# The Design and Implementation of BER MetaOCaml

## System Description

http://okmij.org/ftp/ML/MetaOCaml.html

FLOPS 2014
June 4, 2014

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.

# The Death and Resurrection of MetaOCaml

## System Description

http://okmij.org/ftp/ML/MetaOCaml.html

FLOPS 2014
June 4, 2014

The title of this talk should've been 'The death and resurrection of MetaOCaml'. I've been using MetaOCaml since 2003 and can talk a lot about its evolution from personal experience. Alas, I don't have time for that. Therefore, to save time I'll have to skip death and move straight to resurrection.

# Outline

- Introduction to BER MetaOCaml
- Showing off the scope extrusion check
- Implementation

Thus I'll be talking about the *new* MetaOCaml, or BER MetaOCaml N101. I will be talking about its most distinguished feature: the scope extrusion check. It is a complex feature and takes some time to explain. I will start with a simpler introduction to MetaOCaml.

# MetaOCaml look and feel

MetaOCaml is a superset of OCaml for writing code generators

| | MetaOCaml | is not quite like | Lisp |
|---|---|---|---|
| bracket | ⟨x + y⟩ | quasiquote | `(+ x y) |
| escape | ~body | unquote | ,body |
| run | !. code | eval | (eval code) |
| persist | ⟨ pi ⟩ | | `( ',pi ) |

BER MetaOCaml is a conservative extension of OCaml with staging annotations to construct and run typed code values. MetaOCaml code without staging annotations is regular OCaml 4.

MetaOCaml adds to OCaml brackets and escapes to construct code values, and `run` (or, eval) to execute them. Brackets and escapes look quite like Lisp's quasi-quotation. There is another feature: the ability to use within brackets identifiers bound outside brackets. This is called cross-stage persistence, CSP for short. Lisp also has something like that, but not quite. The next talk is specifically about CSP, so I skip CSP.

# MetaOCaml look and feel

MetaOCaml is a superset of OCaml for writing code generators

| | MetaOCaml | is not quite like | Lisp |
|---|---|---|---|
| bracket | ⟨x + y⟩ | quasiquote | `(+ x y) |
| escape | ∼body | unquote | ,body |
| run | !. code | eval | (eval code) |
| persist | ⟨ pi ⟩ | | `( ',pi ) |

⟨**fun** x → ∼(**let** body = ⟨x⟩ **in** ⟨**fun** x → ∼body⟩ )⟩
⇝ ⟨**fun** x_1 → **fun** x_2 → x_1⟩

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

The MetaOCaml-generated code is well-typed and well-scoped

Here is a small example, which also shows that the generated code can be printed, even the code of functions. The expression .<x>. is a code value that represents a free variable, to be bound later on. So, MetaOCaml can manipulate open code and deal with variables so to speak symbolically.

The example is meant to illustrate hygiene, and the crucial difference between brackets and antiquotation in Lisp. MetaOCaml respects lexical scoping!

If we write the example in Lisp and use antiquotation and unquotation, the generated code would have two instances of x, indistinguishable. The generated code will mean quite a different thing though. MetaOCaml maintains the distinction between the variables that although named identically like x but bound at different places. So, a variable in MetaOCaml is not just a symbol.

# MetaOCaml look and feel

MetaOCaml is a superset of OCaml for writing code generators

| | MetaOCaml | is not quite like | Lisp |
|---|---|---|---|
| bracket | ⟨x + y⟩ | quasiquote | `(+ x y) |
| escape | ~body | unquote | ,body |
| run | !. code | eval | (eval code) |
| persist | ⟨ pi ⟩ | | `( ',pi ) |

```
`( lambda (x) ,( let ((body `(+ x 1)))
   `( lambda (x) (string -append ,body x))))
⇝ (lambda (x) (lambda (x) (string -append (+ x 1) x)))

(((lambda (x) (lambda (x) (string -append (+ x 1) x))) 1) "a")
⇝ Error in +: "a" is not a number.
```

Let's look again at the slightly changed Scheme generator, which produces the shown code. The generated code can be successfully evaluated and applied. It is only we submit the second argument that we see a problem. By that time, the original generator has long finished. It is very hard now to tell which part of the generator is responsible for the problem and how to fix it.

# MetaOCaml look and feel

MetaOCaml is a superset of OCaml for writing code generators

| | MetaOCaml | is not quite like | Lisp |
|---|---|---|---|
| bracket | ⟨x + y⟩ | quasiquote | `(+ x y) |
| escape | ∼body | unquote | ,body |
| run | !. code | eval | (eval code) |
| persist | ⟨ pi ⟩ | | `( ',pi ) |

⟨**fun** x → ∼(**let** body = ⟨x+ 1⟩ **in** ⟨**fun** x → x ^ ∼body⟩ )⟩

⇝

⟨**fun** x → ∼(**let** body = ⟨x+ 1⟩ **in** ⟨**fun** x → x ^ ∼body⟩ )⟩
^^^^

Error: This expression has **type** int code
but an expression was expected **of type** string code
Type int is not compatible **with type** string

The MetaOCaml-generated code is well-typed and well-scoped

MetaOCaml is typed, and so typing problems with the *generated* code are reported right away, when type checking the *generator*, before even running it. The error is hence reported in terms of the generator. We see from the error message that code values have their own types like int code. Although the generated code is compiled later, it is type checked *now*.

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 generates only well-typed programs: The generated code shall compile without type errors. There are no longer problems of puzzling out a compilation error in the generated code (which is typically large, obfuscated and with unhelpful variable names).

The above benefits all come about because MetaOCaml is typed.

Types, staged types in particular, help write the code.

# BER MetaOCaml N101

- A clean-slate re-implementation
- Different algorithms, different data structures
- *Different design decisions*
- Extensive comments, regression test suite
- Modular structure: easier to maintain, easier to contribute
- The operation to run the code: user-definable, no longer a built-in
- No environment classifiers
- Generated code is always well-typed and well-scoped, even in the presence of effects

BER N101 is the current (and the only) version MetaOCaml. It is a complete re-implementation of MetaOCaml. It has not only new code and new algorithms, but also new design decisions. I stress the main differences: modular structure, making it easier to maintain and contribute to – and especially the highlighted ones. The highlighted features are new, and I'll talk about them next.

BER MetaOCaml is a re-implementation of MetaOCaml. It has not only new code and new algorithms, but also new design decisions. It also has comments in the code, and a regression test suite! There only small piece inherited from the old MetaOCaml are the changes to OCaml parser and lexer to recognize brackets, escape, and run.

The goal of the BER MetaOCaml project is to reduce as much as possible the differences between MetaOCaml and the mainline OCaml, to make it easier to keep MetaOCaml up-to-date and ensure its long-term viability. We aim to find the most harmonious way of integrating staging with OCaml, with the remote hope that some of the changes would make it to the main OCaml branch.

# Outline

- Introduction to BER MetaOCaml
- Showing off the scope extrusion check
  - larger example of code generation
  - abstracting code generators: building DSLs
  - effects in code generation (let-insertion)
  - danger of scope extrusion
  - scope extrusion check
  - convenient and safe let-insertion *for the first time*
- Implementation

Here is the more detailed outline for the rest of the talk. We use a larger example to show off the code generation with effects, the danger of scope extrusion and how BER MetaOCaml prevents it.

# Matrix-matrix multiplication

$$c_{ij} = \sum_k a_{ik} b_{kj}$$

## Many variations

- single, double-precision FP numbers, integers, ...
- different matrix representations: row-major, column-major, tiled, sparse
- unrolling loops, fully or partly
- *let-insertion*
- loop interchange
- loop tiling

all have to be efficient

## Need DSL

Everyone knows how to multiply two matrices A and B with the result in C (assumed zeroed out), so I don't have to explain it. The example is real – an incredible amount of effort in HPC is spent optimizing matrix-matrix multiplication.

The example is simple, but with many variations: the matrix may be represented in many ways, etc. All the variants must be utmost efficient. To increase performance, we may need to unroll the loops, by an architecture-specific amount. Code generation is inevitable. We concentrate on let-insertion (loop interchange and tiling with MetaOCaml has been described elsewhere).

# Small Linear Algebra DSL

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

To handle the many variations of the matrix-matrix multiplication, let's make a DSL. We abstract out the type of the scalars, tdom, the type of the index tind, the unit type and of course the matrix type. We define arithmetic on scalars, getting the dimensions of the matrix, accessing an element of the matrix given its indices, and increment it. And we need an operation to do loops.

# Generic matrix-matrix multiplication

$$c_{ij} = \sum_k a_{ik} b_{kj}$$

```
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
```

# Meta-circular implementation

```
module LAint = struct
  type tdom    = int
  type tind    = int
  type tunit   = unit
  type tmatrix = int  array  array
  let  ( + )   = Pervasives.( + )
  ...
```

The first implementation of our DSL is metacircular. It is makes matrix multiplication very slow, but is useful for testing.

# Code-generating implementation

```
module LAintcode = struct
  type tdom    = int  code
  type tind    = int  code
  type tunit   = unit 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

 ⟨fun a  b  c  →
   ∼(let  module M = MMUL(LAintcode) in
       M.mmul ⟨a⟩ ⟨b⟩  ⟨c⟩  )
   ⟩
```

The second implementation of the signature LINALG uses MetaOCaml to generate code. The domain of scalars is integer *code expressions*; the operation plus now generates the code of addition rather than adding the numbers.

Using the same DSL code with the code-generating interpretation gives us the following code.

# Generated code

```
val smmul1 :
  (int array array → int array array → int array array → unit) co
⟨fun a_1 b_2 c_3 →
    for i_4 = 0 to (Array.length a_1) − 1 do
     for i_5 = 0 to (Array.length b_2) − 1 do
      for i_6 = 0 to (Array.length (b_2.(0))) − 1 do
        c_3.( i_4 ).( i_6 ) ←
        c_3.( i_4 ).( i_6 ) + a_1.( i_4 ).( i_5 ) ∗ b_2.( i_5 ).( i_6 )
      done
     done
    done⟩
```

The generated code looks how we expected it to look: three nested loops. We can save the code in a file and compile to build a library of various matrix-matrix multiplications.

# Loop-unrolling

```
module LAintcode_unroll (S:sig val unroll_factor : int end) =
struct
  include LAintcode
  let loop n body = ...
    ...
end


⟨fun a b c →
  ∼(let module M =
      MMUL(LAintcode_unroll(struct let unroll_factor = 2 end)) in
    M.mmul ⟨a⟩ ⟨b⟩ ⟨c⟩ )
⟩
```

We can also partially unroll loops by the given factor – using the same generic code MMUL code. We include the LAintcode implementation and redefine the loop combinator to do unrolling. The generated code, with partially unrolled loops, is the complete mess – as expected. If the code generation is done right, we never have to look at the generated code.

We can do further code transformations like that. We concentrate on a different one.

# Moving loop-invariant code?

```
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
```

## Goal

- implement the moving mat_get a i k out of the loop
- do *not* modify MMUL
- program this optimization by writing a different implementation of LINALG

Let's look again at the generic matrix-matrix multiplication code. We notice the expression of accessing $a_{ik}$ in the inner loop that does not depend on the loop variable j. It ought to be moved out. Can we program this optimization, again without modifying this code, using a different interpretation of LINALG?

Of course in real-file, the compiler may notice that a.(i).(k) does not depend on the index j and automatically move the code. But for matrices with the complex layouts, the access operation may be a function call. The compiler cannot (and ought not) to move the code, unless it can see the access function is pure. Sometimes it is not, if a matrix is too large and has to be stored on disk. Here the compiler really cannot move the code, without any domain-specific knowledge.

# Let-insertion

$$\textbf{val}\ \text{genlet} : \omega\ \text{code prompt} \to \alpha\ \text{code} \to \alpha\ \text{code}$$
$$\textbf{val}\ \text{with\_prompt} : (\omega\ \text{prompt} \to \omega) \to \omega$$

$$\text{with\_prompt}\ (\textbf{fun}\ \text{p} \to \langle 1 + {\sim}(\text{genlet}\ \ \text{p}\ \langle 2 + 3\rangle)\rangle\ )$$
$$\rightsquigarrow\ \ \langle \textbf{let}\ \ \text{t\_1}\ = 2 + 3\ \textbf{in}\ 1 + \text{t\_1}\rangle$$

$$\text{with\_prompt}\ (\textbf{fun}\ \text{p} \to \langle \textbf{fun}\ \text{x} \to \text{x} + {\sim}(\text{genlet}\ \ \text{p}\ \langle 2 + 3\rangle)\rangle\ )$$
$$\rightsquigarrow\ \ \langle \textbf{let}\ \ \text{t\_5}\ = 2 + 3\ \textbf{in}\ \textbf{fun}\ \text{x\_4}\ \ \to \text{x\_4}\ + \text{t\_5}\rangle$$

So, we need so-called let-insertion. Let's talk about it a bit on a simple example. Let-insertion is accomplished by these two functions: genlet takes a code expression to bind and inserts let somewhere up. The function with_prompt marks the place where to insert this let. These two functions communicate via so-called prompt. In the first example, genlet took expression 2+3 and let-bound it where with_prompt was. The binding place can be arbitrarily away from genlet, as the second example shows. This is of course very desirable: in the second example, we generate code in which 2+3 is computed only once rather than on each call to the function. But this also can be very dangerous.

# Problematic let-insertion

Up to two years ago:

```
with_prompt (fun p →
  ⟨fun x → x + ∼(genlet  p ⟨x+ 3⟩)⟩ )
```

$\rightsquigarrow$ `⟨let t_5 = x_4 + 3 in fun x_4 → x_4 + t_5⟩`

Scope extrusion!

Environment classifiers do not help!

Consider the example on the slide. We attempt to move out the expression that contains x outside the binding of x! Before, this attempt was successful. We could truly generate the shown code, which exhibits so-called scope extrusion.

Some of you may have heard of environment classifiers in the old MetaOCaml. Alas, they do not help with scope extrusion. That's why they have been removed.

# Problematic let-insertion

Now:

```
with_prompt (fun p →
  ⟨fun x → x + ∼(genlet  p ⟨x+ 3⟩)⟩ )

⇝ propagating exc Exception: Failure
 Scope extrusion detected at Characters 89−117 for code built at:
    ⟨fun x → x + ∼(genlet  p ⟨x+ 3⟩)⟩ );;
                                  ^^^
 for the identifier   x_6 bound at Characters 39−40:
    ⟨fun x → x + ∼(genlet  p ⟨x+ 3⟩)⟩ );;
          ^
```

In the current version of BER MetaOCaml, the example type checks as before. However, running it no longer succeeds. Rather, running the generator throws the exception with a rather informative message.

# Implementing let-insertion

```
open Delimcc

let genlet : ω code prompt → α code → α code = fun p cde →
    shift p (fun k → ⟨let t = ∼cde in ∼(k ⟨t⟩ )⟩ )

let with_prompt : (ω prompt →ω) → ω = fun thunk →
  let p = new_prompt () in
  push_prompt p (fun () → thunk p)
```

Let-insertion is user-defined, not a primitive

Importantly, genlet is not a primitive in MetaOCaml. It is an ordinary library function written with the Delimcc library of delimited control.

# Let-insertion: summary

- Generating code with (control) effects
- Let-insertion: powerful and needed, but dangerous
- Safety guarantee, finally:
  if the code is successfully generated, it is well-scoped (and well-typed)

We have just seen how to generate code with control effects, that let-insertion is highly desirable and highly dangerous, and that in the present MetaOCaml, it is finally safe. It is safe in the following sense: if the generator successfully finished generating the code, the result is well-typed and well-scoped.

# Let-insertion: summary

- Generating code with (control) effects
- Let-insertion: powerful and needed, but dangerous
- Safety guarantee, finally:
  if the code is successfully generated, it is well-scoped (and well-typed)

Now, let-insertion is safe. But is it convenient?

We have seen that let-insertion is safe. But is it convenient? We have to explicitly mark the place where to insert. If there are multiple prompts in scope, that is, locations to insert, we, or the user, have to choose. Although we can accomplish let-insertion for our matrix-matrix multiplication, we cannot move out the matrix-indexing expression without modifying the generic code. At least, without complex logic. Complexities snowball...

# Safe and convenient let-insertion

Can't you just insert let where it causes no exceptions?!

The programmer is tempted to shout to the program: can't you just insert let where it causes no exceptions? Come to think of it, if we get an exception when inserting let at a wrong place, can't we just try inserting at a higher and higher place, where it is still safe?

# Safe and convenient let-insertion

Can't you just insert let where it causes no exceptions?!

**val** genlet : $\alpha$ code $\to$ $\alpha$ code
**val** let_locus : (unit $\to$ $\omega$ code) $\to$ $\omega$ code

- Library functions, not primitives
- Convenient, and safe
  for the first time

The answer is yes, and the following two library functions do exactly this. They are safe since they are not MetaOCaml primitives and don't modify MetaOCaml. We rely on the existing guarantee: so long as no exception is raised, the code is well-typed and well-scoped.

For the first time, we show off self-adjusting, safe *and* convenient let-insertion with static guarantees. This part is *not* in the paper.

# Safe and convenient let-insertion

```
module LAintcode_opt = struct
  include LAintcode
  let mat_get a i j = genlet @@ LAintcode.mat_get a i j
  let loop n body =
    let_locus (fun () →
      LAintcode.loop n
        (fun i → let_locus (fun () → body i)))
end

⟨fun a b c →
  ~(let module M = MMUL(LAintcode_opt) in
    M.mmul ⟨a⟩ ⟨b⟩ ⟨c⟩ )
  ⟩
```

(Convenient let-insertion is not in the paper)

What is shown are all the changes. Fully re-using the earlier implementation LAintcode, we ask to let-bind all matrix access operations and indicate that the places before and after the loop are good locations to insert the code at.

# Generating code with let-insertion

```
val smmul3 :
  (int array array → int array array → int array array → unit) co
⟨fun a_124 b_125 c_126 →
    for i_127 = 0 to (Array.length a_124) − 1 do
      for i_128 = 0 to (Array.length b_125) − 1 do
        let t_131 = (a_124.(i_127)).(i_128) in
        for i_129 = 0 to (Array.length (b_125.(0))) − 1 do
          let t_130 = (b_125.(i_128)).(i_129) in
          c_126.(i_127).(i_129) ←
          c_126.(i_127).(i_129) + t_131 ∗ t_130
        done
      done
    done⟩
```

now, the same generic code produces the shown code. Matrix access operations are indeed let-bound, at the appropriate places, ‘as high’ as possible.

# Implementation

The most useful part of MetaOCaml is OCaml

## A patch to OCaml

- using *abstraction facilities of OCaml*
- back-ends for free
- source compatible with OCaml (familiarity)
- binary compatible with OCaml (reuse of libraries, tools, etc)
- part of the OCaml community

However ...

The upside of using OCaml, mature language: code generator and backends, tools, libraries, familiarity, community. It will take a long time to re-implement OCaml from scratch.

# The downside of a language dialect

There is quite a difference between an academic paper presenting a simple calculus with a few expression forms – and the real system, with the huge amount of code with lots and lots of details and edge cases.

The other side: show typing/typecore.ml in small font, and scroll and scroll. This is just one filel unification is in a separate file, dealing with typing environment is in another file, type checking of modules is another file, etc. The total lines of code in the type checker: do "wc -l *.ml" Let me zoom-in on the code: typecore.ml (in a larger font): code, code, code. Hardly any comments, except for this. But this comment is mine. This is an important place in the type checker for the MetaOCaml interface. There are other places in this typecore.ml file with MetaOCaml changes. To make the changes, one has to have a good idea what all these 27000+ code lines are doing (and it some places, one has to have a very good idea).

Let me also show what was involved with re-writing MetaOCaml: old trx.ml; code, code, hardly any comments. Here is the new, re-written. From the distribution of color you can see my style is different. There are many comments, in blue color. I hope my successor will have easier time understanding what's going on.

I hope I demonstrated that there is quite a difference between an academic paper presenting a simple calculus with 3-4-7 expression forms and the real system, with the huge amount of code with lots and lots of details and edge cases. Dealing with the real system motivated the split into kernel and user-level in MetaOCaml.

# MetaOCaml

MetaOCaml is a superset of OCaml for writing code generators (and generators of code generators, etc.)

- A conservative extension of OCaml
- Pure generative: no examination of the generated code
- Generators and the generated code are *typed*
- Guaranteeing the generation of ...
  - the well-formed code
  - the well-typed code
  - code with no unbound or unexpectedly bound identifiers
- Reporting errors in terms of the generator rather than the generated code
- Generators take advantage of all abstraction facilities of ML (higher-order functions, modules, objects, etc)

BER MetaOCaml is a conservative extension of OCaml with staging annotations to construct and run typed code values. MetaOCaml code without staging annotations is regular OCaml 4.

First, the generated code is assuredly well-formed: all parentheses match. This is better than using printf to generate C (cf. ATLAS). 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 generates only well-typed programs: The generated code shall compile without type errors. There are no longer problems of puzzling out a compilation error in the generated code (which is typically large, obfuscated and with unhelpful variable names).

The generated code is well-scoped: there are no unbound variables in the generated code and no insidious surprisingly bound variables.

The above benefits all come about because MetaOCaml is typed.

Types, staged types in particular, help write the code.

# Parting thoughts

The successfully generated code is well-typed and well-scoped, and always compiles

. . . unconditionally, no matter what effects were used when generating the code.

Scope extrusion check: not only prevents but also enables