MetaOCaml: Ten Years Later
System Description

Oleg Kiselyov[0000−0002−2570−2186]

Tohoku University, Japan
oleg@okmij.org https://okmij.org/ftp/

Abstract. MetaOCaml is a superset of OCaml for convenient code generation with static guarantees: the generated code is well-formed, well-typed and well-scoped, by construction. Not only the completed generated code always compiles; code fragments with a variable escaping its scope are detected already during code generation. MetaOCaml has been employed for compiling domain-specific languages, generic programming, automating tedious specializations in high-performance computing, generating efficient computational kernels and embedded programming. It is used in education, and served as inspiration for several other metaprogramming systems.
Most well-known in MetaOCaml are the types for values representing generated code and the template-based mechanism to produce such values, a.k.a., brackets and escapes. MetaOCaml also features cross-stage persistence, generating ordinary and mutually-recursive definitions, first-class pattern-matching and heterogeneous metaprogramming.
The extant implementation of MetaOCaml, first presented at FLOPS 2014, has been continuously evolving. We describe the current design and implementation, stressing particularly notable additions. Among them is a new, efficient, the easiest to retrofit translation from typed code templates to code combinators. Scope extrusion detection unexpectedly brought let-insertion, and a conclusive solution to the 20-year–old vexing problem of cross-stage persistence.

Keywords: metaprogramming · staging · code generation

1 Introduction

(BER) MetaOCaml [15, 18] is a superset of OCaml to generate assuredly well-formed, well-scoped and well-typed code using code templates, also known as brackets and escapes (see §2 for the extended example). If code is successfully generated, it is certain to compile. Not only all variables in it are bound: they are bound as intended (see [13] for the discussion of unintended binding). The guarantees apply not only to the completed code: ill-formed or ill-typed code fragments are rejected already by the type checker. MetaOCaml permits unrestricted manipulation of open code fragments, including storing them in reference cells or memo tables. However, as soon as it is detected that a free variable in

such fragment cannot possibly be bound by its intended binder, an exception is raised with a detailed error message.

MetaOCaml has been employed for compiling domain-specific languages [25, 32, 20], generic programming [30], automating tedious specializations in high-performance computing [17], modeling of digital signal processing, generating efficient computational kernels [9, 2, 19] and embedded programming. It is spreading into industry.

MetaOCaml is used in metaprogramming courses at the University of Cambridge and Tsukuba University, and in programming language courses at the University of Montreal, McMaster University, etc. MetaOCaml has had an influence on the design of Scala 3 metaprogramming facilities and Eliom [27], among others. An unexpected application is implementing sophisticated type systems such as session types [21]: fancy (linear, dependent, resource, etc.) types are treated as 'run-time tags' but at a code generation stage. Type errors produce stack traces and can be debugged with an ordinary debugger.

MetaOCaml is being considered for merging into the mainline OCaml, in part due to requests from industry. Preliminary steps are already taken.

The first incarnation of MetaOCaml was described at GPCE 2003 [3]. The current, completely re-designed and re-written version was presented at FLOPS ten years ago [15]. It was called BER MetaOCaml, to distinguish from the original version. The original was unavailable even back then, and has faded by now. The 'BER' qualification has lost its significance, too: it is no longer just about brackets and escapes. Therefore, we shall refer to the sole extant version as MetaOCaml.

Since 2014, MetaOCaml has developed significantly: not only has it kept up with OCaml, it also evolved as a metaprogramming system. Notable milestones include native (native-code, as opposite to bytecode) compilation, offshoring, ordinary and mutually recursive let-insertion, first-class patterns, the conclusive solution to cross-stage persistence. Achieving them required solving long-standing theoretical problems [19, 16, 31, 24]. Here we touch upon hereto unpublished features, also requiring theoretical development, focusing on their design and implementation. More detailed history can be found on the MetaOCaml home page.[1]

Concretely, the paper makes the following contributions:

1. New, efficient, the easiest to retrofit into the extant type-checker translation from typed code templates to code combinators: §3;
2. Let-insertion as the evolution of scope extrusion: §4;
3. The conclusive solution to cross-stage persistence, specifically, implementing cross-stage persistence at all types: §5.

We start with the brief introduction to staging and MetaOCaml, and finish with the related work in §6. MetaOCaml is available from Opam,[2] among other sources. The current version is N114.

[1] `https://okmij.org/ftp/ML/MetaOCaml.html#history`
[2] `https://opam.ocaml.org/`

2 Introduction to Staging and MetaOCaml

The standard example to introduce staging – the "Hello World" of metaprogramming – is the specialization of the power function, first described by A.P.Ershov in 1977 [8]. The well-deserved popularity has made the example a cliche, however. This section uses a related example: more realistic and designed to introduce many facilities of MetaOCaml.[3]

Suppose we are writing code for an embedded system with a low-level CPU that has no multiplication instruction (or it is too slow). It is worth then to try to optimize an important particular case: multiplication by a constant, using shifts and addition. For concreteness, let's take the following target OCaml code intended for the device[4]

```
let x = read_int () in let y = read_int () in 5*(x+1)+y
```

where read_int is a stand-in for reading a sensor value. For optimization we shall use OCaml as well, now as a metalanguage. To be exact, we shall use MetaOCaml, which adds to OCaml the facility to represent, or quote code, using code templates, or brackets .⟨...⟩.:

```
let c = .⟨ let x = read_int () in let y = read_int () in 5*(x+1)+y ⟩.
```

Brackets are akin to string quotation marks '"'. In fact, the above code template can be converted to a string and written to a file. Unlike strings, however, code templates have structure: the code within a template must be a well-formed – moreover, well-typed OCaml code. Since the sample enclosed code has the type int, the entire template has the type int code. Code templates like above are values – also called 'code values' – and can be named (bound to variables), passed as arguments and returned from functions. The code within a template is only quoted (and type-checked), but not evaluated. It can be written to a file, compiled and then executed – at a 'future stage', so to speak. In contrast, unquoted MetaOCaml code, which is ordinary OCaml, is executed when the program runs: 'now', at the present stage.

To optimize the constant multiplication in c, we change the template to

```
let copt = .⟨ let x = read_int () in let y = read_int () in .~(mul 5 .⟨(x+1)⟩.)+y ⟩.
```

Here, .~ (called 'escape') marks the hole in the template; the escaped expression mul 5 .⟨(x+1)⟩. is evaluated to generate the code to plug into the hole. A template with a hole is no longer a value then. The function mul defined below is a code generator: it takes the known multiplicand (as integer) and the *code* for the other multiplicand (as a code value) and produces the code for the product. The code value .⟨(x+1)⟩. passed as the second argument to mul is open, with the free variable "x". Passing around, splicing, storing in reference cells, etc., open

[3] The complete code for the example is available at `https://okmij.org/ftp/meta-programming/tutorial/mult.ml`

[4] One may quip that a platform with no support for multiplication unlikely supports OCaml. Later in this section we mention using MetaOCaml for generating C (or Wasm) instead.

code is the the source of MetaOCaml power. In effect, we manipulate (future-stage) variables *symbolically*. Although we can splice variables into larger future-stage expressions, we cannot compare or substitute them, learn their name, or examine the already generated code and take it apart.[5] This pure generativity of MetaOCaml helps maintain hygiene: open code can be manipulated but the lexical scoping is still preserved.[6]

The code generator mul is as follows (the type annotations are optional):

```
let rec mul (n:int) (x:int code) : int code = match n with
  | 0 → .⟨0⟩.
  | 1 → x
  | n when n < 0 → .⟨ − .~(mul (−n) x) ⟩.
  | n when n land 1 = 1 → .⟨ .~x + .~(mul (n−1) x) ⟩.
  | n → let (m,k) = factors_of_two n in .⟨ Int.shift_left .~(mul m x) k ⟩.
```

where factors_of_two n computes the representation of the positive integer n as $m2^k$ with m odd, and returns (m,k) as a pair. Brackets and escapes are also called staging annotations, for a reason: if we erase them from mul, it becomes the ordinary, well-typed OCaml function for correctly, but slowly, multiplying two integers.

Before applying mul to copt, one may want to test it. First, let's see the code mul generates – in a simple context, provided by the so-called eta (which is often used in partial evaluation and called 'the trick' [6]):

```
let eta = fun f → .⟨fun x → .~(f .⟨x⟩.)⟩.
⇝ val eta : (α code → β code) → (α → β) code = <fun>

eta (mul 5)
⇝ − : (int → int) code = .⟨fun x_1 → x_1 + (Int.shift_left x_1 2)⟩.
```

(shown after ⇝ is the response of the MetaOCaml top-level). The expression eta (mul 5) hence generates the code template of a function. Code values can be printed, which is what we see in the top-level response. The bound variable is renamed: important to ensure hygiene [3].

Code values can also be saved as text into a file, to be compiled as ordinary code. Code values can also be 'run': that is, their code can be compiled, linked in and executed within the generator. It is useful for run-time specialization, and also for testing. For example, we can evaluate Runcode.run (eta (mul 5)) 7 and check that it returns 35 as expected.

Returning to the earlier copt, it evaluates to

```
.⟨let x_1 = read_int () in let y_2 = read_int () in
    ((x_1 + 1) + (Int.shift_left (x_1 + 1) 2)) + y_2⟩.
```

[5] At first blush, the inability to examine the generated code seems to preclude any optimizations. Nevertheless, generating optimal code is possible [22, 23, 17, 19].

[6] Unless we store open code in reference cells outside the template that binds the variables. Code generation with effects hence brings in the danger of *scope extrusion*. MetaOCaml takes great pains to detect and report scope extrusion: §4.

Compared to the original c, it uses only shifts and additions and should be faster. There is a problem: the expression x_1 + 1 is duplicated. The problem can be severe for a complicated expression, or even in a call to an imperative function. To avoid duplication, MetaOCaml lets us bind an expression to a variable, using letl : α code $\to$ ((α code $\to$ ω code) $\to$ ω code) (local let-insertion). We can also use a general, floating let-insertion genlet (exp : α code) : α code to bind exp at the highest possible position determined by data dependencies[7] to a fresh variable, obtaining the code value containing the variable:

```
.⟨ let x = read_int () in let y = read_int () in .~(letl .⟨(x+1)⟩. (mul 5))+y ⟩.
⇝ .⟨let x_1 = read_int () in let y_2 = read_int () in
      (let t_3 = x_1 + 1 in t_3 + (Int.shift_left t_3 2)) + y_2⟩.

.⟨ let x = read_int () in let y = read_int () in .~(mul 5 (genlet .⟨(x+1)⟩.))+y ⟩.
⇝ .⟨let x_1 = read_int () in let t_2 = x_1 + 1 in let y_3 = read_int () in
      (t_2 + (Int.shift_left t_2 2)) + y_3⟩.
```

We could have used letl or genlet in the implementation of mul. A better idea is to leave the decision as to what, where and how to let-bind to the user, and merely require the second argument to mul be the code value that is safe to duplicate. MetaOCaml provides a special type α val_code for such code values, which which is a subtype of α code. Values of val_code types are produced from literals (with a particular MetaOCaml annotation) or using genletv.[8] Here is the re-written mul:

```
let rec mul (n:int) (x:int val_code) : int code = match n with
  | 0 → .⟨0⟩.
  | 1 → (x :> int code)
  | n when n < 0 → .⟨ − .~(mul (−n) x) ⟩.
  | n when n land 1 = 1 → .⟨ .~(x :> int code) + .~(mul (n−1) x) ⟩.
  | n → (* as before *)
```

to be invoked like mul 5 (.⟨y⟩. [@metaocaml.value]) or mul 5 (genletv .⟨(x+1)⟩.). The invocation mul 5 .⟨(x+1)⟩. does not type check: .⟨(x+1)⟩. is not of the type α val_code; mul 5 (.⟨(x+1)⟩. [@metaocaml.value]) does not type either since x+1 is not syntactically a value.

One might have wished for the optimization to apply to the original c template as it was, without adding mul explicitly by hand. The explicitness is intentional. One has to keep in mind that staging was developed as a push-back against partial evaluators: a magic box that did everything automatically, and sometimes to an impressive result (which could inexplicably change upon a small, seemingly innocent modification). Programmers had no explicit control, or understanding of what it did. Still, the point that MetaOCaml is too explicit stands. It is indeed better thought of as an 'assembler' of metaprogramming. The end users should generate code not with code templates but with abstractions suitable to their domain – as was demonstrated in [23, 19]. The explicitness of Meta-

[7] There is also a way to specify the desired binding locus [24].
[8] Thus genlet is genletv followed by the upcast to code.

OCaml has an upside: knowing exactly what code will be produced, with no surprises.

The reader may have noted that an embedded system with no or very slow hardware multiplication would unlikely run OCaml code. Generating OCaml code was not a waste however: since the code is simple (as is often the case), it may be converted to a low-level language such as C, using *offshoring* [19]. For our example code, offshoring produces

```
int fn(){
  int const x_26 = read_int();
  int const t_28 = x_26 + 1;
  int const y_27 = read_int();
  return ((t_28 + (t_28 << 2)) + y_27); }
```

One may quip that GCC will automatically convert multiplication by constants to shifts and additions (at least on x86 platform). However, an embedded platform may not be supported by GCC. In fact, our running example is modeled after two student projects of developing a simple DSL for robot control, using MetaOCaml to generate and then offshore the code. The robot platform was rather peculiar, underpowered and not supported by GCC.[9]

3 Implementing MetaOCaml

MetaOCaml is a programming language system, and hence looks like most other (typed) language systems: the compiler with parsing, type-checking, optimization and code-generation passes producing an executable; the standard library; tools. MetaOCaml is deliberately designed to share, or piggy-back, on the parent OCaml language as much as possible. It is intended to be fully source- and binary-compatible with OCaml.[10] Therefore, MetaOCaml code can use (in source or binary) any OCaml standard or third-party library and any tool. Compiled MetaOCaml code can be linked with any other OCaml code. One may use MetaOCaml as a daily driver for ordinary OCaml development, as the author has been doing for over a decade.

MetaOCaml compiler is also engineered to be a *small* set of patches to the OCaml front-end (parser and type-checker). The OCaml back-end (optimizer and code generator) is reused, *exactly as is*. To this end, MetaOCaml deliberately avoids extending any OCaml compiler data structures.

When it comes to syntax, MetaOCaml only adds three new tokens: two brackets and the escape, which require all but a simple change to the OCaml grammar. The brackets and escapes are parsed into so-called extension nodes of the OCaml AST (a.k.a. Parsetree). One may create them using OCaml's own notation. For example, eta in §2 could be entered as

```
let eta (f: α code → β code) : (α → β) code =
```

[9] The robot is based on Daizen's e-Gadget CORE, whose development environment uses MPLAB C compiler for PIC18 MCU by Microchip Technology.

[10] It was not the case for the original MetaOCaml.

```
[%metaocaml.bracket fun x → [%metaocaml.escape f [%metaocaml.bracket x]]]
```

without the bracket-escape syntax. The two notations can be mixed-and-matched. One may design a source-level preprocessor to create the extension nodes from any other syntax for code templates. (So far, no candidates have been proposed however.)

The common approach of implementing quasi-quotation (of which brackets and escapes are an instance), which goes back to Lisp, is translating to code-generating combinators [3, 5]. In Lisp, this translation is a source-level, macro-expansion–like transformation. In MetaOCaml, quasi-quotation is typed, however. It may be surprising that a source-to-source translation for brackets and escapes is possible, in principle [16]. No changes to OCaml would be needed then. On the other hand, type errors will be reported in terms of the translated code, which may be confusing. Any translation to code-generating combinators needs to associate variables with their stage, and hence to maintain a variable environment. Handling data types requires type/constructor information. Therefore, a translation to code-generating combinators has to do some amount of type checking anyway. All in all, it seems a better idea to do the translation at or after type checking.

In the original MetaOCaml, the translation to code-generating combinators was post type-checking. A translation before or after type checking is a separate pass, over the entire code. In the current MetaOCaml, the translation is integrated with type checking, avoiding the overhead of a separate pass and of scanning the code outside brackets. The cost of specifically MetaOCaml processing is hence proportional only to the amount of the bracketed code, which is normally a small portion of code base. Furthermore, the current type-checking–integrated translation is designed to be the least invasive, the easiest to retrofit into an existing type-checker, and hence easily portable. Uncannily, the translation is using what feels like only two stages to support multiple. As the warm-up, §3.1 describes the type-checking of staged programs with brackets and escapes, introducing the notation. §3.2 presents the modification to translate brackets and escapes away.

3.1 Type-checking Staged Programs

The present and the following section present the theory of MetaOCaml implementation. They use the standard in theoretical CS mathematical notation and look theoretical. The notation, however, is the *pseudo-code of the actual implementation*. The efficient translation, §3.2, was first designed in the mathematical notation, to clarify its subtle points. The implementation later transcribed the notation into OCaml code.

We start with the base calculus: it is the utterly standard simply typed lambda calculus with integers, shown merely for the sake of notation, particularly the notation of the typing judgment: $\Gamma \vdash e \Rightarrow e : t$. The notation makes it explicit that type checking is type reconstruction: converting an 'untyped' expression e to the type-annotated form $e : t$ – or, in terms of the OCaml type checker, converting from Parsetree to Typedtree.

Variables	f, x, y, z
Types	$t ::= \mathsf{int} \mid t \to t$
Integer constants	$i ::= 0, 1, \ldots$
Expressions	$e ::= i \mid x \mid e\ e \mid \lambda x.\, e$
Environment	$\Gamma ::= \cdot \mid \Gamma, x{:}t$

Variables	f, x, y, z
Types	$t ::= \mathsf{int} \mid t \to t \mid \texttt{<}t\texttt{>}$
Integer constants	$i ::= 0, 1, \ldots$
Expressions	$e ::= i \mid x \mid e\ e \mid \lambda x.\, e \mid \texttt{<}e\texttt{>} \mid \texttt{~}e$
Stage	$n, m \geq 0$
Environment	$\Gamma ::= \cdot \mid \Gamma, x^n : t$

Fig. 1. Base calculus: simply-typed lambda calculus with integers (left) and the corresponding staged calculus (right)

$$\frac{}{\Gamma \vdash i \Rightarrow i : \mathsf{int}} \qquad \frac{x : t \in \Gamma}{\Gamma \vdash x \Rightarrow x : t} \qquad \frac{\Gamma \vdash e \Rightarrow e : t' \to t \quad \Gamma \vdash e' \Rightarrow e' : t'}{\Gamma \vdash e\ e' \Rightarrow (e : (t' \to t)\ e' : t') : t}$$

$$\frac{\Gamma, x : t' \vdash e \Rightarrow e : t}{\Gamma \vdash \lambda x.\, e \Rightarrow (\lambda x : t'.\, e : t) : (t' \to t)}$$

We assume that the initial environment Γ_{init} to type check the whole program contains the bindings of the standard library functions such as succ, addition, etc. In the rule for abstraction, one may wonder where does the type t' come from. For the purpose of the present paper, one may consider it a 'guess'. After all, our subject is not type inference, but staging – to which we now turn.

Figure 1 (right) presents the staged calculus: the Base calculus extended with bracket <*e*> and escape ~*e* expression forms and code types <*t*>.[11] The calculus (as MetaOCaml) is actually *multi-staged*: brackets may nest arbitrarily, e.g., <<1>>. The level of nesting is called *stage*. The present stage, stage 0, is outside of any brackets. An expression at stage 1 or higher is called future-stage. The typing judgment $\Gamma \vdash_n e \Rightarrow e : t$ is now annotated with stage $n \geq 0$. All variable bindings in Γ are also annotated with their stage: $x^n : t$.

The rules for integer constants and application remain the same, modulo replacing $\vdash$ with $\vdash_n$: in general, most typing rules are unaffected by (or, are invariant of) staging. This is a good news for implementation: adding staging to an extant language does not affect the type checker to large extent. Here are the changed and new rules:

$$\frac{x^m : t \in \Gamma}{\Gamma \vdash_n x \Rightarrow x^m : t}\ m \leq n \qquad \frac{\Gamma, x^n : t' \vdash_n e \Rightarrow e : t}{\Gamma \vdash_n \lambda x.\, e \Rightarrow (\lambda x^n : t'.\, e : t) : (t' \to t)}$$

$$\frac{\Gamma \vdash_{n+1} e \Rightarrow e : t}{\Gamma \vdash_n \texttt{<}e\texttt{>} \Rightarrow \texttt{<}e : t\texttt{>} : \texttt{<}t\texttt{>}} \qquad \frac{\Gamma \vdash_n e \Rightarrow e : \texttt{<}t\texttt{>}}{\Gamma \vdash_{n+1} \texttt{~}e \Rightarrow \texttt{~}(e : \texttt{<}t\texttt{>}) : t}$$

The type-checker also annotates variable references with the stage, in addition to the type. A variable bound at stage n may be used at the same stage – or higher (but not lower!). A present-stage variable may appear within brackets: so-called *cross-stage persistence* (or, CSP). As one may expect, bracket increments

[11] The code type in the current MetaOCaml is not pre-defined, but is a library type like Stdlib.Complex.t. Since the set of pre-defined types and values remains the same as in OCaml, binary compatibility is maintained.

$\mathsf{lift}_t \quad : t \to \langle t \rangle$ $\qquad$ $\mathsf{mkl} \quad : (\langle t_2 \rangle \to \langle t_1 \rangle) \to \langle t_2 \to t_1 \rangle$

$\mathsf{mkid}_t : \mathsf{string} \to \langle t \rangle$ $\qquad$ $\mathsf{mkbr} : \langle t \rangle \to \langle\langle t \rangle\rangle$

$\mathsf{mka} \quad : \langle t_2 \to t_1 \rangle \to \langle t_2 \rangle \to \langle t_1 \rangle$ $\qquad$ $\mathsf{mkes} : \langle\langle t \rangle\rangle \to \langle t \rangle$

Fig. 2. Code-generating combinators, see §4 for more discussion and possible implementation. Here lift_t is the family indexed by type t, to be discussed in §5.

the stage for its containing expression and escape decrements. Furthermore, escapes must appear within a bracket.

For example, $\langle\langle \tilde{}(\langle 1 \rangle) \rangle\rangle$ has the type $\langle\langle \mathsf{int} \rangle\rangle$, the expression $\langle\langle \lambda x.\, \tilde{}(f\, x) \rangle\rangle$ is ill-typed but $\langle\langle \lambda x.\, \tilde{}(f\, \langle x \rangle) \rangle\rangle$ is well-typed in an environment where f is bound to a function $\langle \mathsf{int} \rangle \to \langle \mathsf{int} \rangle$ at stage 0.

After a program is type-checked and converted to the type-annotated form (a.k.a., Typedtree), we have to compile it. The type-annotated form contains brackets and escapes, so our compilation has to account for them. One popular approach [3, 5] is to post-process the type-annotated expression to eliminate all brackets and escapes. The post-processed Typedtree then has the same form as in the ordinary OCaml; therefore, we can use the OCaml back-end (optimizer and code generator) as it is – which is what MetaOCaml does.

Formally, the result of post-processing is the Base calculus enriched with code types (as well as string types and literals) and whose initial environment contains the functions in Fig. 2. These code-generating combinators are the producers of values of the code type. We call this calculus Base_1.

3.2 Optimized Translation of Brackets and Escapes

We now present the optimized translation of Staged expressions that converts brackets and escapes into invocations of code-generating combinators.

Figure 3 presents the translation $\lfloor e : t \rfloor$ of the interior of outer brackets in Staged to the code-generating combinators.[12] As mentioned earlier, we do not have to scan the whole staging program, but only the part within brackets. The interior translation exploits the fact that, surprisingly, the translation does not depend on the exact future stage number. The case for x^0 is discussed in §5.

$$
\begin{aligned}
\lfloor i : \mathsf{int} \rfloor &= \mathsf{lift}_{\mathsf{int}}\, i : \langle \mathsf{int} \rangle \\
\lfloor x^{m+1} : t \rfloor &= x : \langle t \rangle \\
\lfloor x^0 : t \rfloor &= \begin{cases} \mathsf{mkid}_t\ \text{"x"} : \langle t \rangle & \text{if } x \in \Gamma_{init} \\ \mathsf{lift}_t\ x : \langle t \rangle & \text{otherwise} \end{cases} \\
\lfloor (e\, e') : t \rfloor &= \mathsf{mka}\, \lfloor e \rfloor\, \lfloor e' \rfloor : \langle t \rangle \\
\lfloor \lambda x^{n+1} : t'.\, e : t \rfloor &= \mathsf{mkl}\, (\lambda x : \langle t' \rangle.\, \lfloor e : t \rfloor) : \langle t' \to t \rangle \\
\lfloor \tilde{}(e : \langle t \rangle) \rfloor &= e : \langle t \rangle
\end{aligned}
$$

Fig. 3. Translation of the interior of outer brackets into Base_2.

The typing judgment is now $\Gamma \vdash_n e \Rightarrow e' : t$ where e is an (un-annotated) expression of the Staged calculus and e' is the type-annotated expression of Base_1 extended with $\tilde{}e$ and stage-annotated variables. (Bindings in Γ are also stage-annotated. For present stage, the annotation may be dropped.) Such extended

[12] performed by `trx_translate` of `typing/trx.ml`

calculus is called Base_2. Quite unexpectedly, Base_2 has no need for brackets; it only needs escapes, hence the changes to the OCaml Typedtree are minimal. In fact, there are no changes at all, thanks to Typedtree attributes: an escape is indicated by a dedicated attribute attached to a Typedtree node.

$$\frac{}{\Gamma \vdash_n i \Rightarrow i : \mathsf{int}} \qquad \frac{x^m : t \in \Gamma}{\Gamma \vdash_n x \Rightarrow x^m : t}\, m \leq n$$

$$\frac{\Gamma \vdash_n e \Rightarrow e : t' \to t \quad \Gamma \vdash_n e' \Rightarrow e' : t'}{\Gamma \vdash_n e\, e' \Rightarrow (e : (t' \to t)\ e' : t') : t} \qquad \frac{\Gamma, x^n : t' \vdash_n e \Rightarrow e : t}{\Gamma \vdash_n \lambda x.\, e \Rightarrow (\lambda x^n : t'.\, e : t) : (t' \to t)}$$

$$\frac{\Gamma \vdash_1 e \Rightarrow e : t}{\Gamma \vdash_0 \texttt{<}e\texttt{>} \Rightarrow \lfloor e : t \rfloor : \texttt{<}t\texttt{>}} \qquad \frac{\Gamma \vdash_{n+2} e \Rightarrow e : t}{\Gamma \vdash_{n+1} \texttt{<}e\texttt{>} \Rightarrow \tilde{}(\mathsf{mkbr}\ \lfloor e : t \rfloor) : \texttt{<}t\texttt{>}}$$

$$\frac{\Gamma \vdash_0 e \Rightarrow e : \texttt{<}t\texttt{>}}{\Gamma \vdash_1 \tilde{}e \Rightarrow \tilde{}(e : \texttt{<}t\texttt{>}) : t} \qquad \frac{\Gamma \vdash_{n+1} e \Rightarrow e : \texttt{<}t\texttt{>}}{\Gamma \vdash_{n+2} \tilde{}e \Rightarrow \tilde{}(\mathsf{mkes}\ \lfloor e : \texttt{<}t\texttt{>} \rfloor) : t}$$

Fig. 4. Type-checking and translation of Staged into Base_2.

Figure 4 presents the pseudo-code of the optimized translation integrated with type reconstruction. The figure makes it clear how the Base type reconstruction – that is, the Typedtree construction in the ordinary OCaml – has to be modified for staging. Most of the rules (see constant and application rules) are unmodified. We still need to maintain the stage (as a global mutable variable in the current implementation). The rule for lambda (and other binding forms) has to annotate the bound variable with its stage as it is put into the environment. We do it by adding an attribute bearing the stage to the value_description of the variable. The variable rule has to check that the stage of the variable is less than or equal the current stage, and to put the stage-annotated variable into Typedtree. In the implementation, nothing needs to be done for the latter: The Texp_ident node of the Typedtree carries the value_description taken from the environment, which already has the stage attribute. The only significant changes are the rules for brackets and escapes (represented in Parsetree as extension nodes).

The selective translation $\lfloor - \rfloor$ is indeed done only on the parts of the overall Typedtree that represent future-stage sub-expressions. Therefore, when compiling plain OCaml programs, MetaOCaml imposes *no* overhead: MetaOCaml-specific processing is not even activated.

Proposition If $\Gamma \vdash_n e \Rightarrow e : t$ in the Staged calculus then $\Gamma \vdash_n e \Rightarrow e' : t$ in the optimized translation.

Proposition If $\Gamma \vdash_n e \Rightarrow e' : t$, then e' has no nested escapes.

Corollary If $\Gamma_{init} \vdash_0 e \Rightarrow e' : t$ then e' is strictly a Base_1 expression: it contains no escape nodes or stage-annotated bindings. The type reconstruction hence gives the ordinary OCaml Typedtree, which can then be processed by the OCaml back-end as is.

Theorem If $\Gamma \vdash_0 e \Rightarrow e' : t$ then $\Gamma \vdash \bar{e'} \Rightarrow e' : t$ in Base_1 where $\bar{e'}$ is e' with all type annotations removed.

Evaluation The integrated translation sounds almost too good to be true. The goal of the formalization, and of the main theorem, is to convince that the translation is correct. It is implemented in the current MetaOCaml (version N114) – by literally transcribing the pseudo-code of Fig. 4 into OCaml – resulting in simpler and shorter code. It worked on the first try, passing all tests in the extensive MetaOCaml testing suite. No issues have been reported.

Since the very beginning BER MetaOCaml took pains to make the Typedtree after the translation look exactly as in the plain OCaml. Time has showed that it was wise. We remind that MetaOCaml is designed to use the OCaml back-end as is. The OCaml back-end has been constantly enhanced with new optimizations and facilities (FLambda, Multi-core, to name the biggest). MetaOCaml comes to benefit from these optimizations automatically.

4 Let-insertion

As we have seen in §2, let-insertion (particularly, genlet) is useful for effecting sharing and avoiding code duplication. The importance of let-insertion has been recognized early on in partial evaluation [11, §5.5.4]. It is commonly accomplished via continuation-passing (monadic) style [1, 4, 28] or, more conveniently, via delimited control [26, 12]. In fact, the primary motivation for the OCaml delimited control library delimcc [14] was implementing genlet. Surprisingly, genlet turns out realizable in MetaOCaml much simpler, without any delimited control, piggy-backing on what MetaOCaml has had for a decade: detecting scope extrusion.

Detecting scope extrusion (that is, open code whose free variables shall remain unbound) was introduced in version N103 and described in [15]. Here is a brief reminder, using the formalization from the previous section. Fig. 2 introduced code-generating combinators, but did not say what they generate. Indeed, how is the type <*t*> realized, what exactly is the code value? As §2 has hinted, a code value (code template) is essentially a string containing code text. An easier to generate representation is an algebraic data type [3]. An algebraic data type representing code is nothing but the abstract syntax tree (AST), called Parsetree in the OCaml compiler. From the very beginning and up until N103, code values in MetaOCaml were Parsetree.expression values.

In the formalism of §3.1, the AST corresponding to Base can be described by the following OCaml data type[13]

```
type vname = string
type ast = Int of int | Var of vname | App of ast * ast | Lam of vname * ast
```

The code-generating combinators are then

```
type α code = ast
let mkid (n:vname) : α code = Var n
let mka (e1: (α → β) code) (e2: α code) : β code = App (e1,e2)
```

[13] For simplicity, we hereafter restrict ourselves to two stages, as most common. Therefore, the generated code contains no staging annotations.

```
let mkl (f: α code → β code) : (α→β) code = let v = gensym () in Lam (v, f (Var v))
```

The function mkl (whose real MetaOCaml name is build_fun_simple) chooses a fresh bound variable name, as explained in [3].

Such simple implementation does not suffice for detecting scope extrusion: we need to keep track of free variables. To this end, version N103 introduced an annotated AST:

```
type annot
type α code = annot * ast
let mkid n = (empty, Var n)
let mka (a1,e1) (a2,e2) = (merge a1 a2, App (e1,e2))
```

where annot is a monoid with the unit empty and the operation merge. Specifically, annot is a set of variable names that are free in the code value. The function mkl, which introduces a new variable name, also dynamically binds it for the dynamic extent of generating its body. It uses a dynamic binding facility

```
val dlet : vname  → (unit → α) → α
val dbound : vname → bool
```

to dynamically bind a given vname for the duration of executing a thunk, and check if vname is dynamically bound. The scope extrusion check is hence the check that each free variable occurs only within the dynamic extent of mkl that introduced it. Concretely,

```
type annot = VSet.t   (* set of variable names *)
let empty = VSet.empty
let merge a1 a2 = if VSet.all dbound a1 && VSet.all dbound a2 then VSet.union a1 a2
                  else error "Scope_extrusion"
let mkl f = let v = gensym () in
            let (a,c) = dlet v (fun () → f (VSet.singleton v, Var v)) in
            (VSet.remove v a, Lam (v, c))
```

Every code-generating combinator performs the scope extrusion check on its arguments (for mka above, the check is integrated into the merge operation). In reality, MetaOCaml uses a priority heap rather than set; it also takes great pains to generate a detailed error message upon scope extrusion. See [15] for more detail, and also the discussion on lexical scoping in the generated code corresponding to dynamic scoping in the generator.

Turning to let-insertion, recall from §2 that genlet (exp : α code) : α code binds exp 'somewhere above' to a fresh variable and returns the code value containing the variable. One may say that genlet exp creates a *promise* of a let-binding of a fresh v to exp – so-called 'virtual let-binding' – and returns Var v. This is the key idea of the implementation. The virtual let-binding is carried as yet another annotation to the code value. Concretely, annot is extended to

```
type vbindings = (vname * (annot * ast)) list
and  annot = VSet.t * vbindings
```

The merge function is extended to merge vbindings – checking, as before, for scope extrusion and merging free variable sets. The code generating combinator mka (and many more like it in the actual MetaOCaml) remain unchanged. The

let-insertion introduces the virtual binding, which is then propagated (floats) as composite expressions form and their annotations are merged:

```
let genlet exp = let v = gensym () in ((VSet.empty,[(v,exp)]),Var v)
```

The combinator mkl now has to check if any exp_i in list of virtual bindings $(\mathsf{v}_i,\mathsf{exp}_i)$ contains the variable that is bound by that mkl. If so, the corresponding virtual binding has to be converted to the real let-binding; it cannot be allowed to float further as scope extrusion occurs otherwise. We see now where exactly the let-binding corresponding to genlet exp will be inserted: right under the closest binder that binds a variable that is free in exp (or at the top level, if exp is closed).

There is a bit more than meets the eye: for example, the exp in genlet exp may itself use genlet, which induces a dependency and an order on let-bindings.

Evaluation The implementation of let-insertion followed the just presented outline and was relatively short, changing hardly any code, because the code annotation infra-structure was already in place and could be reused. The extension to mutually-recursive let-insertion [24] proved to be just as straightforward. Also straightforward is the explicit control of the insertion locus [24].

Let-insertion proved to be a valuable addition to MetaOCaml, appearing quite often in code bases: see, e.g., the scalar promotion optimization in [19].

5 Cross-stage persistence

Looking back to the fragment of mul code from §2:

```
let (m,k) = factors_of_two n in .⟨ Int.shift_left .~(mul m x) k ⟩.
```

we see k appearing inside the bracket but bound outside. The bracket contains one more free variable: Int.shift_left, bound in the OCaml standard library. Program variables appearing in templates are called "cross-stage persistence" (CSP).[14] Cross-stage persistence is ubiquitous: for one, all references to standard library are CSPs.

The example highlights the two varieties of CSP: global, (standard) library identifiers; and locally-bound identifiers. The difference is visible in the translation rules for $\lfloor x^0 : t \rfloor$ in Fig. 3. For our example, the translation to code-generating combinators gives:

```
let (m,k) = factors_of_two n in mka (mka (mkid "Int.shift_left") (mul m x)) (lift_int k)
```

The generated code will hence include the identifier Int.shift_left, which, when the code is compiled, will be taken to refer to the standard library function – the same function it refers to in the generator. Globally-bound CSPs are hence references to the libraries available at the present stage and assumed available at a future stage: the 'common knowledge' so to speak.[15]

[14] We will take 'CSP' to also abbreviate 'cross-stage–persistent variable'.

[15] A pun on the modal logic of code types [7].

Locally-bound CSPs, in contrast, are by their very nature valid only within their local scope and are not accessible from other code. Their values have to be incorporated into the generated code: somehow represented in AST and eventually converted to text. This lifting, or serialization, is performed by the family of functions lift_t of Fig. 2. Serialization clearly cannot be done the same way for all types. For integers, lift_{int} n is just Int n. Booleans, strings, and other easily serializable values are similar. Users may also define their own lifting functions [17, §3.2.1].

On the other hand, when t is a function, reference, input channel, etc., type, lift_t is a puzzle. It is not clear if such a lifting is possible – or even makes sense. Deepens the puzzle is the polymorphic

```
let polylift : α → α code = fun x → .⟨x⟩.
```

which has been definable in MetaOCaml since the very beginning. This definition looks like a type-uniform serialization, which is impossible.

The long-standing, vexing puzzle has been finally solved in the latest version of MetaOCaml – using let-insertion. That is, lift_t v for any non-serializable t is implemented essentially as genlet: choosing a fresh identifier cps_i and returning Var "cps_i" annotated with a special virtual let-binding of cps_i to v. Unlike the ordinary virtual let-binding, its rhs is not necessarily of code type. It is certain however to contain no free future-stage variable; therefore, it always floats to the very top. For example,

```
.⟨ .~(let f = fun x → x in .⟨f⟩.) 1⟩.
```

when translated and evaluated becomes the code value that is the AST

```
App (Var "csp_1", Int 1)
```

annotated with the special virtual let-binding of csp_1 to fun x → x. Such code value may be thought of as a 'staged-separated' let-binding

```
let csp_1 = fun x → x in .⟨ csp_1 1 ⟩.
```

whose binding is in the present-stage but the body is in the future. More constructively, it may be thought of as a pair of the present-stage value fun x → x and the code value .⟨fun csp_1 → csp_1 1⟩.. When such a code value is Runcode.run, the code fun csp_1 → csp_1 1 is compiled and then applied to the first component of the pair (the identity function in our case). One may say that the binding and the body of a stage-separated let-expression are re-united. When a CSP code value is saved into a file, it is saved as a function taking the CSP value as the argument. When eventually invoked, the programmer will have to somehow arrange for the appropriate value (e.g., recomputed, etc.) All in all, in the presence of arbitrary CSP, a code value is a 'staged' closure, over CSPs.

Evaluation CSPs are ubiquitous: although hardly ever mentioned in staging literature, no practical metaprogramming system may afford to ignore them. The experience has shown that locally-bound CSPs at function types are surprisingly common, in cases of run-code specialization. In earlier versions of MetaOCaml, such CSPs have been supported via a horrible hack, which only worked for byte-

code and resulted in printing of non-compilable code. At long last, the problem has been solved.

CSPs are so common that they are used in the MetaOCaml implementation itself. For a simple code template, rather than translating it to the code that builds AST at the generator run-time, we may build the AST at the compilation time. AST is serializable, and lifted as CSP from the compile-time to the generator run-time. The generator then accesses it as a literal constant.

The implementation of MetaOCaml hence uses itself. The code `typing/trx.ml` that contains the type-checking, translation and code-generation combinators is used both at type-checking and code-generation time. In particular, the CSP implementation is used at both times. One gets the feeling of a self-specializer, familiar from Futamura projections.

6 Related work

Due to the lack of space we have to refer to [15] for the detailed discussion of related metaprogramming systems.

Here we have to mention the very recent MacoCaml [29]: a macro-processor for OCaml based on code templates. Unlike MetaOCaml but like Template Haskell, Scala 3 or Zig, it generates code at compile-time to be used later in compilation. MacoCaml has to deal with the difficult problem of modules and module abstractions – which MetaOCaml skirts since it does not support code templates with module expressions.[16] On the other hand, MetaOCaml is more than just brackets and escapes.

7 Conclusions

Ten years have passed since MetaOCaml was first presented [15]. They have seen its increased use in research, education and even industry. It is hard to tell what the next ten years may bring. There is no shortage of problems to solve, however, with MetaOCaml, and in the further development of MetaOCaml. Among the latter are significant theoretical challenges: unsound interaction of template-based metaprogramming with polymorphism (for which [16] outlined a research program) and GADTs.[17]

Acknowledgments. I am very grateful to the users of MetaOCaml for their interest and encouragement, and also comments and suggestions. I particularly thank Jeremy Yallop for many fruitful discussions and valuable suggestions. Comments by anonymous reviewers are gratefully acknowledged. This work was partially supported by JSPS KAKENHI Grants Numbers 17K12662, 18H03218, 21K11821 and 22H03563.

[16] It is not clear if code generation of modules is needed: [10] tried to find a compelling example but ended up implementing all candidates in the ordinary MetaOCaml.

[17] For more detail on the planned features and challenges, see `NOTES.txt` in the MetaOCaml distribution.

Disclosure of Interests. The author has no competing interests to declare.

References

1. Bondorf, A.: Improving binding times without explicit CPS-conversion. In: Lisp & Functional Programming. pp. 1–10 (1992). https://doi.org/10.1145/141471.141483
2. Bussone, G.: Generating OpenMP code from high-level specifications (Aug 2020), Internship report to ENS
3. Calcagno, C., Taha, W., Huang, L., Leroy, X.: Implementing multi-stage languages using ASTs, gensym, and reflection. In: GPCE. pp. 57–76. No. 2830 in Lecture Notes in Computer Science (22–25 Sep 2003). https://doi.org/10.1007/978-3-540-39815-8˙4
4. Carette, J., Kiselyov, O.: Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. Science of Computer Programming **76**(5), 349–375 (2011). https://doi.org/10.1016/j.scico.2008.09.008
5. Chen, C., Xi, H.: Meta-programming through typeful code representation. Journal of Functional Programming **15**(6), 797–835 (2005). https://doi.org/10.1017/S0956796805005617
6. Danvy, O., Malmkjær, K., Palsberg, J.: Eta-expansion does The Trick. ACM Transactions on Programming Languages and Systems **18**(6), 730–751 (1996)
7. Davies, R., Pfenning, F.: A modal analysis of staged computation. Journal of the ACM **48**(3), 555–604 (May 2001)
8. Ershov, A.P.: On the partial computation principle. IPL: Information Processing Letters **6**(2), 38–41 (1977)
9. Hirohara, K.: Generating GPU kernels from high-level specifications using MetaOCaml (Feb 2019), Tohoku University, Master Thesis, in Japanese
10. Inoue, J., Kiselyov, O., Kameyama, Y.: Staging beyond terms: prospects and challenges. In: Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM. pp. 103–108. ACM (Jan 2016). https://doi.org/10.1145/2847538.2847548
11. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice-Hall, Englewood Cliffs, NJ (Jun 1993), `http://www.itu.dk/people/sestoft/pebook/pebook.html`
12. Kameyama, Y., Kiselyov, O., Shan, C.c.: Shifting the stage: Staging with delimited control. Journal of Functional Programming **21**(6), 617–662 (2011). https://doi.org/10.1017/S0956796811000256
13. Kameyama, Y., Kiselyov, O., Shan, C.c.: Combinators for impure yet hygienic code generation. Science of Computer Programming **112 (part 2)**, 120–144 (Nov 2015). https://doi.org/10.1016/j.scico.2015.08.007
14. Kiselyov, O.: Delimited control in OCaml, abstractly and concretely. Theor. Comp. Sci. **435**, 56–76 (Jun 2012). https://doi.org/10.1016/j.tcs.2012.02.025
15. Kiselyov, O.: The design and implementation of BER MetaOCaml - system description. In: FLOPS. pp. 86–102. No. 8475 in Lecture Notes in Computer Science, Springer (2014). https://doi.org/10.1007/978-3-319-07151-0˙6
16. Kiselyov, O.: Generating code with polymorphic let: A ballad of value restriction, copying and sharing. EPTCS **241**, 1–22 (2017). https://doi.org/10.4204/EPTCS.241.1
17. Kiselyov, O.: Reconciling Abstraction with High Performance: A MetaOCaml approach. Foundations and Trends in Programming Languages, Now Publishers (2018). https://doi.org/10.1561/2500000038

18. Kiselyov, O.: BER MetaOCaml N114. `https://okmij.org/ftp/ML/MetaOCaml.html` (May 2023)
19. Kiselyov, O.: Generating C: Heterogeneous metaprogramming system description. Sci. Comput. Program. **231**, 103015 (2023). https://doi.org/10.1016/J.SCICO.2023.103015
20. Kiselyov, O., Biboudis, A., Palladinos, N., Smaragdakis, Y.: Stream fusion, to completeness. In: POPL '17: Conference Record of the Annual ACM Symposium on Principles of Programming Languages. pp. 285–299. ACM Press, New York (Jan 2017). https://doi.org/10.1145/3009837
21. Kiselyov, O., Imai, K.: Session types without sophistry. In: Functional and Logic Programming. Lecture Notes in Computer Science, vol. 12073, pp. 66–87. Springer International Publishing (2020). https://doi.org/10.1007/978-3-030-59025-3˙5
22. Kiselyov, O., Swadi, K.N., Taha, W.: A methodology for generating verified combinatorial circuits. In: EMSOFT. pp. 249–258 (27–29 Sep 2004)
23. Kiselyov, O., Taha, W.: Relating FFTW and split-radix. In: ICESS. pp. 488–493. No. 3605 in Lecture Notes in Computer Science (2005)
24. Kiselyov, O., Yallop, J.: let (rec) insertion without effects, lights or magic. CoRR **abs/2201.00495** (2022). https://doi.org/10.48550/arxiv.2201.00495
25. Krishnaswami, N.R., Yallop, J.: A typed, algebraic approach to parsing. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI. pp. 379–393. ACM (2019). https://doi.org/10.1145/3314221.3314625
26. Lawall, J.L., Danvy, O.: Continuation-based partial evaluation. In: Lisp & Functional Programming. pp. 227–238 (1994). https://doi.org/10.1145/182409.182483
27. Radanne, G.: Tierless Web programming in ML. Ph.D. thesis, Université Sorbonne Paris Cité, Paris, France (Nov 2017)
28. Swadi, K., Taha, W., Kiselyov, O., Pašalić, E.: A monadic approach for avoiding code duplication when staging memoized functions. In: PEPM. pp. 160–169 (2006)
29. Xie, N., White, L., Nicole, O., Yallop, J.: MacoCaml: Staging composable and compilable macros. Proc. ACM Program. Lang. **7**(209), 604—-648 (2023). https://doi.org/10.1145/3607851
30. Yallop, J.: Staged generic programming. Proc. ACM Program. Lang. **1**(ICFP), 29:1–29:29 (2017). https://doi.org/10.1145/3110273
31. Yallop, J., Kiselyov, O.: Generating mutually recursive definitions. In: Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. pp. 75–81. PEPM 2019, ACM, New York, NY, USA (2019). https://doi.org/10.1145/3294032.3294078
32. Yallop, J., Xie, N., Krishnaswami, N.: flap: A deterministic parser with fused lexing. Proc. ACM Program. Lang. **7**(PLDI), 1194–1217 (2023). https://doi.org/10.1145/3591269