MetaOCaml lives on

Lessons from implementing a staged dialect of a functional language

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

ML 2013 Workshop
September 22, 2013

We report the lessons of implementing and re-implementing MetaOCaml, which is a superset of OCaml extending it with staging annotations to construct and run typed code values. It is intended to be a close dialect of OCaml so to share in OCaml's continuing improvement and users. The desire to make as few changes to the core OCaml as possible dictated a number of sometimes non-obvious design decisions. We explicate them as hints for implementors of staged language dialects. We conclude with a research and development agenda for typed staged languages, inviting contributions.

Outline

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.

MetaOCaml look and feel

	MetaOCaml	is not quite like	Lisp
bracket	$\langle$x + y$\rangle$	quasiquote	`` `(+ x y) ``
escape	~body	unquote	`,body`
run	.!code	eval	`(eval code)`
persist	$\langle$ pi $\rangle$		`` `( ',pi ) ``

I borrowed this introductory slide from the ML 2010 talk. 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. We'll talk about CSP later.

MetaOCaml look and feel

	MetaOCaml	is not quite like	Lisp
bracket	$\langle$x + y$\rangle$	quasiquote	`` `(+ x y) ``
escape	~body	unquote	`,body`
run	.!code	eval	`(eval code)`
persist	$\langle$ pi $\rangle$		`` `( ',pi ) ``

$\langle$**fun** x → ~(**let** body = $\langle$x$\rangle$ **in** $\langle$**fun** x → ~body$\rangle$)$\rangle$
⇝ $\langle$**fun** x_1 → **fun** x_2 → x_1$\rangle$

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

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 variables that although named identically like `x` but bound at different places. So, a variable in MetaOCaml is not just a symbol.

Outline

Brief History

Original MetaOCaml

- a fork of OCaml along the lines of MetaML
- designed and architectured by Walid Taha and developed by Cristiano Calcagno
- started in September 2000, reached its current form by 2003, last released in 2006

MetaOCaml started as a fork of OCaml along the lines of MetaML. Designed and architectured by Walid Taha and developed by Cristiano Calcagno, MetaOCaml had reached its current form by 2003. The increasing divergence between OCaml and MetaOCaml made it harder and harder to merge the changes. With the funds for the project dried up and the daunting prospect of merging many changes that appeared in OCaml 3.10 and 3.11, the development of MetaOCaml has ceased. Its last released version was 3.09.1 alpha 030.

BER N100

- A clean-slate re-implementation
- Different algorithms, different data structures
- *Different design decisions*
- Extensive comments
- Smaller kernel and closer to OCaml,
  to ease maintenance

kernel	49K patch to OCaml files, 77K new `trx.ml` 28 OCaml files patched, 6 of which trivially
user-level	58K pretty-printer (Jacques Carette), 3K running the code and top level
tests	54K Regression test suite!

Other differences: `let!`, fixing a few old bugs, better CSP

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.
BER N100 is much less invasive into OCaml. Only 28 of OCaml files are patched, of which 6 trivially (one or two lines, AST printing/dumping). Previously (BER N004) the patch to typing/typecore.ml had 564 lines of additions, deletions and context; now, only 328 lines. (The core MetaOCaml is trx.ml, with 1800 lines.)
The code is restructured into kernel, responsible for producing and type-checking code values, and user level. Processing code values such as printing and executing is done in the 'user-level' libraries. Programmers may write new ways of processing code values, e.g., to compile them to LLVM or JavaScript, without modifying (Meta)OCaml.

Outline

Implementing staging I

- Add staging forms to AST
- Add staging forms to the typed AST
- Add staging forms to IL
- Account for staging forms in the code generator

…might have just as well re-implement the language

So, how can we *add* MetaOCaml-like staging to your functional language system?

The most straightforward way of adding staging to a functional language is the most difficult one. We have to modify everything. Perhaps it is simpler to design the language afresh, with staging from the outset. But there is a much simpler way.

Implementing staging II

Pre-process the staging away

fun $x \to \langle$**fun** $y \to \sim x * y + 1\rangle$

pre-process to

fun x → lam "y" (**fun** y → (add (mul x y) (int 1)))

where *code combinators* are:

type α cod
val int : int → int cod
val add: int cod → int cod → int cod
val mul: int cod → int cod → int cod
val lam: string → (α cod → β cod) → (α→β) cod

A much simpler approach is to translate, or preprocess away, brackets into expressions built with code combinators, of the following signature. When we evaluate the expression, we generate the code. It is that simple.

Implementing staging II

- + easy to implement (preprocessor, camlp4, . . .)
  no modifications to the base language
- + typing is preserved!

Works surprisingly well. Cf. Scala's LMS

Does this really work? Yes, it does. In fact, Scala's Lightweight Modular Staging uses a similar idea.

The technique can be implemented without modifying the base language compiler at all, by writing a stand-alone preprocessor.

The best news is that typing is preserved! That is, if the postprocessed code type checks, the original code is well-typed by MetaOCaml rules, and hence the generated code will be well-typed.

Implementing staging II

+ easy to implement (preprocessor, camlp4, . . .)
  no modifications to the base language
+ typing is preserved!

Works surprisingly well. Cf. Scala's LMS

– conditionals, loops, other special forms?
  introduce thunks

There are complications however. How to deal with special forms such as conditionals and loops? That seems straightforward: introduce thunks.

Implementing staging II

- + easy to implement (preprocessor, camlp4, . . .)
  no modifications to the base language
- + typing is preserved!

Works surprisingly well. Cf. Scala's LMS

- – conditionals, loops, other special forms?
  introduce thunks
- – pattern-matching?

- – type-annotations?

What about pattern-matching forms and type annotations?

Implementing staging II

+ easy to implement (preprocessor, camlp4, . . .)
  no modifications to the base language
+ typing is preserved!

Works surprisingly well. Cf. Scala's LMS

– conditionals, loops, other special forms?
  introduce thunks
– pattern-matching?
  ugly but hackable: see Scala-Virtualized
– type-annotations?
  ugly but hackable

That is quite a nasty problem, but solvable, by translating to deconstructors and special type annotation functions (like Haskell's typeAs). The paper Scala-Virtualized talks about such translations in detail.

Implementing staging II

- + easy to implement (preprocessor, camlp4, …)
  no modifications to the base language
- + typing is preserved!

Works surprisingly well. Cf. Scala's LMS

- – conditionals, loops, other special forms?
  introduce thunks
- – pattern-matching?
  ugly but hackable: see Scala-Virtualized
- – type-annotations?
  ugly but hackable
- – let-polymorphism?
  Fatal
- – polymorphic code values?
  Prevented by value-restriction

Alas, the polymorphic let like `.<let f = fun x -> x in ...>.` is a show stopper.
The let-binding within brackets becomes after translation the lambda-binding, which can't be polymorphic. Polymorphism in general is problematic in this approach: since code is translated to an expression, a polymorphic code value (e.g., `let f = .<fun x -> x>.`) will be translated into the monomorphic expression, due to the value restriction.

Implementing staging III

Translate staging away *after type-checking*

```
fun x → ⟨fun y → ~x * y + 1⟩
```

becomes

```
fun x → let  y = gensym "y" in
        mkLam y (mkApp (mkIdent "+")
                       [mkApp (mkIdent "*") [x; y];
                        mkConst 1])
```

Future-stage bound variable becomes present-stage bound variable, but at the type string loc
Why do we need gensym at run-time?

+ Future-staged code type-checked properly, including polymorphism
+ No modification of the IL or the back-end
– Have to modify the type-checker
– Have to modify the AST

This is the approach used in MetaOCaml. No need to modify the back-end (the code generator) but must modify the type checker to type check staging constructs. Luckily, the modifications are not that big. See the typing rules in many MetaOCaml papers.

The future-stage code is type-checked properly, including polymorphic constructs. The translation of staging away looks like the one before, but it occurs after the type-checking.

What are the code values?

What is α code?

- machine code or IR:
  difficult to optimize and compose
- typed AST:
  difficult to compose (e.g., deal with type environment, polymorphism)
- AST

If a polymorphic record is spliced in in two places, we must refresh the type variables. Dealing with the type environment (merging, etc.) is difficult, especially when type variables are concerned.

Outline

BER vs old MetaOCaml

Substantial differences between BER and old MetaOCaml

- constructor restriction
- scope extrusion check
  prevents building code with unbound or mistakenly bound variables,
  even in the presence of effects

BER MetaOCaml N100 differs from the old MetaOCaml in two substantive respects:

1. Constructor restriction: all data constructors and record labels used within brackets must come from the types that are declared in separately compiled modules;
2. Scope extrusion check: attempting to build code values with unbound or mistakenly bound variables (which is possible with mutation or other effects) is caught early, raising an exception with good diagnostics.

Constructor problem

Generator

$\textbf{type}$ foo = Foo | Bar $\textbf{of}$ int
⟨**let** x = Foo **in** **match** x **with** Bar z →z⟩

Generated code

let x = Foo **in** **match** x **with** Bar z → z

Here is the constructor problem: the shown MetaOCaml code would generate code with undefined constructors Foo and Bar. The generated code won't compile!

Constructor problem

Generator

```
type foo = Foo | Bar of int
⟨let x = Foo in match x with Bar z →z⟩
```

Generated code

```
let x = Foo in match x with Bar z → z
```

A very painful solution in the old MetaOCaml,
which contributed to its demise

The old MetaOCaml had a very painful solution: storing the entire type environment, in a hackish way, in the AST representing the generated code. The generated code was a bizarre mixture of the untyped and typed AST, which required very many changes to the OCaml typechecker. OCaml typechecker is very complex, and is getting more complex. We really don't want to patch it a lot. That invasiveness was one of the factors that MetaOCaml could not be maintained: too much effort to keep up with OCaml.

Constructor problem: Solution 1

Generator

$\langle$**let** x = None **in** **match** x **with** Some z $\rightarrow$z$\rangle$

Generated code

let x_18 = None **in** (**match** x_18 **with** Some (z_19) $\rightarrow$ z_19)

Constructor restriction

All data constructors and record labels used within brackets must come from the types that are declared in separately compiled modules

There is no problems here: the data type `option` is in a standard library, which is presumed available to a compiler. This points to a solution, the constructor restriction that is currently implemented: all data constructors and record labels used within brackets must come from the types that are declared in separately compiled modules.

Constructor problem: Solution 2

Generator

```
type foo = Foo | Bar of int
⟨let x = Foo in match x with Bar z →z⟩
```

Generated code

```
let module M =
  struct
    type foo = Foo | Bar of int
  end
in let open M in
let x = Foo in match x with Bar z →z
```

The code value is the *closure* over the constructor environment

OCaml offers another solution. I'm thinking about it. So far, the constructor restriction isn't really bothersome.

Outline

CSP complexities

Polymorphic lift

let lift x $=$ $\langle$x$\rangle$
$\rightsquigarrow$ **val** lift : $\alpha \to (\beta,\ \alpha)$ code $=$ <**fun**>

Aliasing or copying?

let r $=$ **ref** 0 **in**
let c $=$ $\langle$**let** () $=$ r $:=$ 1 **in** !r$\rangle$ **in**
(.! c, !r)

Really copying?

let c $=$ open_in "/etc/motd" **in** lift c;;
$\rightsquigarrow$ $(\alpha,$ in_channel $)$ code $=$
$\langle$*(* cross−stage persistent value (as id: x) *)*$\rangle$

CSP is tantamount to a polymorphic lift, which is quite problematic. For example, what is the meaning of a CSP of a mutable cell value? It was different for bytecode vs native MetaOCaml. What is the CSP meaning for an open file?

No polymorphic lift

Planned new rules of CSP

- Base types: copy
- Global identifiers: always share
- ADT, immutable records, known-good types: copy
- Abstract, polymorphic, functional and all other types: prohibit

Feedback is appreciated

In short, only global, ‘standard library’ identifiers can be assumed to be defined at the time the generated code is compiled and run. Therefore, when such identifiers occur within brackets, the generated code can refer to them by name. For other identifiers, we CSP by value, or lift – which will work only at specific types. Those are base types, structural immutable types (data types), and abstract types whose manifest type can be deduced and it is liftable. For base types, sharing is not observable anyway. I pretend that strings are immutable (do the same for arrays?)
Values of polymorphic of functional types cannot be lifted. (CSP of code values is allowed if bound to global identifiers. In that case, the code value is ensured to be closed.) We report an error at trx time (compile-time error). Polymorphic lift becomes inexpressible.

I am very interested to know how much code has to be changed if this proposal is implemented. The feedback is greatly appreciated.

No polymorphic lift

Planned new rules of CSP

How to lift functions, code, file descriptors?

- is it needed?
- bind to global identifiers
- be explicit

```
let  glob  = ref None
let  res  = ...  glob  :=  Some v; ...
             ⟨from_some !glob⟩
```

Feedback is appreciated

Lifting has to be restricted. But we don't have bounded polymorphism in OCaml. So, I will have to use a combination of type-directed (when the concrete type is known) and run-time (alas) errors. (run-time inspection doesn't work since (1,2) and array of 1 2 have the same representation, but the latter is mutable.)
Another idea: a family of functions `lift: 'a desc -> 'a -> ('cl,'a) code` and `'a desc` constructors `intdesc: int desc` arrdesc, etc. They do type-specific lifting, depending on 'a desc
Also, it is always possible to use global identifier (global mutable cell or array). Before such a workaround happened behind the scenes in the native code. Now it has to be explicit.

Outline

Research plans

Developing staged calculi that account for

- staging and modules
  Can staging be used as a module system?
- staging and objects
- staging and GADTs

Prove that the restriction on CSP avoids the unsoundness problems with generalizations

I have no students or money, but I do have an extensive program about staging, which includes both research and development.
On the research front is developing a staged calculus that accounts for objects, modules, GADT; prove that the restriction on CSP avoids the unsoundness problems with generalizations.
Also interesting is integration of staging and modules (the latter have became first-class in OCaml). Modules are used more and more, so we need a good story. Staging and modules is a potentially huge area: can staging even be used as a module system?

Kernel level development

- Move `run` out of the kernel: BE MetaOCaml (alternatively, look into levels)
- CSP
- Native MetaOCaml

Longer term

- Relax the restriction on data type declarations

Here is the development program for the kernel part of the MetaOCaml, short- and longer-term.

Relax the constructor restriction using local modules

User-level

- Bring back (re-implement) off-shoring:
  the translation from the generated code (a *small subset* of OCaml) to C, Fortran, LLVM, Verilog, JS, etc.
- More applications, more (Shonan) challenges

Everyone can participate, everyone is welcome

Further on the development agenda: add more ways to ‘run’ code values, by translating them to C, Fortran, LLVM, Verilog. MetaOCaml can then be used for generating libraries of specialized C, etc. code.

This is user level: the barrier of entry is low, no need to be recompiling the system.

Conclusions

The experience of adding staging to a functional language

- Types help
- Piggy-backing on the existing implementation: faster start-up, easier maintenance
- Preprocessing almost works
- Representation of code values is some sort of closure over the signature (user-defined types)
- CSP are important in practice, but hard to get right

In conclusion, what can I say from experience. First, types help, and staged types help especially. So, making your staged forms typed and generating the well-typed code are good ideas.
Another good idea is to piggy-back on the existing implementation, modifying it as little as possible. It is easy to get off the ground and maintain the staged language.

User-defined types are not treated in the staged calculi, but they are very important in practice. CSP are also important in practice, and hard to get right. The proposal I showed is the result of long and painful thoughts.

Suggestions for OCaml

How the OCaml HQ might help

- New structured constant `Const_csp_value`? Tempting to remove from MetaOCaml, tempting to move to OCaml
  Use cases: references to big structures, syntactic extensions
- Merge `metalib/print_code.ml` and `tools/pprintast.ml`
  Use optional arguments to make the printing extensible
- Unify printing functions for various trees and make them prettier. The very same `fmt_longident_aux` occurs 3 times in the OCaml code base
- Better and modular error handling
  `exception Error of (format → unit)`

This is a back-up slide. I just want to show that I have this slide, to talk offline with the members of OCaml team. For now I just want to show that I have suggestions, and they are not that big. Even small things will make me happy.
What part of MetaOCaml can be moved to the OCaml proper
Merge `metalib/print_code.ml` and `tools/pprintast.ml` Both are essentially the same. We could use optional arguments to make the printing extensible and robust to AST changes.
The very same `fmt_longident_aux` occurs three times in the OCaml code base (parsing/printast.ml, tools/pprintast.ml, typing/printtyped.ml)
Currently, to add a new error, say, to typecore.ml one has to add the error to typecore.mli, make the same addition to typecore.ml, augment the printing function in typecore.ml. To add a new category, one has to augment driver/errors.ml. I have tried a uniform `exception Error of (format → unit)` The error carries the function that will print out the error message on the supplied ppf. Handler becomes much more extensible! This approach does seem to work out.

On one hand, it is tempting to eliminate `Const_csp_value`. There will no longer be a need to modify the code generator (in bytecomp/). On the other hand, `Const_csp_value` does make sense as a structured constant. Consider a constant that refers to a big array, or even array