MPhil Thesis

Michael Jing Long Lee

April 11, 2025

Contents

Abstract

There are many different ways to manage the interaction between compile-time metaprogramming and effect handlers, but there has been no way to evaluate them against each other. This dissertation introduces such a mechanism, focusing on correctness, expressiveness, and efficiency. Second, we evaluate existing approaches by the aforementioned mechanism, illustrating the trade-offs in the design space. Finally, we introduces a novel approach, a dynamic Best-Effort check, that we show is correct, maximally expressive, and likely more efficient in the common case, as compared to existing checks.

1 Introduction

The promise of an optimising compiler is that the programmer can focus on writing maintainable code, entrusting the responsibility of *optimising* said code to the compiler.

Entrusting optimisation to the compiler is sufficient for many cases, but not all. For reasons that are both theoretical [7] and practical [8], compilers will never be able to identify *all* opportunities for optimisation. What can be done when compiler optimisation is not enough?

Metaprogramming (for example, `C++` templates) [1] is one possible solution. In languages that support metaprogramming, programmers can identify what should be executed at compile-time. Importantly, programmers may write **maintainable** code, which when executed by the compiler, generates **efficient** code. For example, assume two implementations of the same (widely-used) function, one that runs faster on Arm, and one on Intel.

The maintainable way to switch between these functions is via a conditional:

```ocaml
OCaml
let f () = if (cpu_type == Arm) then
             (* return this program *)
           else
             (* return this program *)
f ()
```

However, it might be expensive to determine processor type at run-time. If efficiency is paramount, programmers might have to duplicate their code, doubling maintenance costs. With metaprogramming, however, we may write

```
MacoCaml
(* macro and $ can be read as annotations which
   indicate that the code should be run at compile-time *)
macro m () = if (cpu_type == Arm) then
               (* generate this program, to be run later *)
             else
               (* gen this program, to be run later *)
$(m ())
```

Importantly, the macro `m` is executed by the compiler, *at compile time*. The Arm and Intel variants are *generated from a single specification*, not manually duplicated.

Importantly, to support metaprogramming, languages have to provide the ability to build programs *to be run later*. These "suspended programs" are best thought of as abstract syntax trees. The following code constructs the program $(\lambda x.x)1$.

```ocaml
if (cpu_type == Arm) then
  let build_app (f: (int -> int) ast) (v: int ast) = App(f, v)
  let id_ast: (int -> int) ast = Lam(Var(x), Var(x))
  build_app(id_ast, Int(1)) (* int ast *)
else
  (*...*)
```

Effect handlers are a powerful language construct that can simulate many other language features (state, I/O, greenthreading) [6], and have recently been added to `OCaml` [9]. It is thus strategic, and timely, to investigate the interaction between metaprogramming and effect handlers.

Unfortunately, metaprogramming is known to interact unexpectedly, and poorly, with effect handlers, a problem known as *scope extrusion* [4]. Scope extrusion occurs when the programmer accidentally directs the compiler to generate code with unbound variables. In the following program, we store the variable `Var(x)` into a heap cell (`l`), and retrieve it outside its scope. The program thus evaluates to `Var(x)`, which is unbound.

```ocaml
let l: ast int ref = new(Int(1)) in
let id_ast : ast (int -> int) = Lam(Var(x), l := Var(x); Int(1)) in
!l
```

The problem of scope extrusion has been widely studied, resulting in multitudinous mechanisms for managing the interaction between metaprogramming and effect handlers. Some solutions adapt the type system (Refined Environment Classifiers [5, 3], Closed Types [2]), others insert dynamic checks into the generated code [4]. However, there are two issues.

First, we are a picky lot. Type-based approaches require unfeasible modification of the `OCaml` type-checker, and tend to limit expressiveness, disallowing a wide range of programs that do not lead to scope extrusion. Dynamic solutions are inefficient, reporting scope extrusion long after the error occurred, or unpredictable, allowing some programs, but disallowing morally equivalent programs. Further, many of the dynamic solutions have not been proved correct.

Second, and more importantly, we lacked a way to evaluate solutions fairly and holistically. Our evaluation criteria is three-pronged: correctness, efficiency, and expressiveness. Each solution is described in its own calculus, which made solutions difficult to compare. It is non-trivial to show that the definitions of scope extrusion in differing calculi agree. Hence, one solution may be correct with respect to its definition, but wrong with respect to another. Further, expressiveness and efficiency are inherently comparative criteria.

This thesis will solve both these problems, in decreasing order of priority.

1.1 Contributions

Concretely, our contributions are:

1. A novel "universal" calculus that allows for effect handlers both in the generating and generated code.
2. In the calculus, three precise (but different) definitions of scope extrusion.
3. In the calculus, encodings of the following existing solutions:
    a) Lazy dynamic check (with a proof of correctness)
    b) Eager dynamic check (with a refutation of correctness)
    c) Refined Environment Classifiers
4. A new solution, a Best-Effort dynamic check, that we justify is correct, expressive, and efficient.
5. Implementations of the three dynamic checks in Macocaml.

2 Background

The aim of this dissertation is to evaluate, and propose, policies for mediating the interaction between Macocaml-style compile-time **metaprogramming** and **effect handlers**, by addressing the issue of **scope extrusion**.

In this chapter, I lay the groundwork for the evaluation. I provide technical overviews of each of the three key concepts: metaprogramming (Section 2.1), effect handlers (Section 2.2), and scope extrusion (Section 2.3).

2.1 Metaprogramming

How does one write fast and maintainable code? A naïve answer is “by being a skilled programmer”[1].

2.2 Effect Handlers

2.3 Scope Extrusion

[1]Though not the worst answer –

Bibliography

[1] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004. ISBN 0321227255.

[2] C. Calcagno, E. Moggi, and W. Taha. Closed types as a simple approach to safe imperative multi-stage programming. In U. Montanari, J. D. P. Rolim, and E. Welzl, editors, *Automata, Languages and Programming*, pages 25–36, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-45022-1.

[3] K. Isoda, A. Yokoyama, and Y. Kameyama. Type-safe code generation with algebraic effects and handlers. In *Proceedings of the 23rd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE '24, page 53–65, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400712111. doi: 10.1145/3689484.3690731. URL https://doi.org/10.1145/3689484.3690731.

[4] O. Kiselyov. The design and implementation of ber metaocaml. In M. Codish and E. Sumii, editors, *Functional and Logic Programming*, pages 86–102, Cham, 2014. Springer International Publishing. ISBN 978-3-319-07151-0.

[5] O. Kiselyov, Y. Kameyama, and Y. Sudo. Refined environment classifiers. In A. Igarashi, editor, *Programming Languages and Systems*, pages 271–291, Cham, 2016. Springer International Publishing. ISBN 978-3-319-47958-3.

[6] M. Pretnar. An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic Notes in Theoretical Computer Science*, 319:19–35, 2015. ISSN 1571-0661. doi: https://doi.org/10.1016/j.entcs.2015.12.003. URL https://www.sciencedirect.com/science/article/pii/S1571066115000705. The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI).

[7] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953. ISSN 00029947, 10886850. URL http://www.jstor.org/stable/1990888.

[8] A. D. Robison. Impact of economics on compiler optimization. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande*, JGI '01, page 1–10, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 1581133596. doi: 10.1145/376656.376751. URL `https://doi.org/10.1145/376656.376751`.

[9] K. Sivaramakrishnan, S. Dolan, L. White, T. Kelly, S. Jaffer, and A. Madhavapeddy. Retrofitting effect handlers onto ocaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 206–221, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454039. URL `https://doi.org/10.1145/3453483.3454039`.