MPhil Thesis

Michael Jing Long Lee

April 14, 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 [8] and practical [9], 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
```
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) [7], and have recently been added to `OCaml` [11]. 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* [5]. 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 (`1`), 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 [6, 3], Closed Types [2]), others insert dynamic checks into the generated code [5]. 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 **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

What is Macocaml-style metaprogramming? This section will provide an answer in three steps. First, I motivate metaprogramming, by illustrating the challenge of writing code that is both fast and maintainable (Section 2.1.1). Second, I will consider the design space of metaprogramming (Section 2.1.2). Finally, I will describe the design decisions made by MacoCaml (Section 2.1.3).

2.1.1 Metaprogramming for Fast and Maintainable Code

Metaprogramming helps programmers write fast and maintainable code. How does one write fast and maintainable code? A naïve answer is "by being a skilled programmer"[1]. Programmer skill is insufficient, because maintainability and efficiency are in constant tension.

Consider building a library for neural networks. As part of this library, we need to define function for calculating gradients, `grad`, such that `grad`$fx = f'(x)$. `grad` is needed for performing backpropagation over neural networks.

More precisely, `f` is of type `differentiable`, defined as follows

```OCaml
type differentiable = Sin | Tanh | Sigmoid | ...
                    | Polynomial of float list
                    | Compose of differentiable * differentiable
```

For example, the following expression represents $\sin \circ \tanh(x)$.

```OCaml
Compose(Tanh, Sin)
```

[1] Though not the worst answer – "use ChatGPT"

Assume further a function, `grad_of`, that returns the gradient of the base constructors, for example, `grad_of Sin 0 = cos 0 = 1`.

How would one write `grad` *maintainably*? How would one write it *efficiently*?

To write this maintainably, we might write a function as such:

```OCaml
let rec grad f x =
  match f with
  | Sin
  | Tanh
  | Sigmoid
  | ...
  | Polynomial(cs) -> grad_of f x
  | Compose(f, g) -> (grad f x) * (grad g (app f x))
```

This function is maintainable: it works for any differentiable function. However, it is not as efficient as it could be: most obviously, walking the function and performing a `match` on every recursive call might result in expensive conditional branches. If we assume `x` is a vector, and the weights of the polynomial are vectors, then this representation could limit opportunities for cache prefetching.

If the function is known in advance, for example, `f = Compose(Tanh, Sin)`, we could replace the `grad` function with a cheap hardcoded equation.

```OCaml
let grad_fast x = (cos x) /. (cosh (sin x) ** 2)
```

The example illustrates the trade-off between maintainability and efficiency. In writing maintainable code, we sought to parameterise over the function. Abstraction is the key to maintainability. However, the aforementioned optimisations rely on specialisation, assuming **a** function. More generally, many compiler optimisations, like monomorphisation, eliminate abstraction, simplifying functions by applying known arguments in advance.

The tension between efficiency and maintainability has also been observed in regex matching [12], parsing [15], linking [10], statistical modelling [14], and hardware design [13].

A more informed answer might therefore be "by letting the compiler optimise my maintainable code". Not quite – for reasons both theoretical and practical, compiler optimisations can be insufficient. In theory, we proposed an optimisation that assumed we would always know the function at, or before, compile-time. Is this a reasonable assumption? It is: we assumed that we were using `grad` to perform backpropagation over neural networks. The network over which backpropagation is performed is known at compile-time. However, notice that this justification appeals to domain-specific knowledge regarding how `grad` will be used. In the general case, `grad` could be applied to a function not known until runtime. It is not feasible to expect a compiler

to spot all opportunities for optimisation. In practice, while compiler engineers might have an economic incentive to write optimisations for the machine learning community, this may not be true for less lucrative domains [9]. Even in machine learning, many libraries are built on top of existing languages, like `Python`, which might not perform the desired optimisations.

How does one write maintainable and efficient code, **when one cannot trust the compiler to optimise one's code**?

One answer is metaprogramming – annotations that explicitly identify optimisation opportunities to the compiler, instructing it to apply certain arguments at compile-time. The `grad` function in JAX, a Python-based machine learning framework, uses metaprogramming for precisely this purpose [4].

Speeding up exponentiation with Metaprogramming

The `grad` example was useful for motivating metaprogramming, but not for understanding metaprogramming. To explain how metaprogramming works, we switch to a morally equivalent, but simpler example: raising an integer `x` to an exponent `n`.

A maintainable exponentiation function may be written as follows

```OCaml
let rec pow n x = if n == 0 then 1 else x * pow (n-1) x
```

`pow`, which can be applied to any exponent `n`, is analogous to the `grad` function, which could be applied to *any* differentiable function.

However, should we know the exponent in advance, for example $n = 2$, then a more efficient, but less maintainable function, is

```OCaml
let square x = x * x
```

`square` is analogous to the `grad_fast` expression in the machine learning example.

Metaprogramming can be utilised to write a function that resembles `pow`, inheriting its maintainability, but that generates `square`, inheriting its efficiency. I will now present the meta-programmed `pow` function, and explain how it generates `square`.

```MacoCaml
macro rec powGen (n: int) (x: int expr) =
  if n == 0 then <<1>>
  else <<$x * $(pow (n-1) x)>>

let square y = $(powGen 2 <<y>>)
```

Metaprogramming allows the user to create, and manipulate, abstract syntax trees (AST). This allows the programmer to generate programs as data,

by constructing the AST of the program. For example, consider the program `1 * 2`. Assume every code construct (`1`, `2`, `*`) has a corresponding AST constructor (`Int(1)`, `Int(2)`, `Mul(x, y)`).

We can create the AST of the program by writing `Mul(Int(1), Int(2))`. We may also manipulate ASTs, for example, we could write

```
MacoCaml
let add_zero (x: int expr) = Plus(x, Int(0))
add_zero Mul(Int(1), Int(2)) (* Plus(Mul(Int(1), Int(2)), Int(0)) *)
```

With compile-time metaprogramming, programmers can write functions that, when executed at compile-time, generate efficient functions for run-time. Take the `pow` function. We assumed `n` is known, but `x` is not. Therefore, the aim is to build a generator, that, when applied to $n = 2$ at compile-time, builds the AST of `square`. This is easily achieved by replacing `1` and `*` with their corresponding AST constructors. Note also that `x` is an AST of an integer, rather than an integer.

```
OCaml
let rec powGen (n: int) (x: int expr) =
  if n == 0 then Int(1)
  else Mul(x, pow (n-1) x)
```

`powGen` may now be used as follows

```
OCaml
LetFun(Var(square), Lam(Var(y), powGen 2 Var(y)))
```

This expression evaluates to

```
OCaml
LetFun(Var(square), Lam(Var(y), Mul(Var(y), Mul(Var(y), Int(1)))))
```

Which is the abstract syntax tree of `let square y = y * y * 1`.

This AST-constructor style is cumbersome, requiring extensive re-writes. For ease of both writing and reading, MacoCaml introduces two language constructs, `<<e>>` (quote) and `$e` (splice). Quote converts expressions into their AST form, for example,

```
<<1*2>> = Mul(Int(1), Int(2))
```

While splice converts ASTs into expressions, and is therefore sometimes called "anti-quotation"

```
$(Mul(Int(1), Int(2))) = <<1*2>>
```

Additionally, since MacoCaml performs code generation at compile-time, all expressions are treated as AST constructors by default. Re-writing `pow` in this style, one obtains

```
MacoCaml
$(let rec powGen n x =
    if n == 0 then <<1>>
    else <<$x * $(pow (n-1) x)>>
  in
  <<let square y = $(powGen 2 <<y>>)>>)
```

To allow powGen to be re-used globally, much the same way as a top-level **let**, MacoCaml introduces the **macro** keyword. Rewriting in this style, we obtain the metaprogrammed powGen function introduced at the start of the section.

2.1.2 The Design of Metalanguages

While all metaprogramming languages allow for the creation and manipulation of abstract syntax, the specific mechanisms for doing so can be markedly different. In this section, I taxonomise the space of metalanguages, by highlighting key design decisions.

Homogenous vs Heterogenous

One key design decision is whether the generated language (also known as the object language), and the generating language (the meta language) coincide. If the object and meta languages are the same, this is known as **homogenous** metaprogramming. Otherwise, it is **heterogenous**.

In JAX, Python (the meta language) generates MLIR (the object language), and this is thus heterogenous metaprogramming. C++ templates use C++ (the meta language) to generate C++ (the object language), and this is homogenous.

Run-Time vs Compile-Time

Two-Stage vs Multi-Stage

2.1.3 MacoCaml

2.2 Effect Handlers

Effect handlers

2.2.1 Effect Handlers for Composable User-Defined Effects

2.3 Scope Extrusion

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] Jax-Ml. Using grad on vmap on map on function containing sinc results in error. URL https://github.com/jax-ml/jax/issues/10750.

[5] 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.

[6] 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.

[7] 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).

[8] 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.

[9] 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.

[10] M. Servetto and E. Zucca. A meta-circular language for active libraries. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation*, PEPM '13, page 117–126, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450318426. doi: 10.1145/2426890.2426913. URL https://doi.org/10.1145/2426890.2426913.

[11] 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.

[12] L. Tratt. Domain specific language implementation via compile-time meta-programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(6):1–40, 2008.

[13] J. Vandebon, J. G. F. Coutinho, W. Luk, and E. Nurvitadhi. Enhancing high-level synthesis using a meta-programming approach. *IEEE Transactions on Computers*, 70(12):2043–2055, 2021. doi: 10.1109/TC.2021.3096429.

[14] H. Wickham. *Advanced R*. Chapman and Hall/CRC, 2019.

[15] J. Yallop, N. Xie, and N. Krishnaswami. flap: A deterministic parser with fused lexing. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023. doi: 10.1145/3591269. URL https://doi.org/10.1145/3591269.