The MetaOCaml files

Status report and research proposal

Oleg Kiselyov
FNMOC
oleg@pobox.com

Chung-chieh Shan
Rutgers University
ccshan@cs.rutgers.edu

Abstract

Custom code generation is the leading approach to reconciling generality with performance. MetaOCaml, a dialect of OCaml, is the best-developed way today to write custom code generators and assure them type-safe across multiple stages of computation. Nevertheless, the continuing interest from the community has yet to result in a mature implementation of MetaOCaml that integrates cleanly with OCaml's type checker and run-time system. Even in theory, it is unclear how staging interacts with effects, polymorphism, and user-defined data types.

We report on the status of the ongoing MetaOCaml project, focusing on the gap between theory and practice and the difficulties that arise in a full-featured staged language rather than an idealized calculus. We highlight foundational problems in type soundness and cross-stage persistence that demand investigation. We also suggest a lightweight implementation of a two-stage dialect of OCaml, as syntactic sugar.

1. Introduction

From the Berkeley Packet Filter (Begel et al. 1999) to the Fastest Fourier Transform in the West (Frigo and Johnson 2005), the world is full of metaprogramming applications. Most metaprogrammers use `printf` to generate program texts, when they are not abusing the C preprocessor or C++ templates. The ML community knows better (Kamin 1996; Sheard 2001).

Static type-checking is even more helpful to metaprogrammers than to ordinary programmers, because the typical metaprogram is both very abstract and very computation-intensive. But when it comes to assuring that a well-typed code generator not only never goes wrong but also generates well-formed programs that never go wrong, pretty much the only practical game in town today is MetaOCaml, a dialect of OCaml. Although it receives no French government funding and is maintained only by volunteers, MetaOCaml is now widely used to generate ML and C code in a broad range of applications (Carette and Kiselyov 2008; Lengauer and Taha 2006). Carette's petition drive[1] witnesses continuing interest.

MetaOCaml is not just an implementation of a small and metatheoretically convenient multistage calculus such as λ^{α} (Taha and Nielsen 2003), λ^{i}_{let} (Calcagno et al. 2004), or λ-U (Calcagno et al. 2003). Rather, MetaOCaml adds staging constructs to full OCaml, so both code generators and generated code can use user-defined data types, records, mutable state, exceptions, polymorphic variants, and so on. As one would expect, MetaOCaml is implemented by extending the OCaml implementation.

Retrofitting a production compiler with staging, without changing the underlying bytecode machine or hardware instruction set, poses both undocumented implementation challenges and unresolved theoretical issues. This talk attempts to explain the major pitfalls and their current, imperfect solutions, then invites contributions from the community in the form of code as well as theorems.

[1] `http://caml.inria.fr/mantis/view.php?id=4608`

2. Generated code is type-checked again

The metatheory of multistage programming assures us that a well-typed code generator never generates code that is ill-formed (for example, containing an unbound variable) or goes wrong. In addition to making it easier to develop code generators that work, this theorem might make one think that a MetaOCaml program that generates some code then runs it would not need to type-check the generated code at all. That is unfortunately not the case.

The OCaml compiler uses a typed intermediate representation, so any generated code must be annotated with types in order to be run. In principle, these type annotations can be generated along with the code, but the size and impurity of OCaml's type checker stymies this approach. Just to take one example, OCaml annotates each AST node by its type environment, so the code generator would have to apply weakening (along the lines of Kameyama et al. (2008)) whenever splicing code under a binder.

Instead of generating typed code directly, MetaOCaml programs as currently compiled generate untyped code and wait until it is run to invoke the type checker again. This strategy fortuitously avoids Taha and Nielsen's (2003) complicated *demotion* operation and prevents some type unsoundness due to effects (described in §4). However, the MetaOCaml program must keep all the information that the type checker needs to understand the generated code. Serializing this information is challenging—for example, OCaml 3.11 introduced closures in the type checker's environment.

3. Environment persistence needs to be modeled

Algebraic data types illustrate the problem of serializing type information and the gap between theory and practice in typed multistage programming. Take for example the following interaction, in which the identifier `Foo` is redefined. (MetaOCaml crash course: `.< >.` is quasiquote; `.~` is unquote; `.!` is eval.)

```
# type foo = Foo
  let x = .<Foo>.
  type bar = Foo | Bar
  let y = .<Foo>.
  let z = .<(.~x, .~y)>.;;
val z : ('a, foo * bar) code = .<((Foo), (Foo))>.
```

OCaml uses timestamps to distinguish internally between the two definitions of `Foo`. For `z` to be type-checked correctly when it is eventually run, its two occurrences of `Foo` are annotated with *different* type environments—one without `bar` and one with. Essentially, each part of a code value is a closure over the type environment where it was created. This way, even if `z` is run in yet another

type environment, where `Foo`, `foo`, and `bar` are defined differently or not at all, the type information from when `z` was created would persist and be available for run-time code generation.

It may be simpler to implement MetaOCaml by renaming shadowed identifiers rather than serializing type environments. Using this alternative implementation strategy, the code above would produce the value `.<Foo1, Foo2>.`, which can be type-checked at run time in the renamed type environment `type foo = Foo1; type bar = Foo2 | Bar`. We can store this *single* type environment in a `.cmo` file, without worrying about timestamps.

Neither implementation strategy just described is guided by theory, because no multistage calculus today models environments (that is, explicit substitutions). We need such a calculus.

4. Imperative polymorphism redux

The type soundness of basic, pure multistage programming is well understood, even in the presence of polymorphism. In fact, parametric polymorphism—a la the `ST` monad (Launchbury and Peyton Jones 1994)—is used to prevent running generated code with unbound variables (Calcagno et al. 2004; Taha and Nielsen 2003). However, the presence of side effects brings MetaOCaml into uncharted territory.

It is not just a formality for a code generator written in MetaOCaml to type-check the code it just generated before running it. It is useful for the generator to involve code values in effects such as state, exceptions, or control (Bondorf 1992; Danvy and Filinski 1990; Dussart and Thiemann 1996; Kameyama et al. 2009; Lawall and Danvy 1994; Sumii and Kobayashi 2001), but *scope extrusion* may then occur:

```
# let code =
    let x = ref .<1>. in
    let _ = .<fun v -> .~(x := .<v>.; .<()>.)>. in
    !x;;
val code : ('a, int) code = .<v_1>.
# .!code;;
Unbound value v_1
Exception: Trx.TypeCheckingError.
```

More recently, we discovered that imperative *polymorphism* in MetaOCaml makes it possible to generate code that passes the type checker yet dumps core:

```
# .!.<let foo x =
        let t = .~(let r = ref None in .<r>.) in
        match !t with | None   -> t := Some x; x
                      | Some y -> t := Some x; y
      in (foo "xxx", foo true, foo [1,2])>.;;
Segmentation fault (core dumped)
```

We hope (but have yet to prove) that this problem can be prevented by generalizing generated code only when it is pure not only at the future stage but also at the present stage. In general, the interaction between polymorphism and effects in multistage programming is a wide-open problem.

5. Multistage programming as syntactic sugar?

To sidestep OCaml's complex internals, we are tempted to implement multistage programming by turning quotations into AST constructor calls using `camlp4/5`. This way works for generating C code, but generating polymorphic `let` requires a typed representation of polymorphism. That is tricky, especially when there are more than two stages, because the metalanguage seems to need strictly more polymorphism than the object language (Carette et al. 2009; Rendel et al. 2009).

References

Begel, Andrew, Steven McCanne, and Susan L. Graham. 1999. BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture. *SIGCOMM Computer Communication Review* 29(4):123–134.

Bondorf, Anders. 1992. Improving binding times without explicit CPS-conversion. In *Proceedings of the 1992 ACM conference on Lisp and functional programming*, ed. William D. Clinger, vol. V(1) of *Lisp Pointers*, 1–10. New York: ACM Press.

Calcagno, Cristiano, Eugenio Moggi, and Walid Taha. 2004. ML-like inference for classifiers. In *Programming languages and systems: Proceedings of ESOP 2004, 13th European symposium on programming*, ed. David A. Schmidt, 79–93. Lecture Notes in Computer Science 2986, Berlin: Springer.

Calcagno, Cristiano, Walid Taha, Liwen Huang, and Xavier Leroy. 2003. Implementing multi-stage languages using ASTs, gensym, and reflection. In *Proceedings of GPCE 2003: 2nd international conference on generative programming and component engineering*, ed. Frank Pfenning and Yannis Smaragdakis, 57–76. Lecture Notes in Computer Science 2830, Berlin: Springer.

Carette, Jacques, and Oleg Kiselyov. 2008. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. *Science of Computer Programming*. In press.

Carette, Jacques, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming* 19(5):509–543.

Danvy, Olivier, and Andrzej Filinski. 1990. Abstracting control. In *Proceedings of the 1990 ACM conference on Lisp and functional programming*, 151–160. New York: ACM Press.

Dussart, Dirk, and Peter Thiemann. 1996. Imperative functional specialization. Tech. Rep. WSI-96-28, Universität Tübingen.

Frigo, Matteo, and Steven G. Johnson. 2005. The design and implementation of FFTW3. *Proceedings of the IEEE* 93(2):216–231. Special issue on program generation, optimization, and platform adaptation.

Kameyama, Yukiyoshi, Oleg Kiselyov, and Chung-chieh Shan. 2008. Closing the stage: From staged code to typed closures. In *Proceedings of the 2008 ACM SIGPLAN symposium on partial evaluation and semantics-based program manipulation*, ed. Robert Glück and Oege de Moor, 147–157. New York: ACM Press.

———. 2009. Shifting the stage: Staging with delimited control. In *Proceedings of the 2009 ACM SIGPLAN symposium on partial evaluation and semantics-based program manipulation*, ed. Germán Puebla and Germán Vidal, 111–120. New York: ACM Press.

Kamin, Samuel. 1996. Standard ML as a meta-programming language. `http://loome.cs.uiuc.edu/pubs.html`.

Launchbury, John, and Simon L. Peyton Jones. 1994. Lazy functional state threads. In *PLDI '94: Proceedings of the ACM conference on programming language design and implementation*, vol. 29(6) of *ACM SIGPLAN Notices*, 24–35. New York: ACM Press.

Lawall, Julia L., and Olivier Danvy. 1994. Continuation-based partial evaluation. In *Proceedings of the 1994 ACM conference on Lisp and functional programming*, 227–238. New York: ACM Press.

Lengauer, Christian, and Walid Taha, eds. 2006. *Special issue on the 1st MetaOCaml workshop (2004)*, vol. 62(1) of *Science of Computer Programming*. Amsterdam: Elsevier Science.

Rendel, Tillmann, Klaus Ostermann, and Christian Hofer. 2009. Typed self-representation. In *PLDI '09: Proceedings of the ACM conference on programming language design and implementation*, ed. Michael Hind and Amer Diwan. New York: ACM Press.

Sheard, Tim. 2001. Accomplishments and research challenges in meta-programming. In *Proceedings of SAIG 2001: 2nd international workshop on semantics, applications, and implementation of program generation*, ed. Walid Taha, 2–44. Lecture Notes in Computer Science 2196, Berlin: Springer.

Sumii, Eijiro, and Naoki Kobayashi. 2001. A hybrid approach to online and offline partial evaluation. *Higher-Order and Symbolic Computation* 14(2–3):101–142.

Taha, Walid, and Michael Florentin Nielsen. 2003. Environment classifiers. In *POPL '03: Conference record of the annual ACM symposium on principles of programming languages*, 26–37. New York: ACM Press.