaOCaml, for his encouragement, and a great number of stimulating conversations. I thank Cristiano Calcagno, Jacques Carette, Jun Inoue, Yukiyoshi Kameyama and Chung-chieh Shan for many helpful discussions and encouragement. Many helpful comments by the anonymous reviewers are gratefully acknowledged.

References

[1] Swadi, K., Taha, W., Kiselyov, O., Pašalić, E.: A monadic approach for avoiding code duplication when staging memoized functions. In: PEPM. (2006) 160–169

[2] Carette, J., Kiselyov, O.: Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. Science of Computer Programming **76** (2011) 349–375

[3] Kiselyov, O., Taha, W.: Relating FFTW and split-radix. In: ICESS. Number 3605 in LNCS (2005) 488–493

[4] Lengauer, C., Taha, W., eds.: MetaOCaml Workshop 2004. In Lengauer, C., Taha, W., eds.: Special Issue on the 1st MetaOCaml Workshop (2004). Volume 62(1) of Science of Computer Programming. (2006)

[5] Calcagno, C., Taha, W., Huang, L., Leroy, X.: Implementing multi-stage languages using ASTs, gensym, and reflection. In: GPCE. Number 2830 in LNCS (2003) 57–76

[6] Kiselyov, O.: BER MetaOCaml N101. `http://okmij.org/ftp/ML/MetaOCaml.html` (2013)

[7] Kiselyov, O., Swadi, K.N., Taha, W.: A methodology for generating verified combinatorial circuits. In: EMSOFT. (2004) 249–258

[8] Kiselyov, O.: Modular, convenient, assured domain-specific optimizations: Can generative programming deliver? Poster at APLAS, `http://okmij.org/ftp/meta-programming/Shonan1.html` (2012)

[9] Kameyama, Y., Kiselyov, O., Shan, C.c.: Shifting the stage: Staging with delimited control. Journal of Functional Programming **21** (2011) 617–662

[10] Calcagno, C., Moggi, E., Taha, W.: ML-like inference for classifiers. In: ESOP. Number 2986 in LNCS (2004) 79–93

[11] Taha, W., Nielsen, M.F.: Environment classifiers. In: POPL. (2003) 26–37

[12] Bawden, A.: Quasiquotation in Lisp. In: PEPM. Number NS-99-1 in Note, BRICS (1999) 4–12

[13] Herman, D.: A Theory of Typed Hygienic Macros. PhD thesis, Northeastern University, Boston, MA (2010)

[14] Sheard, T., Peyton Jones, S.L.: Template meta-programming for Haskell. In Chakravarty, M.M.T., ed.: Haskell Workshop. (2002) 1–16

[15] Thiemann, P., Dussart, D.: Partial evaluation for higher-order languages with state. `http://www.informatik.uni-freiburg.de/~thiemann/papers/mlpe.ps.gz` (1999)

[16] Rompf, T., Amin, N., Moors, A., Haller, P., Odersky, M.: Scala-Virtualized: linguistic reuse for deep embeddings. Higher-Order and Symbolic Computation (2013)

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

A BER MetaOCaml and the old MetaOCaml

This section details the differences of BER MetaOCaml (the current version N101) from the original MetaOCaml (version 3.09.1 alpha 030).

The constructor restriction (§4), the scope extrusion check and the retirement of environment classifiers (§5) have been covered already. The removal of environment classifiers makes the operation to run code an ordinary function with the ordinary type, named run, alias (!.). It is defined in the module Runcode, outside the MetaOCaml kernel. The old syntax .! is obsoleted. The main MetaOCaml kernel module, trx.ml, has been completely re-written, with the scope extrusion check and new algorithms for other operations. For example, translating the Typedtree, representing the type-checked code, to replace brackets and escapes with code combinators now maintains sharing as much as possible. If the Typedtree has no brackets, it is returned as it was. Previously, it was copied. The implementation of cross-stage persistence has also changed, improving the printing of CSP values.

BER MetaOCaml added the test for the well-formedness of recursive let: ⟨**let rec** f = f **in** f⟩ and ⟨**let rec** [] = [] **in** []⟩ are now prohibited. They were allowed in all previous versions of MetaOCaml; therefore, a well-typed generator could produce a well-typed code which nevertheless fails to compile.

BER MetaOCaml builds code values faster, especially for (non-binding) functions. The speed of the generation has not been a problem though. BER MetaOCaml supports applications with labeled arguments and records with polymorphic fields.

The separation into the 'kernel' and the 'user-level' has been described in §3. BER MetaOCaml has introduced the 'system interface', the API for running code, with the special type closed_code and the operations

val close_code : α code $\to$ α closed_code
val open_code : α closed_code $\to$ α code

The latter is total, the former does a scope extrusion check. There may be many ways to 'run' closed_code. Currently, MetaOCaml provides

val run_bytecode : α closed_code $\to$ α

to run the closed code by byte-code compiling it and then executing. More such functions are possible. The function Runcode.run : αcode $\to$ α and its alias, the prefix operation (!.), are the composition of close_code and run_bytecode. BER MetaOCaml is built as a custom top-level, using the standard tool ocamlmktop.

BER MetaOCaml source code is now extensively commented. It comes with a comprehensive regression test suite along with a number of moderate-size tests and benchmarks.

BER MetaOCaml is not only source-compatible with OCaml 4.01 – it is also binary compatible. Any 4.01-built OCaml library and plugin can be used with BER MetaOCaml in their binary form. The building of BER MetaOCaml no longer involves bootstrapping and is hence much faster.

B Implementation of the scope-extrusion check in detail

We describe the implementation of the scope-extrusion check in detail, showing the actual code.

As we saw in §5, the key insight underlying the scope-extrusion check is the dynamic binding of the name of a future-stage variable. A free variable has ‘escaped’ if its name is no longer dynamically bound. To accommodate delimited control, we represent the name of a future-stage variable as a pair of a string with a unique suffix annotated with the source code location information, and a so-called stackmark. A stackmark API is truly minimalist: with_stack_mark (**fun** mark →body) creates a unique stackmark and dynamically binds it during the evaluation of body. There is only one operation on stackmark, to check if it is valid, that is, still dynamically bound. Therefore, a stackmark is realized as a thunk unit →bool. If delimited control is not used, the stackmark API is implemented using a reference cell (containing bool if the stackmark is valid) – so-called ‘shallow binding’.

```
let with_stack_mark_simple = fun body →
    let mark = ref true in
    try
      let r = body (fun () → !mark) in
      mark := false;                          (* invalidate the mark *)
      r
    with e → mark := false; raise e
```

Delimited control should install a different implementation, in which a stackmark is a prompt. Open code is represented as the OCaml AST (realizing the future-stage code)

```
type code_repr = Code of string loc heap * Parsetree.expression
```

paired with a list of free variables. For the sake of merge efficiency, we use a heap rather than a list.

The function build_fun_simple and other binding-form generators use with_stack_mark_simple to enter a new stackmark region, the binding region, for the the execution of the generator of their body. When the body is generated, we check that it contains only valid stackmarks. In OCaml, future-stage bindings are introduced by patterns in let, fun, match, ‘try’ and ‘for’ forms. Furthermore, general functions can have complex binding patterns. They are handled similarly.

Every code generator function checks to see that the stackmarks in the incorporated fragments are all valid, that is, correspond to currently alive variables. These code building functions merge the free variable lists (heaps actually) from the incorporated fragments.