MPhil Thesis

Michael Jing Long Lee

May 13, 2025

Contents

1 Introduction **1**
1.1 Contributions . . . 2

2 Background **4**
2.1 Metaprogramming . . . 4
2.1.1 Metaprogramming for Fast and Maintainable Code . . . 4
2.1.2 The Design Space of Metalanguages . . . 8
2.2 Effect Handlers . . . 9
2.2.1 Composable and Customisable Effects . . . 9
2.2.2 λ_{op}: A Calculus for Effect Handlers . . . 11
2.2.3 The Design Space of Effect Handlers . . . 20
2.3 Scope Extrusion . . . 21
2.3.1 Existing Solutions to the Scope Extrusion Problem . . . 22

3 Calculus **28**
3.1 The source language: $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ . . . 28
3.1.1 Type System . . . 28
3.2 The core language: $\lambda_{\mathsf{AST(op)}}$ . . . 33
3.2.1 Operational Semantics . . . 37
3.2.2 Type System . . . 40
3.3 Elaboration from $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ to $\lambda_{\mathsf{AST(op)}}$ . . . 41
3.3.1 Elaborating Types . . . 41
3.3.2 Elaborating Contexts . . . 42
3.3.3 Elaborating Terms . . . 42
3.3.4 Elaborating Typing Judgements . . . 42
3.4 Metatheory . . . 43

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 [18] and practical [19], 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 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.

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) expr) (v: int expr) = App(f, v)
  let id_ast: (int -> int) expr = Lam(Var(x), Var(x))
  build_app(id_ast, Int(1)) (* int expr *)
else
  (*...*)
```

Effect handlers are a powerful language construct that can simulate many other language features (state, I/O, greenthreading) [17], and have recently been added to `OCaml` [21]. 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* [11]. 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: int expr ref = new(Int(1)) in
let id_ast : (int -> int) expr = 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 [13, 8], Closed Types [3]), others insert dynamic checks into the generated code [11]. 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.

The scope of this dissertation thus extends to effect systems with **deep** (but **unnamed**) handlers and **multi-shot continuations**.

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 provide technical overviews of each of the 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? I will provide an answer in two 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), highlighting decisions made by MacoCaml.

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.

I illustrate this tension by considering a concrete problem. Consider computing the gradient of a differentiable function as part of backpropagation over a neural network. More precisely, assume f of type `differentiable`

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

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

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

We wish to write a function `grad` such that `grad` $f = f'$. For simplicity, assume the existence of a helper function, `grad_of`, that returns the gradient of basic functions. For example, `grad_of Sin 0.0` $=$ `cos 0.0` $=$ `1.0`. `grad_main` (Listing 1) illustrates one way to compute gradients maintainably:

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

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

Listing 1: A maintainable implementation of `grad`

However, `grad_main` may not be the most efficient implementation. Performing a `match` on every recursive call might result in expensive branches. If `x` is a vector, and the weights of a polynomial are vectors, then `grad_main` could hide opportunities for cache prefetching.

If f is known in advance, for example, $f =$ `Compose(Tanh, Sin)`, we could implement a more efficient `grad_fast` function (Listing 2), whose body is simply a hardcoded equation:

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

Listing 2: A fast implementation of `grad`, assuming $f =$ `Compose(Tanh, Sin)`

Although `grad_fast` only works for a single f, it has eliminated the branching overhead, and enabled opportunities for prefetching. It is thus likely to be faster.

The `grad` example illustrates the trade-off between maintainability and efficiency. `grad_main` is maintainable in part because it parameterises over f. More generally, abstraction centralises implementations, thus reducing maintenance costs. However, `grad_fast` is more efficient because it is more specialised to a specific f (`Compose(Tanh, Sin)`). More generally, many compiler optimisations, like monomorphisation, eliminate abstraction, simplifying functions by applying known arguments in advance.

The tension between maintainability (abstraction) and efficiency (specialisation) has also been observed in regex matching [22], parsing [27], linking [20], statistical modelling [24], and hardware design [23].

A more informed answer might therefore be "by letting the compiler generate optimised versions of 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 f at, or before, compile-time. Is this a reasonable assumption? It is: we assumed that `grad` would 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 [18]. 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 [19]. 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, which gives users the ability to perform code-generation. Programmers may thus take matters into their own hands: manually generating optimised code when the compiler may not automatically do so for them. The `grad` function in `JAX`, a Python-based machine learning framework, uses metaprogramming for precisely this purpose [9].

Speeding up exponentiation with Metaprogramming

Metaprogramming allows for code-generation via the creation and manipulation of abstract syntax trees (ASTs). I will now illustrate how metaprogramming works with reference to `MacoCaml`, which implements *compile-time* metaprogramming. While the `grad` example motivated metaprogramming, for pedagogical reasons, I switch to a morally equivalent, but simpler example: raising an integer `x` to an exponent `n`. One maintainable implementation is the `pow` function (Listing 3):

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

Listing 3: A maintainable implementation of an exponentiation function

`pow`, which can be applied to any exponent `n`, is analogous to `grad_main` (Listing 1), which could be applied to *any* differentiable function f.

However, should we know the exponent in advance, for example `n` $= 2$, then a more efficient, but less maintainable implementation, is the `square` function (Listing 4)

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

Listing 4: An efficient implementation of exponentiation, assuming `n` $= 2$

`square` is analogous to `grad_fast` (Listing 2).

Metaprogramming can be utilised to write a function, `pow_gen`, which resembles `pow` (inheriting its maintainability), but that generates a program which resembles `square` (inheriting its efficiency). Listing 5 presents the meta-programmed `pow_gen` function. Compilation generates the body of `square y` (line 4), `y * y * 1`. I will now explain the mechanics of generation.

```ocaml
OCaml
1 macro rec pow_gen (n: int) (x: int expr) =
2   if n == 0 then <<1>>
3   else <<$x * $(pow_gen (n-1) x)>>
4 let square y = $(pow_gen 2 <<y>>) (*after compile-time: y * y * 1 *)
5 square 3 (*at runtime: 9*)
```

Listing 5: A meta-programmed `pow_gen` function, which resembles `pow` but generates `square`

Recall that (compile-time) metaprogramming gives the programmer the ability to generate programs at compile-time, for use at run-time. We may build this in two steps, by:

1. Deciding on a representation for code values, such that code can be created, and manipulated by programs. Once we have a representation for code values, it is possible to write expressions that return code values. These expressions serve as program generators.

2. Building a mechanism for executing expressions *at compile-time*. We can constrain this mechanism, using types, so only generators can be executed at compile-time.

First, we represent code values as ASTs. Generated programs are ASTs, and program generators are expressions that evaluate to ASTs. We can assume, for clarity, that the language offers, for each program construct, a corresponding AST node: for example, the integer `1` has AST node `Int(1)`. If a program has type `'a`, then its AST node has type `'a expr`. One can now write program generators, that evaluate to ASTs, for example:

MacoCaml

```
let rec pow_gen (n: int) (x: int expr) =
  if n == 0 then Int(1)
  else Mul(x, (pow_gen (n-1) x))
pow_gen 2 Int(3) (*Mul(Int(3), Mul(Int(3), 1))*)
pow_gen 2 Var(y) (*Mul(Var(y), Mul(Var(y), 1))*)
pow_gen 3 Var(y) (*Mul(Var(y), Mul(Var(y), Mul(Var(y), 1)))*)
```

Second, we need a mechanism to execute expressions at compile-time. In `MacoCaml`, this is the "top-level splice", a splice (`$`) annotation not surrounded by quotes (`<<>>`). For example, in Listing 5, there is only one top-level splice, on line 4: `$(pow_gen 2 <<y>>)`. We may now shift program generators (and only program generators) under top-level splices, to perform generation at compile time. Note that to access `pow_gen` at compile-time, we must also move it under the top-level splice.

MacoCaml

```
let square y = $(let rec pow_gen (n: int) (x: int expr) = ...
                   in pow_gen 2 Var(y))
                 (*Mul(Var(y), Mul(Var(y), 1))*)
let cube y   = $(let rec pow_gen (n: int) (x: int expr) = ...
                   in pow_gen 3 Var(y))
                 (*Mul(Var(y), Mul(Var(y), Mul(Var(y), 1)))*)
```

To allow compile-time functions, like `pow_gen`, to be re-used across multiple top-level splices, `MacoCaml` introduces the **`macro`** (Listing 6)

OCaml

```
macro rec pow_gen (n: int) (x: int expr) =
  if n == 0 then Int(1)
  else Mul(x, (pow_gen (n-1) x))
let square y = $(pow_gen 2 Var(y)) (*Mul(Var(y), Mul(Var(y), 1))*)
let cube y = $(pow_gen 3 Var(y)) (*Mul(Var(y), Mul(Var(y), Mul(Var(y), 1)))*)
```

Listing 6: In `MacoCaml`, **`macro`** allows for definitions to be shared across top-level splices

Further, rather than explicit AST constructors, ASTs are created by the `<<>>` ("quote") and `$` annotations. Quotation creates ASTs, by converting a program into its AST representation. For example,

`<< $x + 0 >>` = `Plus(Var(x), Int(0))`

Under a quotation, the `$` annotation stops this conversion, allowing for programs that *manipulate* ASTs.

`<< $x + 0 >>` = `Plus(x, Int(0))`

In `MacoCaml`, the programmer interleaves quotes and splices to perform code generation

`<< $(add_zero <<1>>) + 0 >>` = `Plus(add_zero Int(1), Int(0))`

Notice that `$` is overloaded. We must be careful to disambiguate between "top-level splices", which execute programs at compile-time, and splices under quotations, which stop conversion to AST.

Re-writing Listing 6 in this style (being careful about non-top-level splices), we obtain exactly Listing 5.

Applying this technique to the `grad` example, we obtain

MacoCaml

```
macro rec grad_gen f x = match f with
  | Sin
  | Tanh
  | Sigmoid
  | ...
  | Polynomial(_) -> grad_of f x
  | Compose(f, g) -> <<$(grad_gen f x) * $(grad_gen g (app f x))>>
let grad_fast x = $(grad_gen Compose(Tanh, Sin) <<x>>)
```

where `grad_of` and `app` are appropriately modified.

2.1.2 The Design Space of Metalanguages

Different metalanguages provide slightly different variants of metaprogramming to the user. In this section, I broadly taxonomise these languages by considering three key design decisions:

1. **Homogenous or Heterogenous**

   Do the generated ("object") and generating ("meta") languages agree or differ?

   If the object and meta languages are the same, this is known as homogenous metaprogramming. Otherwise, it is heterogenous [12].

   `MacoCaml` allows for homogenous metaprogramming, where `OCaml` code generates `OCaml` code. In contrast, `MetaHaskell` [15] programs generate `C` code, allowing for heterogenous metaprogramming.

2. **Run-time or Compile-Time**

   When does the generation take place?

   Code generation could take place at compile-time (as with `MacoCaml` programs or `C` macros), or at run-time (as with `MetaOCaml` [11]).

   Run-time and compile-time metaprogramming differ non-trivially. Run-time metaprogramming requires a language construct (`!`, or "run") for explicit invocation of the compiler. Further, in run-time metaprogramming, generated and generating programs may share a heap.

   `MacoCaml` supports compile-time metaprogramming, and we will pay no further attention to run-time metaprogramming.

3. **Two-stage or Multi-stage**
   How many stages of code generation are allowed?

   When introducing `MacoCaml`, I illustrated how one uses top-level splices to perform shift computation from run-time ("level 0") to compile-time ("level -1"). Might it be possible to shift computation from compile-time to a pre-compile-time ("level -2" phase), for example, via a nested splice?

   ```
   $($ pow_gen 2 Var(y))
   ```
   MacoCaml

   In a two-stage system, one is restricted to operating between two levels, so this is disallowed. In contrast, in a multi-stage system, one can operate between any number of levels. Multi-stage metaprogramming is thus strictly more general than two-stage metaprogramming.

   Although nested splices are disallowed in `MacoCaml`, it is a multi-stage system, since entire modules may be imported at a decremented level [26].

`MacoCaml` offers homogenous, compile-time, multi-stage metaprogramming. The scope of this dissertation is slightly more restrictive: I focus on two-stage, not multi-stage metaprogramming. This restriction was motivated by a cost-benefit analysis:

1. **Cost**: Since in `MacoCaml`, the module system is the only mechanism for achieving multi-stage programming, investigating multi-stage metaprogramming would require the investigation of module systems, effects, and metaprogramming. The interaction between module systems and metaprogramming is still an ongoing area of research [4].

2. **Benefit**: In practice, "almost all uses" of multi-stage metaprogramming only use two stages [7]. Further, scope extrusion can be observed, and is often studied, in two-stage systems [8, 13].

2.2 Effect Handlers

What is an effect handler? I will first motivate effect handlers by considering the problem of adding resumable exceptions to `OCaml` (Section 2.2.1). Second, I will introduce a calculus for studying the operational behaviour of effect handlers, à lá Pretnar [17] (Section 2.2.2). This calculus will be useful both for precise description of effect handlers, and as a basis for investigating the interaction between metaprogramming and effect handlers (once the calculus has been extended with metaprogramming facilities). Finally, since different design decisions for effect handlers could affect the nature of their interaction with metaprogramming, I will consider the design space of effect handlers (Section 2.2.3).

2.2.1 Composable and Customisable Effects

Effects are a mechanism by which a program interacts with its environment. Examples of effects include state, (resumable) exceptions, non-determinism, and I/O. Effects are typically defined and understood separately, meaning they are not easily composable. They are also implemented by compiler engineers rather than programmers, meaning they are not customisable. Effect handlers provide a programmable, unifying framework that may be instantiated into different effects. This allows for composable and customisable treatment of effects.

To illustrate the need for effect handlers, consider the following problem, by Kiselyov [10]. Assume a binary search tree of (key, value) pairs. The following code provides two functions. The first `find`s a value `v` associated with key `k`, raising a `NotFound` exception if `k` is not in the tree. The second `update`s the dictionary with a fresh key value pair, overwriting old values.

```ocaml
type ('a, 'b) tree = Lf | Br of 'a * 'b * tree * tree

let rec find (t: tree) (k: 'a) = match t with
 | Lf -> raise NotFound()
 | Br(k', v, l, r) -> if k == k' then v
                      else if k < k' then find l k
                           else find r k

let rec update (t: tree) (k: 'a) (v: 'b) = match t with
  | Lf -> Br(k, v, Lf, Lf)
  | Br(k', v', l, r) -> if k == k' then Br(k, v, l, r)
                       else if k < k' then Br(k', v', update l k v, r)
                            else Br(k', v', l, update r k v)
```

Assume the task is to build a `findOrInsert` function that either finds the value associated with a key, *or* inserts a default value. A naïve approach to writing this function would be

```ocaml
let rec findOrInsert (t: tree) (k: 'a) (default: 'b) =
  try find t k with NotFound -> insert t k default
```

This function is **inefficient**. If a `NotFound` exception is raised, then the `find` function will have raised the exception at the point where the default value should be inserted. The function could be twice as efficient if the exception could be resumed at the point where the exception was raised, in the following style.

```ocaml
let rec findOrInsert (t: tree) (k: 'a) (default: 'b) =
  try find t k with NotFound(p) -> continue p Br(k, default, Lf, Lf)
```

`p` represents the suspended program to be resumed, and is known as a *delimited continuation*.

The aforementioned problem motivates the need for resumable exceptions. To understand the need for effect handlers, consider how one might go about **implementing** resumable exceptions. One approach might be to fork the implementation of handlers and tweak it ever-so-slightly. This solution does not scale well. First, the solution may not be **composable**. The intended informal semantics for resumable exceptions is “effectively equivalent to exceptions, with the additional power to resume programs”. Resumable exceptions should thus interact with other exceptions in a predictable way, but this is difficult to guarantee, and *continually* guarantee, especially as implementations evolve, and more variants of exceptions are demanded. Second, the solution is not **customisable**. To add resumable exceptions requires a compiler engineer to modify the compiler. With the exception of raising an issue, there is nothing the programmer may do, in the moment, to meet their need.

Effect handlers resolve both composability and customisability issues. Much like how exception handlers allow users to create custom exceptions with custom semantics, effect handlers provide a general framework for creating custom effects with custom semantics. The interaction between effect handlers is described abstractly, parameterising over the exact semantics of the effect. Hence, implementing effects (in the earlier example, resumable exceptions, but more generally, state, I/O, greenthreading, non-determinism, and more) as effect handlers ensures composability by design.

With effect handlers, we can re-write the previous example to obtain the behaviour of resumable exceptions, even if the `OCaml` compiler does not support it, with the guarantee that `NotFound` will interact predictably with other defined effects.

OCaml

```ocaml
type _ Effect.t += NotFound: unit -> tree t

let rec find (t: tree) (k: 'a) = match t with
  | Lf -> NotFound()
  | Br(k', v, l, r) -> if k == k' then v
                       else if k < k' then find l k
                             else find r k

let rec findOrInsert (t: tree) (k: 'a) (default: 'b) =
  match find t k with NotFound(p) with
  | v -> v
  | effect NotFound k -> continue p Br(k, default, Lf, Lf)
```

Since effect handlers may be instantiated into a range of different effects, considering the interaction of metaprogramming with effect handlers is an exercise in killing many birds with a single stone. Additionally, effect handlers were recently added to `OCaml` [21], making their interaction a timely problem.

2.2.2 λ_{op}: A Calculus for Effect Handlers

Having motivated effect handlers, I will now describe a calculus, which I call λ_{op}, for reasoning about their operational behaviour. λ_{op} is a slight variant of the calculus described by Pretnar [17]. Understanding λ_{op} will be useful for two reasons. First, a precise description of the operational behaviour of effects will aid reasoning about their interaction with metaprogramming. Second, my universal calculus will be described by extending λ_{op}. Throughout this section, I will use the λ_{op} program in Listing 7 as a running example.

λ_{op}

```
handle
  do x ← print(1); return 1 in do y ← print(2); return 2 in x + y
with
  {return(x) ↦ return (x, "");
   print(x, k) ↦ do (v, s) ← continue k () in return (v, f"{x};" ^ s)}
--------------------------------------------------------------------
return (3, "1;2")
```

λ_{op}

Syntax

Values	v	$:=$	$x \mid n \mid \lambda x.c \mid \kappa x.c$
Computations	c	$:=$	$v_1 v_2 \mid \texttt{return}\ v \mid \texttt{do}\ x \leftarrow c_1\ \texttt{in}\ c_2$
			$\mid \texttt{op}(v) \mid \texttt{handle}\ c\ \texttt{with}\ \{h\} \mid \texttt{continue}\ v_1\ v_2$
Handlers	h	$:=$	$\texttt{return}(x) \mapsto c \mid h; \texttt{op}(x,k) \mapsto c$

Figure 2.1: The syntax of λ_{op}. Terms are syntactically divided into values v, computations c, and handlers h

Listing 7: An λ_{op} program that returns $(3, \texttt{"1;2"})$. It will be used as a running example throughout this section.

Figure 2.1 collates the base syntax of λ_{op}. In addition to this base syntax, in this section, I will assume λ_{op} is extended with the following language extensions: a unit value $()$, pairs $(1, 2)$ which can be destructured $\texttt{do}\ (x, y) \leftarrow \texttt{return}\ (1, 2)\ \texttt{in}\ x + y$, strings `"Hello"`, format strings `f"{1}"`, and string concatenation `^`. For example, the following code evaluates to `"Revolution 9"`.

λ_{op}

$\texttt{do}\ (x, y) \leftarrow \texttt{return}\ (\texttt{"Revolution"}, \texttt{f"\{9\}"})\ \texttt{in}\ x \mathbin{\hat{}} y$

$\texttt{return "Revolution 9"}$

Further, I use $c_1; c_2$ as syntactic sugar for $\texttt{do}\ \_ \leftarrow c_1\ \texttt{in}\ c_2$. I will explain key language constructs in turn.

Sequencing computations: `do` and `return`

Effects force us to carefully consider the order of evaluation. For example, consider the following `OCaml` programs

OCaml

```
let pure      = (1+0) + (2+0)
let effectful = let l = new 0 in (l := 1; 1) + (l := 2; 2)
```

The result of `pure`, which has no effects, is independent of the evaluation order. In contrast, the result of `effectful` *is* dependent on the evaluation order. If terms are evaluated left-to-right, the value of `!l` is `2`, otherwise, it is `1`.

In order to be precise about the order of evaluation, λ_{op} terms are stratified into two syntactic categories, "inert values" v and "potentially effectful computations" c [17]. $\texttt{return}\ v$ lifts values into computations, and is also the result of fully evaluating a computation. $\texttt{do}\ x \leftarrow c_1\ \texttt{in}\ c_2$ sequences computations, forcing programmers to be explicit the order of evaluation. First, c_1 is fully evaluated to obtain some $\texttt{return}\ v$. The value v is then bound to x, and finally c_2 is evaluated.

For example, extending λ_{op} with a `plus` function, what is the order of evaluation of `plus 1 2`? Do we evaluate both arguments before applying them, or interleave evaluation and application? The syntax forces programmers to choose explicitly. We can either fully evaluate both arguments before applying them in turn,

λ_{op}
$$\texttt{do}\ x \leftarrow \texttt{return}\ 1\ \texttt{in}\,(\texttt{do}\ y \leftarrow \texttt{return}\ 2\ \texttt{in}\,(\texttt{do}\ f \leftarrow \texttt{plus}\,x\ \texttt{in}\ f y))$$

or alternatively, evaluate 1, apply it, then evaluate 2

λ_{op}
$$\texttt{do}\ x \leftarrow \texttt{return}\ 1\ \texttt{in}\,(\texttt{do}\ f \leftarrow \texttt{plus}\,x\ \texttt{in}\,\texttt{do}\ y \leftarrow \texttt{return}\ 2\ \texttt{in}\ f y)$$

Both choices are valid, but the programmer must choose. For clarity, where the ordering cannot affect the result (both of the aforementioned choices evaluate to **return** 3), I will abuse notation and write (for instance) $1 + 2$.

Performing effects: op, handle, and continue

Having made explicit the order of operation, we may now add effect handlers. Recall that effect handlers allow users to register custom effects with custom semantics. I will now illustrate how this is supported by λ_{op}.

For simplicity, λ_{op} assumes that the effects have been registered in advanced, parameterising over them with the placeholder **op**(v). Assume that the user has declared the effects **print** and **read_int** in advance. This would allow the user to write programs like

λ_{op}
$$\texttt{do}\ x \leftarrow \texttt{print}(1);\texttt{return}\ 1\ \texttt{in}\,\texttt{do}\ y \leftarrow \texttt{print}(2);\texttt{return}\ 2\ \texttt{in}\ x + y$$

In the program fragment above, we know that **print** is an effect, but we do not know its semantics. Effect handlers, which comprise a **return handler** and zero or more **operation handlers**, specify how effects interact with their environment, and thus may be used to give effects meaning. I will define an effect handler that accumulates print statements in a string (some "`stdout`"). For example, the aforementioned program should return (3, `"1; 2"`).

We begin by considering how to handle the case where there are no calls to **print**. For example, in the program **return** 3. We may wish to return both the value, and the empty string (empty `stdout`) to the environment: in this case, (3, `""`). We can achieve this by specifying a *return handler*.

$$\texttt{return}(x) \mapsto c$$

In this case, we set c to **return** $(x, \texttt{""})$. All effect handlers must specify a return handler. In many cases, the return handler is simply the identity (c is set to **return** x): for brevity and clarity, if the return handler is the identity, I may drop it.

Next, we consider how to handle a call to **print**. We use an operation handler of the form

$$\texttt{print}(x, k) \mapsto c$$

Where c is the user-defined semantics for **print**. Concretely, one instance of c is

$$\texttt{print}(x, k) \mapsto \texttt{do}\ (v, s) \leftarrow \texttt{continue}\,k\,()\ \texttt{in}\,\texttt{return}\ (v, \texttt{f"}\{x\}\texttt{;"}\ \hat{}\ s)$$

In the definition of c, the programmer may refer to x and k, which I will now explain. x allows programs to send values (for example, values to be printed) to their environment. k is a delimited continuation representing a suspended program, awaiting a value from the environment. Effects also allow programs to receive data from their environment, as in

$$1 + \texttt{get\_int\_from\_user}()$$

Note that the program is suspended until the value is receive. We may write the suspended program as $1 + [-]$, where $[-]$ indicates an as-yet-unknown value. This suspended program is represented by the continuation k. The expression

$$\texttt{continue}\ k\ v$$

is used to resume the suspended program with value v.

We are now able to interpret the concrete operation handler c: we resume the suspended program, supplying a unit value, since **print** effects do not receive values from their environment. This returns a value v and some partially accumulated `stdout` s. We prepend the printed value, x, onto s.

Having defined the semantics for **print**, the user may now interpret the earlier example with their semantics, using the **handle** e **with** $\{h\}$ construct. Doing so results in the program in Listing 7.

Notice that multiple effects may be handled by the same handler, and the same effect might be handled by multiple handlers, potentially with different semantics.

Operational Semantics

Having described informally the desired semantics of λ_{op}, we may now make our intuitions precise, by means of an operational semantics. The operational semantics is collated in Figure 2.2.

The operational semantics is given on configurations of the form $\langle c, E \rangle$, where c is a term and E is an evaluation context, in the style of Felleisen and Friedman [5]. Evaluation contexts are represented as a stack of evaluation frames F, à lá Kiselyov [10]. Most of the rules are standard. We will focus on two rules: Eff-Op, the mechanism for giving effects custom semantics, and Eff-Cnt, the mechanism for resuming programs.

To illustrate the operation of Eff-Op and Eff-Cnt, consider the evaluation of the running example in Listing 7, beginning with an empty context. Let h be the handler body

$$\begin{aligned}&\{\texttt{return}(x) \mapsto \texttt{return}\ (x, \texttt{""});\\ &\ \texttt{print}(x, k) \mapsto \texttt{do}\ (v, s) \leftarrow \texttt{continue}\ k\ ()\ \texttt{in}\ \texttt{return}\ (v, \texttt{f"}\{x\}\texttt{;"}\ \hat{}\ s)\}\end{aligned}$$

After several applications of Cng-Psh, we obtain the configuration

$$\begin{aligned}\langle\, &\texttt{print(1)}\ \ ;\ \ \texttt{handle}\\ &\qquad\qquad \texttt{do}\ x \leftarrow [-]; \texttt{return}\ 1\ \texttt{in}\ \texttt{do}\ y \leftarrow \texttt{print(2)}; \texttt{return}\ 2\ \texttt{in}\ x + y\\ &\qquad\quad \texttt{with}\ \{h\}\,\rangle\end{aligned}$$

Let $E = \texttt{do}\ x \leftarrow \texttt{return}\ u; \texttt{return}\ 1\ \texttt{in}\ \texttt{do}\ y \leftarrow \texttt{print(2)}; \texttt{return}\ 2\ \texttt{in}\ x + y$. Applying Eff-Op, we can suspend the program, find the handler h with the user's semantics for **print**, and give the **print** effect the desired semantics

$$\langle\, \texttt{do}\ (v, s) \leftarrow \texttt{continue}\ (\kappa u.\ \texttt{handle}\ E\ \texttt{with}\ \{h\})\ ()\ \texttt{in}\ \texttt{return}\ (v, \texttt{f"}\{1\}\texttt{;"}\ \hat{}\ s)\ \ ;\ \ [-]\rangle$$

Applying Cng-Psh,

$$\langle\, \texttt{continue}\ (\kappa u.\ \texttt{handle}\ E\ \texttt{with}\ \{h\})\ ()\ \ ;\ \ \texttt{do}\ (v, s) \leftarrow [-]\ \texttt{in}\ \texttt{return}\ (v, \texttt{f"}\{1\}\texttt{;"}\ \hat{}\ s)\,\rangle$$

Operational Semantics

λ_{op}

Auxiliary Definitions

Evaluation Frame	F	::=	$\texttt{do}\ x \leftarrow [-]\ \texttt{in}\ c_2 \mid \texttt{handle}\ [-]\ \texttt{with}\ \{h\}$
Evaluation Context	E	::=	$[-] \mid E[F]$

Domain of Handler	$\mathsf{dom}(h)$	$\triangleq$	$\mathsf{dom}(\texttt{return}(x) \mapsto c) = \emptyset$, $\mathsf{dom}(h; \texttt{op}(x,k) \mapsto c) = \mathsf{dom}(h) \cup \{\texttt{op}\}$
Handled Effects	$\mathsf{handled}(E)$	$\triangleq$	$\mathsf{handled}([-]) = \emptyset$, $\mathsf{handled}(E[\texttt{do}\ x \leftarrow [-]\ \texttt{in}\ c_2]) = \mathsf{handled}(E)$, $\mathsf{handled}(E[\texttt{handle}\ [-]\ \texttt{with}\ \{h\}]) = \mathsf{handled}(E) \cup \mathsf{dom}(h)$,

Operational Semantics

(Red-App)	$(\lambda x.c)v; E$	$\to$	$c[v/x]; E$	
(Red-Seq)	$\texttt{do}\ x \leftarrow \texttt{return}\ v\ \texttt{in}\ c; E$	$\to$	$c[v/x]; E$	
(Red-Hdl)	$\texttt{handle return}\ v\ \texttt{with}\ \{h\}; E$	$\to$	$c[v/x]; E$	(where $\texttt{return}(x) \mapsto c \in h$)
(Cng-Psh)	$F[c]; E$	$\to$	$c; E[F]$	
(Cng-Pop)	$\texttt{return}\ v; E[F]$	$\to$	$F[\texttt{return}\ v]; E$	
(Eff-Op)	$\texttt{op}(v); E_1[\texttt{handle}\ E_2\ \texttt{with}\ \{h\}]$	$\to$	$c[v/x, \kappa x.\, \texttt{handle}\ E_2[\texttt{return}\ x]\ \texttt{with}\ \{h\}/k]; E_1$ (where $\texttt{op}(x,k) \mapsto c \in h$ and $\texttt{op} \notin \mathsf{handled}(E_2)$)	
(Eff-Cnt)	$\texttt{continue}\ E_2\ v; E_1$	$\to$	$\texttt{return}\ v; E_1[E_2]$	

Figure 2.2: The operational semantics of λ_{op}. The semantics is given on configurations of the form $\langle c, E\rangle$, with the brackets dropped for clarity. Rules are divided into three classes: reduction rules Red-X, which perform computation, congruence rules Cng-Y which manipulate the evaluation context, and effect rules Eff-Z that are special to λ_{op}

Applying Eff-Cnt, we can resume the program that was suspended

```
⟨ return () ; do(v, s) ←
                handle
                  do x ← ([−]; return 1) in
                  do y ← (print(2); return 2)
                  in x + y
                with {h}
              in return (v, f"{1};" ^ s) ⟩
```

The side-condition on Eff-Op is needed because the user may define multiple handlers with different semantics for the same effect. The side-condition resolves any ambiguity by using the *latest* handler. For example, the following program has a **read** effect that is given two definitions: it could read either 1 or 2. The ambiguity is resolved by choosing the latest handler: in this case, 1.

λ_{op}

```
handle
  handle read() with {return(y) ↦ return y; read(x, k) ↦ continue k 1}
with
  {return(y) ↦ return y; read(x, k) ↦ continue k 2}
```

```
return 1
```

Types

λ_{op}

Effects row	$\Delta ::= \emptyset \mid \Delta \cup \{\mathsf{op}_i\}$	
Value type	$T ::= \mathbb{N}$	
	$\mid T_1 \xrightarrow{\Delta} T_2$	functions
	$\mid T_1 \xrightarrow{\Delta}\!\!\!\!\rightarrow T_2$	continuations
Computation type	$T \,!\, \Delta$	
Handler type	$T_1 \,!\, \Delta \Longrightarrow T_2 \,!\, \Delta'$	

Figure 2.3: λ_{op} types. Notice that, just as terms are divided into values, computations, and handlers, types are divided into value types (T), computation types ($T \,!\, \Delta$), and handler types ($T_1 \,!\, \Delta \Longrightarrow T_2 \,!\, \Delta'$)

Type-and-Effect System

We now give a type-and-effect system to λ_{op}. Figure 2.3 collates the syntax of λ_{op} types, which I will now briefly describe.

Just like terms, types are divided into value types (for example, $\mathbb{N}$), computation types ($\mathbb{N} \,!\, \{\texttt{print}\}$), and handler types ($\mathbb{N} \,!\, \{\texttt{print}\} \Longrightarrow \mathbb{N} \,!\, \emptyset$). Since computations may have effects, computation types track unhandled effects using an effects row (Δ), which in this system is simply a set. This type-and-effect system allows us to distinguish between values, computations that return values, and computations that return values and additionally have some unhandled side effects.

Term	Type
3	$\mathbb{N}$
$\texttt{do}\ x \leftarrow \texttt{return}\ 1\ \texttt{in}\ \texttt{do}\ y \leftarrow \texttt{return}\ 2\ \texttt{in}\ x + y$	$\mathbb{N} \,!\, \emptyset$
$\texttt{do}\ x \leftarrow \texttt{print}(1); \texttt{return}\ 1\ \texttt{in}\ \texttt{do}\ y \leftarrow \texttt{return}\ 2\ \texttt{in}\ x + y$	$\mathbb{N} \,!\, \{\texttt{print}\}$

Functions are values, and are applied to other values, but produce computations on application. For example, the function

$$\lambda x : \mathbb{N}.\, \texttt{print}(x); \texttt{return}\ x$$

is a value that accepts a value of type $\mathbb{N}$ and returns a computation of type $\mathbb{N} \,!\, \{\texttt{print}\}$. We thus say functions have suspended effects, which we write $T_1 \xrightarrow{\Delta} T_2$. In this case, the function has type $\mathbb{N} \xrightarrow{\{\texttt{print}\}} \mathbb{N}$. For technical reasons, continuations and functions need to be distinguished, but in most cases they may be treated equivalently.

Handlers transform computations of one type to computations of another type. This happens in two ways: first, by handling effects, and thus removing them from the effects row (which recall represents unhandled effects). Second, by modifying the return type of computations. To reflect both abilities, handlers are given a type of the form $T_1 \,!\, \Delta \Longrightarrow T_2 \,!\, \Delta'$. For example, a handler of the form

$$\begin{aligned}&\{\texttt{return}(x) \mapsto \texttt{return}\ (x, \texttt{""});\\ &\ \texttt{print}(x, k) \mapsto \texttt{do}\ (v, s) \leftarrow \texttt{continue}\ k\ ()\ \texttt{in}\ \texttt{return}\ (v, \texttt{f"}\{x\}\texttt{;"}\ \hat{}\ s)\}\end{aligned}$$

Typing Rules

λ_{op}

$$\text{(NAT)}\;\frac{}{\Gamma \vdash n : \mathbb{N}} \qquad \text{(VAR)}\;\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \text{(LAMBDA)}\;\frac{\Gamma, x : T_1 \vdash c : T_2 \,!\, \Delta}{\Gamma \vdash \lambda x.c : T_1 \xrightarrow{\Delta} T_2} \qquad \text{(CONTINUATION)}\;\frac{\Gamma, x : T_1 \vdash c : T_2 \,!\, \Delta}{\Gamma \vdash \kappa x.c : T_1 \xrightarrow{\Delta}\!\!\!\!\!\rightarrow T_2}$$

$$\text{(APP)}\;\frac{\Gamma \vdash v_1 : T_1 \xrightarrow{\Delta} T_2 \qquad \Gamma \vdash v_2 : T_1}{\Gamma \vdash v_1\, v_2 : T_2 \,!\, \Delta} \qquad \text{(CONTINUE)}\;\frac{\Gamma \vdash v_1 : T_1 \xrightarrow{\Delta}\!\!\!\!\!\rightarrow T_2 \qquad \Gamma \vdash v_2 : T_1}{\Gamma \vdash \mathbf{continue}\, v_1\, v_2 : T_2 \,!\, \Delta}$$

$$\text{(RETURN)}\;\frac{\Gamma \vdash v : T}{\Gamma \vdash \mathbf{return}\; v : T \,!\, \Delta} \qquad \text{(DO)}\;\frac{\Gamma \vdash c_1 : T_1 \,!\, \Delta \qquad \Gamma, x : T_1 \vdash c_2 : T_2 \,!\, \Delta}{\Gamma \vdash \mathbf{do}\; x \leftarrow c_1 \;\mathbf{in}\; c_2 : T_2 \,!\, \Delta}$$

$$\text{(OP)}\;\frac{\Gamma \vdash v : T_1 \qquad \mathsf{op} : T_1 \to T_2 \in \Sigma \qquad \mathsf{op} \in \Delta}{\Gamma \vdash \mathbf{op}(v) : T_2 \,!\, \Delta} \qquad \text{(HANDLE)}\;\frac{\begin{array}{c}\Gamma \vdash c : T_1 \,!\, \Delta \qquad \Gamma \vdash h : T_1 \,!\, \Delta \Longrightarrow T_2 \,!\, \Delta' \\ \forall \mathsf{op} \in \Delta \setminus \Delta'.\, \mathsf{op} \in \mathsf{dom}(h)\end{array}}{\Gamma \vdash \mathbf{handle}\; c \;\mathbf{with}\; \{h\} : T_2 \,!\, \Delta'}$$

$$\text{(RET-HANDLER)}\;\frac{\Gamma, x : T_1 \vdash c : T_2 \,!\, \Delta'}{\Gamma \vdash \mathbf{return}(x) \mapsto c : T_1 \,!\, \Delta \Longrightarrow T_2 \,!\, \Delta'}$$

$$\text{(OP-HANDLER)}\;\frac{\begin{array}{c}\mathsf{op} : A \to B \in \Sigma \\ \Gamma \vdash h : T_1 \,!\, \Delta \Longrightarrow T_2 \,!\, \Delta' \\ \Gamma, x : A, k : B \xrightarrow{\Delta'}\!\!\!\!\!\rightarrow T_2 \vdash c : T_2 \,!\, \Delta' \qquad \Delta' \subseteq \Delta \setminus \{\mathsf{op}\} \qquad \mathbf{op}(x', k') \mapsto c' \notin h\end{array}}{\Gamma \vdash h; \mathbf{op}(x, k) \mapsto c : T_1 \,!\, \Delta \Longrightarrow T_2 \,!\, \Delta'}$$

Figure 2.4: Typing rules for λ_{op} terms

may be given type $\mathbb{N} \,!\, \{\mathbf{print}\} \Longrightarrow (\mathbb{N} \times \mathsf{String}) \,!\, \emptyset$, reflecting both the handling of the **print** effect and the transformation of the return type to include the collated print statements.

I now consider the typing rules for terms, which are collated in Figure 2.4. Most rules are standard, but a few are worth paying attention to.

First, the RETURN and DO rules. In the RETURN rule, we are allowed to assign the term **return** v any set of effects. For example, we could write:

$$\frac{}{\Gamma \vdash \mathbf{return}\; \mathtt{0} : \mathbb{N} \,!\, \{\mathbf{print}\}}$$

This flexibility is important, because to type **do** $x \leftarrow c_1$ **in** c_2, the DO rule requires both c_1 and c_2 to have the same effects. For example, without this flexibility, we would not be able to complete the following typing derivation

$$\frac{\vdots}{\Gamma \vdash \mathbf{do}\; x \leftarrow \mathbf{print}(\mathtt{0}) \;\mathbf{in}\; \mathbf{return}\; \mathtt{0} : \mathbb{N} \,!\, \{\mathbf{print}\}}$$

A valid alternative would be to forbid this flexibility and add explicit subtyping. However, such an approach would no longer be syntax directed.

Second, the Op rule. Previously, we assumed that the user declared their effects in advance. We also assume that they declare the types of their effects in advance, and that we we store the mapping from effects to types in Σ. For example, we might assume $\Sigma = \{\texttt{print} : \mathbb{N} \to 1\}$. In OCaml, this would correspond to writing:

```ocaml
type _ Effect.t += Print: nat -> unit
```

Note further the $\mathsf{op} \in \Delta$ restriction – flexibility allows us to over-approximate the effects in a term, but never underapproximate them.

Third, the Ret-Handler and Op-Handler rules, which are used to type handlers, which I will explain by means of an example. Assume we are trying to type the handler

$$\{\texttt{return}(x) \mapsto \texttt{return}\ (x, \texttt{""});\ \texttt{print}(x, k) \mapsto \texttt{do}\ (v, s) \leftarrow \texttt{continue}\, k\, ()\, \texttt{in}\, \texttt{return}\ (v, \texttt{f"}\{x\}\texttt{;"}\ \hat{}\ s)\}$$

with the type $\mathbb{N} \,!\, \{\texttt{print}\} \Longrightarrow (\mathbb{N} \times \mathsf{String}) \,!\, \emptyset$. We apply the Op-Handler rule, which is transcribed below. Preconditions are numbered for reference.

(Op-Handler)

$$\frac{\begin{array}{c}(1)\, \mathsf{op} : A \to B \in \Sigma \\ (2)\, \Gamma \vdash h : T_1 \,!\, \Delta \Longrightarrow T_2 \,!\, \Delta' \\ (3)\, \Gamma, x : A, k : B \xrightarrow{\Delta'\!\!\!\!\rightarrow} T_2 \vdash c : T_2 \,!\, \Delta' \qquad (4)\, \Delta' \subseteq \Delta \setminus \{\mathsf{op}\} \qquad (5)\, \texttt{op}(x', k') \mapsto c' \notin h\end{array}}{\Gamma \vdash h; \texttt{op}(x, k) \mapsto c : T_1 \,!\, \Delta \Longrightarrow T_2 \,!\, \Delta'}$$

The preconditions of the Op-Handler rule direct us to check, in turn:

(1) $\texttt{print} : \mathbb{N} \to 1 \in \Sigma$, which is true by assumption

(2) Recursively check the rest of the handler $h = \texttt{return}(x) \mapsto \texttt{return}\ (x, \texttt{""})$, ensuring it has type $\mathbb{N} \,!\, \{\texttt{print}\} \Longrightarrow (\mathbb{N} \times \mathsf{String}) \,!\, \emptyset$. This follows from a trivial application of the Ret-Handler rule.

(3) Assuming x has type $\mathbb{N}$ and k has type $1 \xrightarrow{\emptyset\!\!\!\!\rightarrow} (\mathbb{N} \times \mathsf{String})$, the body

$$\texttt{do}\ (v, s) \leftarrow \texttt{continue}\, k\, ()\, \texttt{in}\, \texttt{return}\ (v, \texttt{f"}\{x\}\texttt{;"}\ \hat{}\ s)$$

has type $(\mathbb{N} \times \mathsf{String}) \,!\, \emptyset$. This is easy to show.

(4) That the handler *only* removes **print** from the effects row, and no other effects. This check passes, but would fail if we tried to type the handler with, for example, $\mathbb{N} \,!\, \{\texttt{print}, \texttt{get}\} \Longrightarrow (\mathbb{N} \times \mathsf{String}) \,!\, \emptyset$

(5) That there are no other handlers for **print** in h.

A full typing derivation may be found in the appendix.

Metatheory

I will build not only on λ_{op}, but on metatheoretic properties of λ_{op}, which are proven by Bauer and Pretnar [2]. We first state the standard progress and preservation properties.

Theorem 2.2.1 (Progress) *If $\Gamma \vdash E[c] : T \,!\, \Delta$ and then either*

1. *c of the form* **return** v *and* $E = [-]$,
2. *c of the form* **op**(v) *for some op* $\in \Delta$,
3. $\exists . E', c'$ *such that* $\langle c; E \rangle \to \langle c'; E' \rangle$

Theorem 2.2.2 (Preservation) *If $\Gamma \vdash E[c] : T \,!\, \Delta$ and $\langle c; E \rangle \to \langle c'; E' \rangle$, then $\Gamma \vdash E'[c'] : T \,!\, \Delta$*

As a corollary, we obtain type safety.

Corollary 2.2.1 (Type Safety) *If $\cdot \vdash E[c] : T \,!\, \Delta$ and then either*

1. *c of the form* **return** v *and* $E = [-]$ *for* $\cdot \vdash v : T$
2. *c of the form* **op**(v) *for some op* $\in \Delta$,
3. $\exists . E', c'$ *such that* $\langle c; E \rangle \to \langle c'; E' \rangle$ *and* $\cdot \vdash E'[c'] : T \,!\, \Delta$

Finally, λ_{op} is a *fine-grained call-by-value* [14] approach. We first define a notion of contextual equivalence $\cong_{ctx}$. Informally, two computations/values are contextually equivalent if they behave the same in all "relevant" contexts and at all ground (first-order) types (in this case, just $\mathbb{N}$).

Definition 2.2.1 (Contextual Equivalence) *c and c' are contextually equivalent at context Γ and type $T \,!\, \Delta$, written $\Gamma \vdash c \cong_{\text{ctx}} c' :' T \,!\, \Delta$ if*

1. $\Gamma \vdash c : T \,!\, \Delta$ *and* $\Gamma \vdash c' : T \,!\, \Delta$
2. *For all E such that $\cdot \vdash E[c] : \mathbb{N} \,!\, \emptyset$ and $\cdot \vdash E[c'] : \mathbb{N} \,!\, \emptyset$,*

$$\langle c; E \rangle \to^* \langle \texttt{return}\ v; [-] \rangle \iff \langle c'; E \rangle \to^* \langle \texttt{return}\ v; [-] \rangle$$

When the context Γ is empty, we write $c \cong_{\text{ctx}} c' : T \,!\, \Delta$, and when the type is unimportant, we write $c \cong_{\text{ctx}} c'$

Theorem 2.2.3 (Fine-Grained CBV) *λ_{op} is a fine-grained call-by-value language, meaning in particular that the following equations hold (notationally, I use c for computations, v for values, and f for function values):*

1. $\texttt{do}\ x \leftarrow \texttt{return}\ v\ \texttt{in}\ c \cong_{\text{ctx}} c[v/x]$
2. $c \cong_{\text{ctx}} \texttt{do}\ x \leftarrow c\ \texttt{in}\ \texttt{return}\ x$
3. $\texttt{do}\ x \leftarrow c_1\ \texttt{in}\ (\texttt{do}\ y \leftarrow c_2\ \texttt{in}\ c_3) \cong_{\text{ctx}} \texttt{do}\ y \leftarrow (\texttt{do}\ x \leftarrow c_1\ \texttt{in}\ c_2)\ \texttt{in}\ c_3$

4. $(\lambda x.c)v \cong_{\text{ctx}} c[v/x]$

5. $f \cong_{\text{ctx}} \lambda x.fx$

In the third equation, we assume x not free in c_3.

2.2.3 The Design Space of Effect Handlers

The design space of effect handlers is large. I consider three key design decisions made by different systems.

1. **Named or Unnamed Handlers**
   Can I invoke a specific handler for an operation?

   In λ_{op}, when there are multiple handlers for the same effect, we invoke the "nearest" or "most recent" handler for that effect. An alternative approach is **named handlers** [25], where, by associating each handler with a *name*, we can more easily specify which handler should be invoked.

   While named handlers can be more ergonomic, they do not provide greater expressiveness than using distinct effects [25]. I do not consider named handlers in this thesis.

2. **Deep, Shallow, or Sheep Handlers**
   Are multiple instances of the same effect handled by the same handler?

   In λ_{op}, continuations reinstate handlers (Eff-Cnt) and thus multiple instances of the same effect are handled by the same handler. For example, in the following example, the effect **addn** is handled by the same handler, adding one each time. We say these handlers are **deep**.

   λ_{op}

   handle
   $\quad$ **addn**$(1) +$ **addn**(2)
   with
   $\quad \{$**return**$(x) \mapsto$ **return** $x;$
   $\quad\;$ **addn**$(y, k) \mapsto$ **continue** $k\ (y+1)\}$

   return 5

   We may also choose *not* to reinstate the handler, in an approach known as **shallow** handlers [6]. The example above would be stuck, since the second **addn** would not be handled.

   Finally, we could choose to modify the interface for **continue** such that it accepts a handler

   $$\textbf{continue}\ k\ v\ h$$

   This would allow multiple effects to be handled by different handlers. That is, we could add one the first time **addn** is performed, and two the second time. These handlers behave as a hybrid of shallow and deep handlers, and are thus termed **sheep** handlers [16].

`OCaml` allows the programmer to choose between shallow and deep handlers. Since most prior work on scope extrusion focuses on deep handlers [8], we focus on those.

3. **One-Shot or Multi-Shot Continuations**
   How many times can one resume the same continuation?

   In λ_{op}, continuations may be resumed multiple times. For example, we can write

```
handle
  performTwice(1)
with
  {return(x) ↦ return x;
   performTwice(y, k) ↦ (continue k y) + (continue k y)}
```
λ_{op}

```
return 2
```

   We say the effect system permits **multi-shot continuations**. Multi-shot continuations are useful for simulating certain effects, like non-determinism [16].

   In other systems, like `OCaml` and `WasmFX` [16], this is not allowed: continuations are only allowed to be resumed once. These systems permit **one-shot continuations**.

   Although continuations in `OCaml` are one-shot, due to the utility of multi-shot continuations, I believe it is worthwhile to study effect systems with multi-shot continuations.

2.3 Scope Extrusion

I now turn my attention to scope extrusion, which arises from the unexpected interaction of effects and metaprogramming. To illustrate scope extrusion, I will first extend λ_{op} (Page 11) with AST constructors $\texttt{Var}(x_T)$, $\texttt{Nat}(n)$, `Lam`, and `Plus`. For example, we may generate the AST of $\lambda x : \mathbb{N}.\, x + 0$ as follows:

```
return Lam(Var(x_ℕ), Plus(Var(x_ℕ), Nat(0)))
```
λ_{op}

Listing 8 illustrates the problem of scope extrusion. The program constructs the AST of $\lambda x : \mathbb{N}.\, x$, but additionally performs an effect, **`extrude`**, with type $\mathbb{N}\,\texttt{expr} \to \mathbb{N}\,\texttt{expr}$. The handler for **`extrude`** discards the continuation, simply returning the value it was given: $\texttt{Var}(x_{\mathbb{N}})$. The entire program evaluates to $\texttt{Var}(x_{\mathbb{N}})$, and the generated AST is ill-scoped. We say that the result of evaluation demonstrates scope extrusion.

```
handle
  do body ← extrude(Var(x_ℕ)) in return Lam(Var(x_ℕ), body)
with
  {return(u) ↦ return Nat(0);
   extrude(y, k) ↦ return y}
```
λ_{op}

```
return Var(x_ℕ)
```

Listing 8: A λ_{op} program that evaluates to the $\texttt{Var}(x_{\mathbb{N}})$. The AST is ill-scoped, and thus exhibits scope extrusion. It will be used as a running example.

It is difficult to give a precise definition to scope extrusion, because there are multiple competing definitions [11, 13], and many are given informally. For example, is scope extrusion a property of the *result* of evaluation [13], as in Listing 8, or is it a property of *intermediate* configurations [11]? We can, for example, build ASTs with extruded variables, that are bound at some future point. In Listing 9, we produce the intermediate AST $\texttt{Plus}(\texttt{Nat}(0), \texttt{Var}(x_{\mathbb{N}}))$, which is not well scoped. However, the result of evaluation is well scoped: $\texttt{Lam}(\texttt{Var}(x_{\mathbb{N}}), \texttt{Plus}(\texttt{Nat}(0), \texttt{Var}(x_{\mathbb{N}})))$. Does Listing 9 exhibit scope extrusion?

λ_{op}

```
handle
  do body ← extrude(Var(x_ℕ)); return Var(x_ℕ) in return Lam(Var(x_ℕ), body)
with
  {return(u) ↦ return u;
   extrude(y, k) ↦ do z ← return Plus(Nat(0), y) in continue k z}
```

```
return Lam(Var(x_ℕ), Plus(Nat(0), Var(x_ℕ)))
```

Listing 9: A λ_{op} program that may, or may not demonstrate scope extrusion, depending on one's definition. The final result of the program is well-scoped, but not all intermediate results are well-scoped. If scope extrusion is a property of the resulting AST, then this does not display scope extrusion. If, instead, it is a property of intermediate ASTs, then this does display scope extrusion.

Nevertheless, all definitions agree on the example in Listing 8. Making precise the competing definitions of scope extrusion these competing definitions, and their relation to one another, is a contribution of this dissertation.

2.3.1 Existing Solutions to the Scope Extrusion Problem

There are multiple solutions to the problem of scope extrusion. The solution space can be broadly divided into two types of approaches: static (type-based) and dynamic. I will now survey two dynamic approaches, which I term the lazy and eager checks, and one static approach, the method of refined environment classifiers [13, 8].

Lazy Dynamic Check

Scope extrusion, at least, of the kind in Listing 12, may seem trivial to resolve: evaluate the program to completion, and check that the resulting AST is well-scoped [11]. I term this the Lazy Dynamic Check. This approach, while clearly correct and maximally expressive, is not ideal for efficiency and error reporting reasons.

To illustrate the inefficiency of this approach, consider a slight variation of Listing 8, Listing 10. In Listing 10, we can, in theory, report a warning as soon scope extrusion is detected. However, waiting for the result of the program can be much more inefficient.

In terms of error reporting, note that, in waiting for the result of execution, we lose information about *which program fragment* was responsible for scope extrusion, reducing the informativeness of reported errors [11].

```
λop
do x ← handle
        do body ← extrude(Var(x_ℕ)) in return Lam(Var(x_ℕ), body)
      with
        {return(u) ↦ return Nat(0);
         extrude(y, k) ↦ return y}
in some very long program; return x
---
return Var(x_ℕ)
```

Listing 10: A λ_{op} program that evaluates to the $\texttt{Var}(x_{\mathbb{N}})$. Executing the entire program to determine if it exhibits scope extrusion is inefficient.

Eager Dynamic Check

A second dynamic check, motivated by the problems with the Lazy Dynamic Check, adopts a stricter definition of scope extrusion. During the code generation process, one inserts checks into the running program, reporting errors when one encounters unbound free variables in intermediate ASTs. Hence, the Eager Dynamic Check would classify the program in Listing 9 as exhibiting scope extrusion. The Eager Dynamic Check has been adopted by BER MetaOCaml, and offers better efficiency and error reporting guarantees over the Lazy Dynamic Check [11].

However, the Eager Dynamic Check is not without issue. The problem relates to the manner in which checks are inserted, which we will make precise later. To illustrate the problem, consider Listing 11, a slight variation of Listing 9 in which we replace the program fragment $\texttt{Plus}(\texttt{Nat}(0), y)$ with y.

```
λop
handle
  do body ← extrude(Var(x_ℕ)); return Var(x_ℕ) in return Lam(Var(x_ℕ), body)
with
  {return(u) ↦ return u;
   extrude(y, k) ↦ do z ← return y in continue k z}
---
return Lam(Var(x_ℕ), Plus(Nat(0), Var(x_ℕ)))
```

Listing 11: A λ_{op} program that is a slight variation of Listing 9, but that (unlike Listing 9) passes the Eager Dynamic Check.

While the program in Listing 11 produces an intermediate AST with extruded variables, because of the mechanism for inserting checks, it passes the Eager Dynamic Check. I assert that this behaviour is unintuitive, and exposes too much of the internals to the programmer.

Refined Environment Classifiers

Refined Environment Classifiers are a static check that uses the type system to prevent scope extrusion. Recall that metaprogramming involves the *creation* and *manipulation* of ASTs. Refined environment classifiers prevent scope extrusion by checking:

1. *Created* ASTs are well-scoped

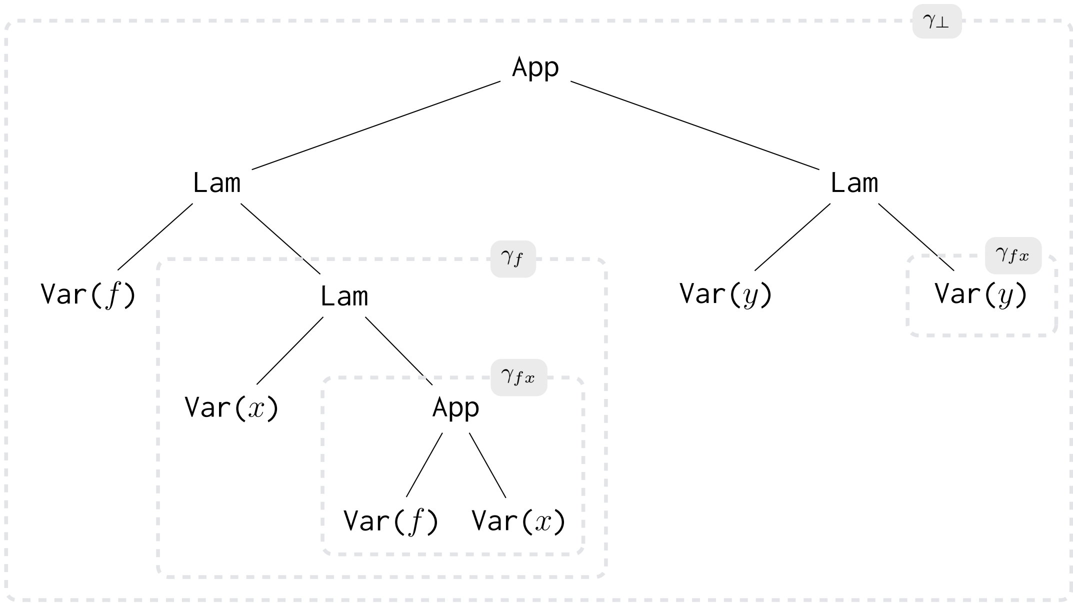


Figure 2.5: The AST of $(\lambda f.\lambda x.fx)(\lambda y.y)$, where each scope is labelled with the corresponding environment classifier.

2. *Manipulating* ASTs preserves well-scopedness

We shall first ignore the ability to manipulate ASTs, and consider how to ensure *created* ASTs are well-scoped. What does it mean to be well-scoped? Consider Figure 2.5, the AST of

$$(\lambda f.\lambda x.fx)(\lambda y.y)$$

Informally, a scope represents a set of variables that are permitted to be free. In the example, there are four scopes: one where no variables are free, one where only f is free, one where f and x are free, and one where y is free. An AST is well-scoped *at a scope* if it is well-typed, and where the only free variables are those permitted by the scope.

Refined environment classifiers make this notion precise. Each classifier represents a scope. The AST has four scopes, corresponding to four classifiers:

$\gamma_\bot$ The top-level, where no free variables are permitted

γ_f Only $\mathtt{Var}(f)$ permitted to be free

γ_{fx} Only $\mathtt{Var}(f)$ and $\mathtt{Var}(x)$ permitted to be free

γ_y Only $\mathtt{Var}(y)$ permitted to be free

As every variable binder creates a scope, we may refer to the classifier created by $\mathtt{Var}(\alpha)$, $\mathsf{classifier}(\mathtt{Var}(\alpha))$. For example, $\gamma_{fx} = \mathsf{classifier}(\mathtt{Var}(x))$.

With classifiers, we may be precise about "the variables permitted to be free (within the scope)". As illustrated by the nesting in Figure 2.5, scopes are related to other scopes. For example, since the scope γ_{fx} is created within scope γ_f, any variable tagged with γ_f may be safely used in the scope γ_{fx}. We say γ_f is compatible with γ_{fx}, and write

$$\gamma_f \sqsubseteq \gamma_{fx}$$

The compatibility relation ($\sqsubseteq$) is a partial order, meaning $\sqsubseteq$ is reflexive, anti-symmetric, and transitive, and identifies a smallest classifier. Reflexivity expresses that $\mathsf{Var}(\alpha)$ may be used within the scope it creates

$$\forall\gamma.\, \gamma \sqsubseteq \gamma$$

anti-symmetric, since nesting only proceeds in one direction

$$\forall\gamma_1, \gamma_2.\, \gamma_1 \sqsubseteq \gamma_2 \wedge \gamma_2 \sqsubseteq \gamma_1 \implies \gamma_1 = \gamma_2$$

and transitive, since we should count nestings within nestings

$$\forall\gamma_1, \gamma_2, \gamma_3.\, \gamma_1 \sqsubseteq \gamma_2 \wedge \gamma_2 \sqsubseteq \gamma_3 \implies \gamma_1 \sqsubseteq \gamma_3$$

$\gamma_\perp$ acts as the least element of this partial order

$$\forall\gamma.\, \gamma_\perp \sqsubseteq \gamma$$

Given a classifier γ, we may now define the variables permitted to be free in γ, written $\mathsf{permitted}(\gamma)$

$$\mathsf{permitted}(\gamma) \triangleq \{\mathsf{Var}(\alpha) \mid \mathsf{classifier}(\mathsf{Var}(\alpha)) \sqsubseteq \gamma\}$$

For example, $\mathsf{permitted}(\gamma_{fx}) = \{\mathsf{Var}(f), \mathsf{Var}(x)\}$

We now say that an AST n is well-scoped at type T and scope γ if it is well-typed at T, and all free variables in n are in $\mathsf{permitted}(\gamma)$. We write

$$\Gamma \vdash^{\gamma} n : T$$

Ensuring that created ASTs are well-scoped is the responsibility of the type system. The key rule is the C-Abs rule, which, **assuming we know that we are creating an AST**, has roughly the following shape[2]:

$$\text{(C-ABS)}\quad \frac{\gamma \in \Gamma \qquad (2)\, \gamma' \text{fresh} \qquad (3)\, \Gamma, \gamma', \gamma \sqsubseteq \gamma', (x : T_1)^{\gamma'} \vdash^{\gamma'} n : T_2}{(1)\, \Gamma \vdash^{\gamma} \lambda x.n : T_1 \longrightarrow T_2}$$

The premises and conclusions have been numbered for reference, and many technical details have been simplified for clarity. Figure 2.6 visually depicts the typing rule.

(1) The goal of the typing rule is to ensure the function is well-scoped at type $T_1 \to T_2$ and scope γ.

(2) Since the function introduces a new variable binder, $\mathsf{Var}(x)$, one has to create a new scope. This is achieved by picking a fresh classifier γ'.

(3) We record the following:

  (a) Since γ' is created within the scope of γ, $\gamma \sqsubseteq \gamma'$.

  (b) $\mathsf{classifier}(\mathsf{Var}(x)) = \gamma'$ (as a shorthand $(x : T_1)^{\gamma'}$)

  With this added knowledge, we ensure that the function body is well-scoped at type T_2 and γ'.

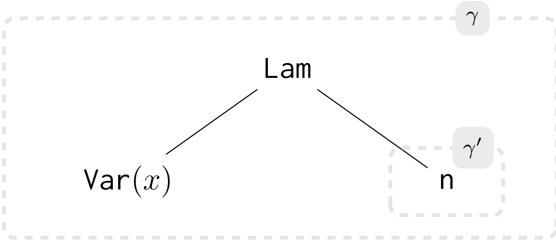


Figure 2.6: Visual depiction of the (C-Abs) typing rule.

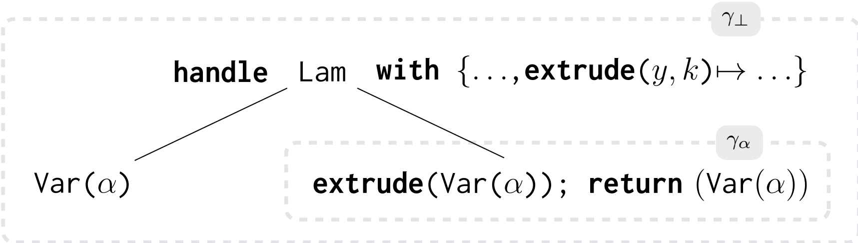


Figure 2.7: The "AST" of the scope extrusion example, Listing 8. Notice that in place of AST nodes, we may now have compile-time executable code that *evaluate* to AST nodes.

The above example focused on **creating** ASTs, and had no compile-time executable code. We now consider how to maintain well-scopedness while **manipulating** ASTs. We consider the "AST" of the scope extrusion example, Listing 8 (Figure 2.7). Notice that in place of AST nodes, we may now have compile-time executable code that *evaluate* to AST nodes. Thus, both code and AST nodes reside within scopes. We have two classifiers: $\gamma_\perp$ and γ_α, with classifier(Var(α)) $= \gamma_\alpha$.

Key to the prevention of scope extrusion is the typing of handlers and operations, like **extrude**, that manipulate ASTs. Since these rules are complex, we describe them informally. The handle expression

$$\textbf{handle } e \textbf{ with } \{h\}$$

is in scope $\gamma_\perp$. Therefore, for each operation handled by h, such as **extrude**, the argument to the operation must either not be an AST, or be an AST that is well-scoped at some $\gamma \sqsubseteq \gamma_\perp$. However, Var($\alpha$) is typed at γ_α, and clearly, $\gamma_\alpha \not\sqsubseteq \gamma_\perp$. There is thus no way to type the scope extrusion example in Listing 8.

Note that the analysis was independent of the *body* of the handler. Therefore, the examples in Listings 9 and 11 are *also* not well-typed. Perhaps somewhat surprisingly, so too is Listing 12 (which would pass both the Eager and Lazy Dynamic Checks). Refined environment classifiers statically prevent variables (Var(α)) from becoming *available* in program fragments (**op**$(y,k) \mapsto \ldots$) where, *if misused*, *might* result in scope extrusion. This is, of course, an over-approximation. It means that the refined environment classifiers check prevents not only both types of scope extrusion, but even more benign examples, such as that in Listing 12.

[2]I will revisit this assumption when introducing the type system for my calculus

λ_{op}

```
handle
  do body ← extrude(Var(x_ℕ)); return Var(x_ℕ) in return Lam(Var(x_ℕ), body)
with
  {return(u) ↦ return App(u, Nat(1));
   extrude(y, k) ↦ return Nat(0)}
```

```
return Nat(0)
```

Listing 12: A λ_{op} program that passes the Eager and Lazy Dynamic Checks, but is not well-typed under the Refined Environment Classifiers type system.

Listing 12 thus illustrates one of the key drawbacks of Refined Environment Classifiers: the check is too stringent, and restricts expressiveness.

3 Calculus

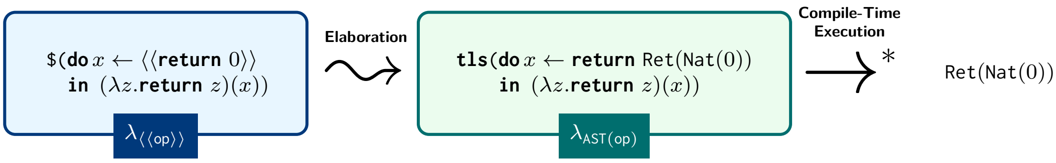


Figure 3.1: $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ is first elaborated into $\lambda_{\mathsf{AST(op)}}$, which is then executed **at compile-time** to obtain the AST of a run-time program.

To re-iterate, I am considering the interaction between homogenous, compile-time, two-stage **metaprogramming** (Page 4), and an **effect system** with deep handlers and multi-shot continuations (Page 9). In this chapter, I describe a calculus, $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$, for studying said interaction. $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ will have both metaprogramming and effect handlers. To the best of my knowledge, this is the first calculus in which one may write effectful compile-time code that generates effectful run-time code. However, $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ will not mediate the interaction between metaprogramming and effects: scope extrusion prevention is not a language feature. Rather, the aim will be to implement different scope extrusion checks as algorithms in $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$, such that the checks may be evaluated in a comparative fashion.

Programs written in $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ cannot be directly executed. Rather, following the style of Xie et al. [26], one must first elaborate (or compile) from $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ (the "source" language) to a "core" language, $\lambda_{\mathsf{AST(op)}}$. Programs written in $\lambda_{\mathsf{AST(op)}}$ may then be executed, to obtain the AST of a run-time program. This process is summarised in Figure 3.1. Elaboration is necessary, since it is the point at which dynamic checks may be inserted.

In this chapter, I will first introduce $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ (Section 3.1), then $\lambda_{\mathsf{AST(op)}}$ (Section 3.2). Following this, I will describe the elaboration from $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ to $\lambda_{\mathsf{AST(op)}}$ (Section 3.3). Finally, I will discuss the metatheoretic properties of $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ (Section 3.4).

3.1 The source language: $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$

$\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ extends λ_{op} with quotes and splices. Recall that λ_{op}, following a fine-grained call-by-value approach, divides terms into two syntactic categories, values v and computations c. $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ is similar, dividing terms into values v and expressions e (Figure 3.7)

Notice that we cannot quote values: we *must* generate effectful programs.

3.1.1 Type System

I will now introduce the $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ type system, by first introducing the types, and then the typing rules. The $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ types are summarised in Figure 3.3. I highlight three important details: types are stratified into two levels (-1 for compile-time and 0 for run-time), effect rows are similarly stratified, and run-time code is made available at compile-time via a Code type.

Syntax

$\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$

$$\begin{array}{llll} \textbf{Values} & v & ::= & \ldots \\ \textbf{Expressions} & e & ::= & \ldots \mid \langle\langle e \rangle\rangle \mid \$e \end{array}$$

Figure 3.2: $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ syntax. The syntax is broadly the same as λ_{op}, except with the addition of quotes and splices.

Effects Row

$\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$

$$\begin{array}{ll} \textbf{Run-Time} & \xi ::= \cdot \mid \xi \cup \{\mathsf{op}_i^0\} \\ \textbf{Compile-Time} & \Delta ::= \cdot \mid \Delta \cup \{\mathsf{op}_i^{-1}\} \end{array}$$

Types

$$\begin{array}{llll} \textbf{Level 0} & \text{Values} & T^0 ::= \mathbb{N}^0 & \text{naturals} \\ & & \quad \mid (T_1^0 \xrightarrow{\xi} T_2^0)^0 & \text{functions} \\ & & \quad \mid (T_1^0 \xrightarrow{\xi}\!\!\!\!\!\rightarrow T_2^0)^0 & \text{continuations} \\ & \text{Computations} & T^0 \,!\, \xi & \\ & & \mid T^0 \,!\, \Delta; \xi & \\ & \text{Handlers} & (T_1^0 \,!\, \xi \Longrightarrow T_2^0 \,!\, \xi') \,!\, \Delta & \\ & & & \\ \textbf{Level } -1 & \text{Values} & T^{-1} ::= \mathbb{N}^{-1} & \text{naturals} \\ & & \quad \mid (T_1^{-1} \xrightarrow{\Delta} T_2^{-1})^{-1} & \text{functions} \\ & & \quad \mid (T_1^{-1} \xrightarrow{\Delta}\!\!\!\!\!\rightarrow T_2^{-1})^{-1} & \text{continuations} \\ & & \quad \mid \mathsf{Code}(T^0 \,!\, \xi)^{-1} & \text{run-time code} \\ & \text{Computations} & T^{-1} \,!\, \Delta & \\ & \text{Handlers} & T_1^{-1} \,!\, \Delta \Longrightarrow T_2^{-1} \,!\, \Delta' & \end{array}$$

Figure 3.3: $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ types. I highlight three important elements: first, stratifying types into two levels, 0 and -1. Second, stratifying effects into two levels, ξ (for run-time effects) and Δ for compile-time effects. Third, the Code type at level -1 allows for compile-time programs to manipulate ASTs of run-time code.

First, **types are stratified into two levels, T^0 (run-time), and T^{-1} (compile-time).** To motivate this stratification, consider the following question: what is the type of the number 3 in $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$? Perhaps surprisingly, the answer is not $\mathbb{N}$. Since we are working with a two-stage system, we must be careful to disambiguate between run-time naturals and compile-time naturals, since these are not interchangeable. For example, the following program should **not** be well-typed, since 3 is a compile-time natural, whereas x is a run-time natural.

$$\lambda x : \mathbb{N}.\, \$(3 + x)$$

However, removing the splice makes the program well-typed

$$\lambda x : \mathbb{N}.\, 3 + x$$

Following Xie et al. [26], I introduce integer levels to enforce separation between compile-time and run-time naturals. While the precise notion of level is slightly more involved (see below), for my purposes, it is sufficient to think of level 0 as run-time (so $\mathbb{N}^0$ is a run-time natural), and -1 for compile-time. The ill-typed example becomes

$$\lambda x : \mathbb{N}^0.\, \$((3 : \mathbb{N}^{-1}) + (x : \mathbb{N}^0))$$

and the well-typed example

$$\lambda x : \mathbb{N}^0.\, (3 : \mathbb{N}^0) + (x : \mathbb{N}^0)$$

More precisely, levels are defined as follows:

Definition 3.1.1 (Level) *The level of an expression e is calculated by subtracting the number of surrounding splices from the number of surrounding quotations.*

The definition of level generalises to multi-stage languages, where negative levels $(-1, -2, \ldots)$ represent compile-time and non-negative levels $(0, 1, \ldots)$ represent run-time. In a multi-staged language, separation is even more granular: for example, level 1 and level 0 naturals, despite both being run-time naturals, are disambiguated. However, since we only deal with two stages, we only consider two levels, 0 and -1. The definition, and the examples above, further imply that the "default" level, in the absence of quotes and splices, is level 0. Intuitively, in the absence of quotes and splices, the programmer is ignoring metaprogramming facilities, and constructing a run-time program.

Notice that the opening question was slightly devious[1]! We cannot assign a type to *program fragments*, like 3, since without knowledge of the wider context, we cannot know which level we are at: in the ill-typed example, 3 occurs under a splice, but no quotes, so it has type $\mathbb{N}^{-1}$, and in the well-typed example, it has type $\mathbb{N}^0$. Unless otherwise stated, I will always assume program fragments are not nested in any quotes or splices, and thus occur at level 0.

Second, **effect rows are stratified into ξ (run-time) and Δ (compile-time).** In the following example, we print 1 at compile-time, and 2 at run-time. Further, we read an integer at run-time.

$$\$(\mathtt{print}(1); \langle\langle \mathtt{print}(2); \mathtt{readInt}() \rangle\rangle)$$

[1] sorry

Hence, $\Delta = \{\texttt{print}\}$ and $\xi = \{\texttt{print}, \texttt{readInt}\}$. We may now disambiguate between different computation types:

T^0	Compile-time value, run-time value (value types) *Example*: The type of x in $\lambda x.\texttt{return}\ x$
$T^0\,!\,\xi$	Compile-time value, run-time computation *Example*: The type of x in $\$(\texttt{do}\ x \leftarrow \langle\langle \texttt{return}\ 1 \rangle\rangle\ \texttt{in}\ \texttt{return}\ x)$
$T^0\,!\,\Delta$	Compile-time computation, run-time value *Example*: $\lambda x.\texttt{return}\ x$
$T^0\,!\,\Delta;\xi$	Compile and run-time computation *Example*: $\$(\texttt{do}\ x \leftarrow \langle\langle \texttt{return}\ 1 \rangle\rangle\ \texttt{in}\ \texttt{return}\ x)$

Third, **there is an level -1 Code type, representing run-time ASTs.** By stratifying types to two levels, we have ensured that run-time (resp. compile-time) terms only interact with run-time (compile-time) terms. However, to enable meta-programming, run-time terms *should be available* at compile-time as ASTs. This is exactly the role of the Code type, thus allowing level -1 programs to manipulate ASTs of level 0 terms.

Putting it all together, we can now interpret complex $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ types, like

Compile-time ② effects

Compile-time ① function

$$(\mathsf{Code}(\mathbb{N}^0!\{\mathsf{print}\})^{-1} \xrightarrow{\{\mathsf{get}\}} \mathsf{Code}(\mathbb{N}^0\,!\,\{\mathsf{print}, \mathsf{readInt}\})^{-1})^{-1}$$

③ Input: AST of a run-time computation of type $\mathbb{N}\,!\,\{\texttt{print}\}$

④ Output: AST of a run-time computation of type $\mathbb{N}\,!\,\{\texttt{print}, \texttt{readInt}\}$

Typing Judgement

Having described the types, I now present the type system. The typing rules are collated in Figures 3.5 and 3.6. I will first explain the typing judgement, then highlight some key rules.

The shape of the typing judgement is mostly familiar, though, as in Xie et al. [26], I add level information and compiler modes. Level information will turn out to be redundant, but we will revisit this later.

$$\Gamma \vdash^{\mathsf{Level}}_{\mathsf{Mode}} e : T$$

Level. Recall that, when describing the λ_{op} types, I argued that one cannot type a program fragment, like 3, directly. One must also know the *level* (0 or -1), which is accordingly attached to the typing judgement.

Mode. For the purposes of elaboration, it can be useful to classify code into three categories:

- **c** Code that is **ambient** and **inert**.
  No surrounding quotes or splices
- **s** Code that **manipulates ASTs** at compile-time.
  Last surrounding annotation is a splice

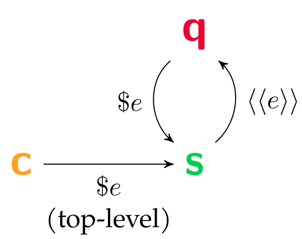


Figure 3.4: Transitions between modes **c**, **s**, and **q**. Top-level splices transition from **c** to **s**, quotes transition from **s** to **q**, and splices (under quotes) transition from **q** to **s**.

q Code that **builds ASTs** to be manipulated at compile time.
Last surrounding annotation is a quote

To illustrate the purpose of the modes, consider the following meta-program, which evaluates to the AST of $\lambda x.1+2+3$. This program adopts the shorthand of Section 2.2.2, where `return`s are implicitly inserted (for example, $\langle\langle 1\rangle\rangle$ should really be $\langle\langle \texttt{return}\ 1\rangle\rangle$), and we elide the order of operation ($1+2$ should really be written out using **do**). The emphasis is on the three modes:

$$\lambda x.\,\$(\mathbf{do}\ f \leftarrow (\lambda y.\langle\langle \$(y) + 2\rangle\rangle)\ \mathbf{in}\,\mathbf{do}\ a \leftarrow \langle\langle 1\rangle\rangle\ \mathbf{in}\ f a) + 3$$

c Identifies AST nodes that are **ambient** (within which computation may take place) and **inert** (cannot themselves be manipulated at compile-time).

$$\lambda x.\,\$(\mathbf{do}\ f \leftarrow (\lambda y.\langle\langle \$(y) + 2\rangle\rangle)\ \mathbf{in}\,\mathbf{do}\ a \leftarrow \langle\langle 1\rangle\rangle\ \mathbf{in}\ f a) + 3$$

s Identifies code that can be executed at compile-time to **manipulate ASTs**. Will be fully reduced at compile-time, and will not appear at run-time.

$$\lambda x.\,\$(\mathbf{do}\ f \leftarrow (\lambda y.\langle\langle \$(y) + 2\rangle\rangle)\ \mathbf{in}\,\mathbf{do}\ a \leftarrow \langle\langle 1\rangle\rangle\ \mathbf{in}\ f a) + 3$$

q Identifies code that **builds ASTs** (like **c**-mode) that can be manipulated at compile-time (unlike **c**-mode), to create run-time programs.

$$\lambda x.\,\$(\mathbf{do}\ f \leftarrow (\lambda y.\langle\langle \$(y) + 2\rangle\rangle)\ \mathbf{in}\,\mathbf{do}\ a \leftarrow \langle\langle 1\rangle\rangle\ \mathbf{in}\ f a) + 3$$

We may also describe how *transitions* between modes occur:

1. Top-level splices ($\$e$) transition from **c** (outside the splice) to **s** (within).
2. Quotes ($\langle\langle e\rangle\rangle$) transition from **s** (outside the quote) to **q** (within).
3. Splices ($\$e$) transition from **q** (outside the splice) to **s** (within).

Transitions between modes are illustrated in Figure 3.4.

Since $\lambda_{\langle\langle \mathsf{op}\rangle\rangle}$ has only two levels, and my type system will ban nested splices and quotations ($\$\$e$ and $\langle\langle\langle\langle e\rangle\rangle\rangle\rangle$ are not valid program fragments), the compiler mode will uniquely identify the level (**c** and **q** imply level 0, and **s** level -1). I thus drop the level from my typing judgement, leaving it implicit.

Further, the typing judgements for c and q will be identical in almost all cases. To avoid repetition, I introduce the following notation

$$\Gamma \vdash_{\mathsf{c|q}} e : T$$

Where, for example, the typing rule

$$\frac{\Gamma_1 \vdash_{\mathsf{c|q}} e_1 : T_1 \qquad \cdots \qquad \Gamma_n \vdash_{\mathsf{c|q}} e_n : T_n}{\Gamma \vdash_{\mathsf{c|q}} e : T}$$

stands for two typing rules, one in c-mode and one in q-mode.

$$\frac{\begin{array}{c}\Gamma_1 \vdash_{\mathsf{c}} e_1 : T_1 \\ \cdots \\ \Gamma_n \vdash_{\mathsf{c}} e_n : T_n\end{array}}{\Gamma \vdash_{\mathsf{c}} e : T} \qquad \frac{\begin{array}{c}\Gamma_1 \vdash_{\mathsf{q}} e_1 : T_1 \\ \cdots \\ \Gamma_n \vdash_{\mathsf{q}} e_n : T_n\end{array}}{\Gamma \vdash_{\mathsf{q}} e : T}$$

The c and q-mode typing rules are summarised in Figure 3.5. The s-mode typing rules are summarised in Figure 3.6. In all modes, rules are extremely similar to the λ_{op} typing rules (Figure 2.4, Page 17). I focus on three important rules: s-Quote, q-Splice, and c-Splice.

Recall that the Code type makes available at compile-time a representation of ASTs of level 0 programs. Recall further that $\langle\langle e \rangle\rangle$ is the mechanism for *creating* ASTs (elements of type Code) from run-time programs. This intuition is captured by the s-Quote rule, where, to verify that $\langle\langle e \rangle\rangle$ is a valid AST of type Code, we verify that e is a run-time program of the corresponding type.

$$\text{(s-Quote)}\quad \frac{\Gamma \vdash_{\mathsf{q}} e : T^0 \,!\, \Delta ; \xi}{\Gamma \vdash_{\mathsf{s}} \langle\langle e \rangle\rangle : \mathsf{Code}(T^0 \,!\, \xi)^{-1} \,!\, \Delta}$$

The dual to $\langle\langle e \rangle\rangle$ is splice $\$e$, which *eliminates* compile-time ASTs by transforming them (back) into run-time code. This intuition is captured by the q-Splice and c-Splice rules, where, to verify that $\$e$ is a valid run-time program, we check that e is a valid compile-time AST of the corresponding type.

$$\text{(c-Splice)}\quad \frac{\Gamma \vdash_{\mathsf{s}} e : \mathsf{Code}(T^0 \,!\, \xi)^{-1} \,!\, \Delta}{\Gamma \vdash_{\mathsf{c}} \$e : T^0 \,!\, \Delta ; \xi} \qquad \text{(q-Splice)}\quad \frac{\Gamma \vdash_{\mathsf{s}} e : \mathsf{Code}(T^0 \,!\, \xi)^{-1} \,!\, \Delta}{\Gamma \vdash_{\mathsf{q}} \$e : T^0 \,!\, \Delta ; \xi}$$

Finally, note that since there are no q-Quote or s-Splice rules, the type system bans nested splices and quotations, and thus we can focus purely on levels 0 and -1.

3.2 The core language: $\lambda_{\mathsf{AST(op)}}$

I now describe the core language, $\lambda_{\mathsf{AST(op)}}$, which is a simple extension of λ_{op}. For clarity, I rename λ_{op} values (v) and compuations (c) to $\lambda_{\mathsf{AST(op)}}$ normal forms (n) and terms (t) respectively.

To construct $\lambda_{\mathsf{AST(op)}}$, I extend λ_{op} in two ways. First, I add machinery for metaprogramming:

c and q Typing Rules

$\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$

$$\text{(NAT)}\quad \frac{}{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} n : \mathbb{N}^0 \,!\, \Delta}$$

$$\text{(VAR)}\quad \frac{\Gamma(x) = T^0}{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} x : T^0 \,!\, \Delta}$$

$$\text{(LAMBDA)}\quad \frac{\Gamma, x : T_1^0 \vdash_{\mathsf{c}|\mathsf{q}} e : {T_2}^0 \,!\, \Delta; \xi}{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} \lambda x.e : (T_1^0 \xrightarrow{\xi} T_2^0)^0 \,!\, \Delta}$$

$$\text{(APP)}\quad \frac{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} v_1 : (T_1^0 \xrightarrow{\xi} T_2^0)^0 \,!\, \Delta \qquad \Gamma \vdash_{\mathsf{c}|\mathsf{q}} v_2 : {T_1}^0 \,!\, \Delta}{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} v_1 v_2 : {T_2}^0 \,!\, \Delta; \xi}$$

$$\text{(CONTINUE)}\quad \frac{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} v_1 : (T_1^0 \overset{\xi}{\twoheadrightarrow} T_2^0)^0 \,!\, \Delta \qquad \Gamma \vdash_{\mathsf{c}|\mathsf{q}} v_2 : {T_1}^0 \,!\, \Delta}{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} \texttt{continue}\, v_1\, v_2 : {T_2}^0 \,!\, \Delta; \xi}$$

$$\text{(RETURN)}\quad \frac{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} v : T^0 \,!\, \Delta}{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} \texttt{return}\ v : T^0 \,!\, \Delta; \xi}$$

$$\text{(DO)}\quad \frac{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} e_1 : {T_1}^0 \,!\, \Delta; \xi \qquad \Gamma, x : T_1^0 \vdash_{\mathsf{c}|\mathsf{q}} e_2 : {T_2}^0 \,!\, \Delta; \xi}{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} \texttt{do}\ x \leftarrow e_1\, \texttt{in}\, e_2 : {T_2}^0 \,!\, \Delta; \xi}$$

$$\text{(OP)}\quad \frac{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} v : {T_1}^0 \,!\, \Delta \qquad \mathsf{op} : T_1^0 \to T_2^0 \in \Sigma \qquad \mathsf{op} \in \xi}{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} \mathbf{op}(v) : {T_2}^0 \,!\, \Delta; \xi}$$

$$\text{(HANDLE)}\quad \frac{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} e : {T_1}^0 \,!\, \Delta; \xi \qquad \Gamma \vdash_{\mathsf{c}|\mathsf{q}} h : ({T_1}^0 \,!\, \xi \Longrightarrow {T_2}^0 \,!\, \xi') \,!\, \Delta \qquad \forall \mathsf{op} \in \Delta \setminus \Delta' \,.\, \mathsf{op} \in \mathsf{dom}(h)}{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} \texttt{handle}\ e\ \texttt{with}\ \{h\} : {T_2}^0 \,!\, \Delta; \xi'}$$

$$\text{(RET-HANDLER)}\quad \frac{\Gamma, x : T_1^0 \vdash_{\mathsf{c}|\mathsf{q}} e : {T_2}^0 \,!\, \Delta; \xi'}{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} \texttt{return}(x) \mapsto e : ({T_1}^0 \,!\, \xi \Longrightarrow {T_2}^0 \,!\, \xi') \,!\, \Delta}$$

$$\text{(OP-HANDLER)}\quad \frac{\begin{array}{c} \mathsf{op} : A^0 \to B^0 \in \Sigma \\ \Gamma \vdash_{\mathsf{c}|\mathsf{q}} h : ({T_1}^0 \,!\, \xi \Longrightarrow {T_2}^0 \,!\, \xi') \,!\, \Delta \\ \Gamma, x : A^0, k : (B^0 \overset{\xi'}{\twoheadrightarrow} T_2^0)^0 \vdash_{\mathsf{c}|\mathsf{q}} e : {T_2}^0 \,!\, \Delta; \xi' \qquad \xi \subseteq \xi \setminus \{\mathsf{op}\} \qquad \mathbf{op}(x', k') \mapsto e' \notin h \end{array}}{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} h; \mathbf{op}(x, k) \mapsto e : ({T_1}^0 \,!\, \xi \Longrightarrow {T_2}^0 \,!\, \xi') \,!\, \Delta}$$

$$\text{(c-SPLICE)}\quad \frac{\Gamma \vdash_{\mathsf{s}} e : \mathsf{Code}(T^0 \,!\, \xi)^{-1} \,!\, \Delta}{\Gamma \vdash_{\mathsf{c}} \$e : T^0 \,!\, \Delta; \xi}$$

$$\text{(q-SPLICE)}\quad \frac{\Gamma \vdash_{\mathsf{s}} e : \mathsf{Code}(T^0 \,!\, \xi)^{-1} \,!\, \Delta}{\Gamma \vdash_{\mathsf{q}} \$e : T^0 \,!\, \Delta; \xi}$$

Figure 3.5: The c-mode and q-mode typing rules for $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$. The rules are nearly identical to the λ_{op} typing rules, with the exception of level annotations on types, except that everything – including values like 1 – are typed as compile-time computations. Further, two additional rules, (c-SPLICE) and (q-SPLICE) formalise the transition to s-mode. The former is top-level splice and the latter is splice.

s Typing Rules

$\lambda_{\langle\langle\mathsf{op}\rangle\rangle}$

(s-Nat)
$$\frac{}{\Gamma \vdash_{\mathsf{s}} n : \mathbb{N}^{-1}}$$

(s-Var)
$$\frac{\Gamma(x) = T^{-1}}{\Gamma \vdash_{\mathsf{s}} x : T^{-1}}$$

(s-Function)
$$\frac{\Gamma, x : {T_1}^{-1} \vdash_{\mathsf{s}} c : {T_2}^{-1} \,!\, \Delta}{\Gamma \vdash_{\mathsf{s}} \kappa x.c : ({T_1}^{-1} \xrightarrow{\Delta} {T_2}^{-1})^{-1}}$$

(s-Continuation)
$$\frac{\Gamma, x : {T_1}^{-1} \vdash_{\mathsf{s}} c : {T_2}^{-1} \,!\, \Delta}{\Gamma \vdash_{\mathsf{s}} \kappa x.c : ({T_1}^{-1} \overset{\Delta}{\twoheadrightarrow} {T_2}^{-1})^{-1}}$$

(s-App)
$$\frac{\Gamma \vdash_{\mathsf{s}} v_1 : ({T_1}^{-1} \xrightarrow{\Delta} {T_2}^{-1})^{-1} \qquad \Gamma \vdash_{\mathsf{s}} v_2 : {T_1}^{-1}}{\Gamma \vdash_{\mathsf{s}} v_1\, v_2 : {T_2}^{-1} \,!\, \Delta}$$

(s-Continue)
$$\frac{\Gamma \vdash_{\mathsf{s}} v_1 : ({T_1}^{-1} \overset{\Delta}{\twoheadrightarrow} {T_2}^{-1})^{-1} \qquad \Gamma \vdash_{\mathsf{s}} v_2 : {T_1}^{-1}}{\Gamma \vdash_{\mathsf{s}} \mathbf{continue}\, v_1\, v_2 : {T_2}^{-1} \,!\, \Delta}$$

(s-Return)
$$\frac{\Gamma \vdash_{\mathsf{s}} v : T^{-1}}{\Gamma \vdash_{\mathsf{s}} \mathbf{return}\ v : T^{-1} \,!\, \Delta}$$

(s-Do)
$$\frac{\Gamma \vdash_{\mathsf{s}} c_1 : {T_1}^{-1} \,!\, \Delta \qquad \Gamma, x : T_1 \vdash_{\mathsf{s}} c_2 : {T_2}^{-1} \,!\, \Delta}{\Gamma \vdash_{\mathsf{s}} \mathbf{do}\ x \leftarrow c_1\ \mathbf{in}\ c_2 : {T_2}^{-1} \,!\, \Delta}$$

(s-Op)
$$\frac{\Gamma \vdash_{\mathsf{s}} v : {T_1}^{-1} \qquad \mathsf{op} : {T_1}^{-1} \to {T_2}^{-1} \in \Sigma \qquad \mathsf{op} \in \Delta}{\Gamma \vdash_{\mathsf{s}} \mathbf{op}(v) : {T_2}^{-1} \,!\, \Delta}$$

(s-Handle)
$$\frac{\Gamma \vdash_{\mathsf{s}} c : {T_1}^{-1} \,!\, \Delta \qquad \Gamma \vdash_{\mathsf{s}} h : ({T_1}^{-1} \,!\, \Delta \Longrightarrow {T_2}^{-1} \,!\, \Delta')^{-1} \qquad \forall \mathsf{op} \in \Delta \setminus \Delta'.\, \mathsf{op} \in \mathsf{dom}(h)}{\Gamma \vdash_{\mathsf{s}} \mathbf{handle}\ c\ \mathbf{with}\ \{h\} : {T_2}^{-1} \,!\, \Delta'}$$

(s-Ret-Handler)
$$\frac{\Gamma, x : {T_1}^{-1} \vdash_{\mathsf{s}} c : {T_2}^{-1} \,!\, \Delta'}{\Gamma \vdash_{\mathsf{s}} \mathbf{return}(x) \mapsto c : ({T_1}^{-1} \,!\, \Delta \Longrightarrow {T_2}^{-1} \,!\, \Delta')^{-1}}$$

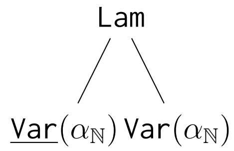
(s-Op-Handler)
$$\frac{\mathsf{op} : A^{-1} \to B^{-1} \in \Sigma \qquad \Gamma \vdash_{\mathsf{s}} h : ({T_1}^{-1} \,!\, \Delta \Longrightarrow {T_2}^{-1} \,!\, \Delta')^{-1} \qquad \Gamma, x : A^{-1}, k : (B^{-1} \overset{\Delta'}{\twoheadrightarrow} {T_2}^{-1})^{-1} \vdash_{\mathsf{s}} c : {T_2}^{-1} \,!\, \Delta' \qquad \Delta' \subseteq \Delta \setminus \{\mathsf{op}\} \qquad \mathbf{op}(x', k') \mapsto c' \notin h}{\Gamma \vdash_{\mathsf{s}} h; \mathbf{op}(x, k) \mapsto c : ({T_1}^{-1} \,!\, \Delta \Longrightarrow {T_2}^{-1} \,!\, \Delta')^{-1}}$$

(s-Quote)
$$\frac{\Gamma \vdash_{\mathsf{q}} e : T^0 \,!\, \Delta; \xi}{\Gamma \vdash_{\mathsf{s}} \langle\langle e \rangle\rangle : \mathsf{Code}(T^0 \,!\, \xi)^{-1} \,!\, \Delta}$$

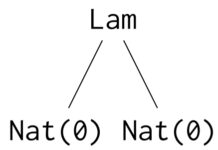
Figure 3.6: The s-mode typing rules for $\lambda_{\langle\langle\mathsf{op}\rangle\rangle}$. The rules are exactly identical to the λ_{op} typing rules, with the exception of level annotations on types, and the additional (Quote) rule, which makes level 0 code available at compile-time.

1. **AST** nodes (like Nat and Var) and **binders** ($\underline{\texttt{Var}}$)

   The syntax explicitly disambiguates between binders ($\underline{\texttt{Var}}$) and AST nodes. For example, consider the AST of $\lambda\alpha{:}\mathbb{N}.\,\alpha$, where I contrast the binder (left subtree) with the usage of the binder (right subtree).

   This syntactic distinction is important because Binders and ASTs need to be distinguished at the type level, and I wish to keep the typing rules syntax directed. To see why it is important to distinguish Binders and ASTs at the *type* level, consider the following malformed AST:

   The aforementioned malformed AST should be ill-typed. To do so, it is insufficient to require that the left sub-tree is of type $\mathsf{AST}(\mathbb{N})$. Thus, Binders and ASTs should be distinguished at the type level.

2. **binderToAST**, a primitive for turning binders ($\underline{\texttt{Var}}$) into *usages* of binders (Var)

   This allows us to write programs like the following:

   $\lambda_{\mathsf{AST(op)}}$

   $\textbf{do}\ x \leftarrow \underline{\texttt{Var}}(\alpha_{\mathbb{N}})\ \textbf{in}$
   $\textbf{do}\ \texttt{body} \leftarrow \textbf{binderToAST}\ x\ \textbf{in}$
   $\textbf{return}\ \texttt{Lam}(x, \texttt{body})$

   $\textbf{return}\ \texttt{Lam}(\underline{\texttt{Var}}(\alpha_{\mathbb{N}}), \texttt{Var}(\alpha_{\mathbb{N}}))$

3. **mkvar** T, a primitive for generating fresh binders of type T ($\underline{\texttt{Var}}$)

   To illustrate why **mkvar** is necessary, recall that $\lambda_{\mathsf{AST(op)}}$ acts as an elaboration target for $\lambda_{\langle\langle \mathsf{op}\rangle\rangle}$. Consider the following $\lambda_{\langle\langle \mathsf{op}\rangle\rangle}$ program, which should ideally generate the AST of $\lambda\alpha.\lambda\beta.\,\textbf{return}\ (\alpha, \beta)$

   $\lambda_{\langle\langle \mathsf{op}\rangle\rangle}$

   $\$(\textbf{do}\ \texttt{mkfun} \leftarrow \lambda k.\,\langle\langle\ \lambda x : \mathbb{N}^0.\ \$(k\ \langle\langle x\rangle\rangle)\ \rangle\rangle\ \textbf{in}$
   $\quad \texttt{mkfun}\ (\ \lambda a.\,\texttt{mkfun}\ (\lambda b.\,\textbf{return}\ (a, b)))\ )$

   $\textbf{return}\ \texttt{Lam}(\underline{\texttt{Var}}(\alpha_{\mathbb{N}}), \texttt{Lam}(\underline{\texttt{Var}}(\beta_{\mathbb{N}}), \texttt{Ret}(\texttt{Pair}(\texttt{Var}(\alpha_{\mathbb{N}}), \texttt{Var}(\beta_{\mathbb{N}})))))$

   The compile-time function mkfun is a higher-order function that takes some compile-time function k. k takes in a binder, x, and constructs the body of a function $\lambda x.\texttt{body}$. For example, in order to generate the body $x + 1$, we could write

   $$\texttt{mkfun}\ (\lambda a.\langle\langle \$(a) + 1\rangle\rangle)$$

Syntax $\lambda_{\mathsf{AST(op)}}$

$$
\begin{array}{llll}
\textbf{Normal Forms} & n & ::= & \ldots \mid \mathtt{Nat}(m) \mid \mathtt{Var}(\alpha_A) \mid \underline{\mathtt{Var}}(\alpha_A) \mid \mathtt{Lam}(n_1, n_2) \\
& & & \mid \mathtt{App}(n_1, n_2) \mid \mathtt{Continue}(n_1, n_2) \mid \mathtt{Ret}(n) \\
& & & \mid \mathtt{Do}(n_1, n_2, n_3) \mid \mathtt{Op}(n) \mid \mathtt{Hwith}(n_1, n_2) \\
& & & \mid \mathtt{Hret}(n_1, n_2)(n_1, n_2) \mid \mathtt{Hop}(n_1, n_2, n_3, n_4) \\
\textbf{Terms} & t & ::= & \ldots \mid \mathbf{check}\, n \mid \mathbf{check}_M\, n \mid \mathbf{dlet}(n, t) \mid \mathbf{tls}(t) \mid \mathbf{err} \\
& & & \mid \mathbf{binderToAST}\, n
\end{array}
$$

Figure 3.7: $\lambda_{\mathsf{AST(op)}}$ syntax. The syntax is broadly the same as λ_{op}, except with the addition of AST constructors and scope extrusion checking machinery.

In the example $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ program, k calls `mkfun`. This means that one constructs a *nested* function, whose formal parameters are bound to a and b respectively. If we simply elaborated x into $\underline{\mathtt{Var}}(x_{\mathbb{N}})$, we would bind both a and b to $\underline{\mathtt{Var}}(x_{\mathbb{N}})$, and generate the following (incorrect) AST:

$$\mathbf{return}\ \mathtt{Lam}(\underline{\mathtt{Var}}(x_{\mathbb{N}}), \mathtt{Lam}(\underline{\mathtt{Var}}(x_{\mathbb{N}}), \mathtt{Ret}(\mathtt{Pair}(\mathtt{Var}(x_{\mathbb{N}}), \mathtt{Var}(x_{\mathbb{N}})))))$$

The problem is that we do not have α-renaming "for free". We thus need to ensure that `mkfun` is elaborated into a function that generates *fresh* names for x each time it is called. This is the purpose of **`mkvar`**.

Second, I extend λ_{op} with machinery for scope extrusion checking. This comprises:

1. **`err`**, an error state for indicating the presence of scope extrusion,
2. **`check`**, a guarded **`return`** construct that either detects scope extrusion, and transitions to **`err`**, or does not detect scope extrusion, and transitions to **`return`**,
3. $\mathbf{check}_M$, which behaves similarly to **`check`**, but (for reasons I will explain in [LATER CHAPTER]) allows a set of *muted* variables M to *temporarily* extrude their scope,
4. **`dlet`**, a primitive for tracking which variables are well-scoped and which have extruded their scope,
5. **`tls`**, a marker for where top-level splices would have occurred in the source program. This is for *ease of reasoning only*.

Notice that, while the calculus the *machinery* for scope extrusion checking, it does not demand that one *use* it, or use it *properly*. Scope extrusion checking is not a language feature, but an algorithm one builds on top of the calculus.

3.2.1 Operational Semantics

I will now describe the operational semantics of $\lambda_{\mathsf{AST(op)}}$. Many rules are identical to those of λ_{op} (Figure 2.2): interesting rules are collated in Figure 3.8. Rules are divided into those related to AST construction (Ast-Rule), and those related to **s**cope **e**xtrusion **c**hecking (Sec-Rule). I will now explain key rules.

Operational Semantics

Selected Rules

$\lambda_{\mathsf{AST(op)}}$

(Ast-Sym)	$\texttt{mkvar}\ A; E; U; M; I$	$\rightarrow$	$\texttt{return}\ \underline{\mathsf{Var}}(\alpha_A); E; U :: \alpha; M; I$ (where $\alpha \notin U$)
(Ast-Use)	$\texttt{binderToAST}\ \underline{\mathsf{Var}}(\alpha_A); E; U; M; I$	$\rightarrow$	$\texttt{return}\ \mathsf{Var}(\alpha_A); E; U; M; I$
(Sec-Chs)	$\texttt{check}\ n; E; U; M; I$	$\rightarrow$	$\texttt{return}\ n; E; U; M; I$ (if $\mathsf{FV}^0(n) \subseteq \pi_{\mathsf{Var}}(E)$)
(Sec-Chf)	$\texttt{check}\ n; E; U; M; I$	$\rightarrow$	$\texttt{err}; E; U; M; I$ (if $\mathsf{FV}^0(n) \not\subseteq \pi_{\mathsf{Var}}(E)$)
(Sec-Cms)	$\texttt{check}_M\ n; E; U; M; I$	$\rightarrow$	$\texttt{return}\ n; E; U; M; I$ (if $\mathsf{FV}^0(n) \setminus M \subseteq \pi_{\mathsf{Var}}(E)$)
(Sec-Cmf)	$\texttt{check}_M\ n; E; U; M; I$	$\rightarrow$	$\texttt{err}; E; U; M; I$ (if $\mathsf{FV}^0(n) \setminus M \not\subseteq \pi_{\mathsf{Var}}(E)$)
(Sec-Tls)	$\texttt{tls}(\texttt{return}\ n); E; U; M; I$	$\rightarrow$	$\texttt{return}\ n; E; U; M'; I'$
(Sec-Dlt)	$\texttt{dlet}(\underline{\mathsf{Var}}(\alpha_T), \texttt{return}\ n); E; U; M; I$	$\rightarrow$	$\texttt{return}\ n; E; U; M'; I'$ (if $I < \mathsf{len}(E)$ then $M' = M, I' = I$ else $M' = \emptyset, I' = \top$)
(Eff-Op)	$\mathbf{op}(v); E_1[\texttt{handle}\ E_2\ \texttt{with}\ \{h\}]; U; M; I$	$\rightarrow$	$c[v/x, \mathsf{cont}/k]; E_1; U; M \cup \pi_{\mathsf{Var}}(E_2); I'$ (where $\mathsf{cont} = \kappa x.\,\texttt{handle}\ E_2[\texttt{return}\ x]\ \texttt{with}\ \{h\}$ and $\mathbf{op}(x,k) \mapsto c \in h$ and $\mathbf{op} \notin \mathsf{handled}(E_2)$ and $I' = I$ if $\mathsf{len(E)} < \mathsf{I}$ else $\mathsf{max}(\mathsf{idx}_{\mathsf{Var}}(E_1))$))

Figure 3.8: Selected rules of the $\lambda_{\mathsf{AST(op)}}$ operational semantics. Many of the rules can be trivially adapted from the λ_{op} semantics (Figure 2.2), and therefore have been omitted. Rules relating to the muting and unmuting of variables is complex, and will be best explained when we discuss scope extrusion checks. For now, these mechanisms are **highlighted**.

Configurations

Like λ_{op}, the operational semantics is defined over *configurations*. In $\lambda_{\mathsf{AST(op)}}$, configurations have the form:

$$\langle t; E; U; M; I \rangle$$

I will describe each element of the configuration at a high level: their roles will become clearer as we introduce each rule. t and E are as they are in λ_{op}: terms and evaluation contexts. U acts as a source of freshness for name generation. M is a set of muted variables, i.e. those that we do not want to trigger a scope extrusion error, even if they have extruded their scope. I indicates the point at which we should *unmute*, setting M to $\emptyset$.

AST Rules

The Ast-Gen rule describes the behaviour of **mkvar**: **mkvar** T produces a `Var` of type T and some name α. Recall that names should be **fresh**: that is, multiple calls to **mkvar** should always return variables with different names. In order to ensure that names are fresh, we need to keep track of names that have been previously generated. This is the purpose of U in the configuration, and the side condition on the rule. To ensure determinacy of the semantics, we will assume that names are chosen *deterministically*.

The other primitive, **binderToAST**, turns binders $\underline{\mathtt{Var}}(\alpha_T)$ into usages of the binder $\mathtt{Var}(\alpha_T)$ (Ast-Use).

Scope Extrusion Checking Rules

More interesting are the primitives for scope extrusion checking. The **check** primitive acts like a guarded **return**, which can catch occurrences of scope extrusion. For some arbitrary AST n, either:

1. All the free variables of n are properly scoped, so $\mathbf{check}\, n$ reduces to $\mathbf{return}\ n$ (Sec-Chs)

2. Some free variables of n are not properly scoped, so $\mathbf{check}\, n$ reduces to **err** (Sec-Chf)

What does it mean to be "properly scoped" (the side condition on Sec-Chs)? The answer is slightly subtle. Consider the following program

$$\mathbf{do}\ \mathtt{body} \leftarrow \mathbf{check}\, n\ \mathbf{in}\, \mathbf{check}\, \mathtt{Lam}(\mathtt{Var}(\alpha_{\mathbb{N}}), \mathtt{body})$$

I argue that $\mathtt{Var}(\alpha_{\mathbb{N}})$ **ought to be** properly scoped in n (should not cause a transition to **err**). However, it is hard to deduce this from the *static* structure of the program. Instead, one has to reason about the *dynamic* execution of the program. Rather than calculating what is properly scoped as a *language feature*, I defer it to the programmer. The programmer must *declare* that a variable is properly scoped through use of the **dlet** keyword.

$$\mathbf{dlet}(\underline{\mathtt{Var}}(\alpha_{\mathbb{N}}), \mathbf{do}\ \mathtt{body} \leftarrow \mathbf{check}\, n\ \mathbf{in}\, \mathbf{check}\, \mathtt{Lam}(\mathtt{Var}(\alpha_{\mathbb{N}}), \mathtt{body}))$$

More precisely, **dlet** places a frame of the form $\mathbf{dlet}(\underline{\mathtt{Var}}(\alpha_A), [-])$ on the evaluation context E. I use the notation $\pi_{\mathsf{Var}}(E)$ to filter out variables declared in this manner. For example,

$$\pi_{\mathsf{Var}}(\mathbf{dlet}(\underline{\mathtt{Var}}(\alpha_A), \mathbf{do}\ x \leftarrow [-]\ \mathbf{in}\, t)) = \{\mathtt{Var}(\alpha_A)\}$$

Given a term of the form $\mathbf{check}\, n$ in some evaluation context E, where n is an AST, **check** thus checks that the free Vars of n, written $\mathsf{FV}^0(n)$, have all been properly declared, or to be precise, a subset of $\pi_{\mathsf{Var}}(E)$.

It may seem lazy of me to define the semantics of **check** in such a way that places the burden onto the user. Recall, however, that $\lambda_{\mathsf{AST(op)}}$ is *not* meant to be programmed in directly. Rather, it acts as an elaboration target for $\lambda_{\langle\langle\mathsf{op}\rangle\rangle}$, and I define the elaboration. Therefore, I am the (only) $\lambda_{\mathsf{AST(op)}}$ user, and the onus is on me to justify that my elaboration uses **check** appropriately.

I also introduce $\mathbf{check}_M$ as a variant of **check**. As I will explain in [LATER CHAPTER], to design a good scope extrusion check, it is necessary to *mute* some variables, pretending that they are properly scoped. $\mathbf{check}_M$ will behave exactly like **check**, except that it will also pretend the muted variables M are properly scoped.

Similarly, when justifying the correctness of scope extrusion checks, it will be useful to remember the position of top-level splices in the $\lambda_{\langle\langle\mathsf{op}\rangle\rangle}$ source program. This is the purpose of **tls**, which should be interpreted as a no-op (Sec-Tls).

The final two rules, Sec-Dlt and Eff-Op, behave as normal, but additionally mute or unmute Vars. The operations of muting and unmuting are best explained in [THE

Types $\lambda_{\mathsf{AST(op)}}$

Computation and Handler Types omitted

Run-time Pre-types

Effects Row $\xi ::= \emptyset \mid \xi \cup \{\mathsf{op}_i\}$

Value type
$$
\begin{aligned}
A ::=\ & \mathbb{N} \\
\mid\ & A_1 \xrightarrow{\xi} A_2 && \text{functions} \\
\mid\ & A_1 \xtwoheadrightarrow{\xi} A_2 && \text{continuations}
\end{aligned}
$$

Computation type $A \,!\, \xi$

Handler type $A_1 \,!\, \xi \Longrightarrow A_2 \,!\, \xi'$

Types

Value type
$$
\begin{aligned}
T ::=\ & \ldots \\
\mid\ & \mathsf{Binder}(A) && \text{binders} \\
\mid\ & \mathsf{AST}(A) && \text{AST (value)} \\
\mid\ & \mathsf{AST}(A \,!\, \xi) && \text{AST (computation)} \\
\mid\ & \mathsf{AST}(A_1 \,!\, \xi \Longrightarrow A_2 \,!\, \xi') && \text{AST (handler)}
\end{aligned}
$$

Figure 3.9: The types of $\lambda_{\mathsf{AST(op)}}$. $\lambda_{\mathsf{AST(op)}}$ types extend λ_{op} types with an AST type (for ASTs), and a Binder type

NEXT CHAPER]. For now, they are **highlighted**, and can be mostly ignored. At a high level, when an operation is performed, we mute some set of variables, and potentially update the point at which they should be unmuted. When we remove a declared variable, we additionally check if we ought to unmute variables (and do so if we should).

3.2.2 Type System

Extending the types is similarly straightforward. I add only two types: a Binder type and an AST type. The types are summarised in Figure 3.9.

Typing Rules

I now describe a selection of $\lambda_{\mathsf{AST(op)}}$ typing rules (Figure 3.10). The rules are extremely straightforward: $\underline{\mathtt{Var}}(\alpha_A)$ is a Binder of type A, and $\mathtt{Var}(\alpha_A)$ is an AST of type A. **mkvar** A is a computation that produces Binders of type A, and $\mathtt{Lam}(n_1, n_2)$ is well-typed if n_1 is a Binder and n_2 an AST.

Notice that this type system does not guarantee that the resulting AST is *well-scoped*, for example, the following is well-typed:

$$\cdot \vdash \mathtt{Var}(\alpha_A) : \mathsf{AST}(A)$$

The typing rules for scope extrusion checks are even more straightforward: they are effectively invisible to the type system.

Typing Rules $\lambda_{\mathsf{AST(op)}}$

Selected Rules

$$\text{(Binder)}\ \frac{}{\Gamma \vdash \underline{\mathsf{Var}}(\alpha_A) : \mathsf{Binder}(A)} \qquad \text{(Variable)}\ \frac{}{\Gamma \vdash \mathsf{Var}(\alpha_A) : \mathsf{AST}(A)} \qquad \text{(Mkvar)}\ \frac{}{\Gamma \vdash \mathbf{mkvar}\, A : \mathsf{Binder}(A) \,!\, \Delta}$$

$$\text{(BinderToAST)}\ \frac{\Gamma \vdash n : \mathsf{Binder}(A)}{\Gamma \vdash \mathbf{binderToAST}\, n : \mathsf{AST}(A) \,!\, \Delta} \qquad \text{(Lambda-AST)}\ \frac{\Gamma \vdash n_1 : \mathsf{Binder}(A_1) \qquad \Gamma \vdash n_2 : \mathsf{AST}(A_2 \,!\, \xi)}{\Gamma \vdash \mathsf{Lam}(n_1, n_2) : \mathsf{AST}(A_1 \xrightarrow{\xi} A_2)}$$

$$\text{(Err)}\ \frac{}{\Gamma \vdash \mathbf{err} : T \,!\, \Delta} \qquad \text{(Tls)}\ \frac{\Gamma \vdash t : T \,!\, \Delta}{\Gamma \vdash \mathbf{tls}(t) : T \,!\, \Delta} \qquad \text{(DLet)}\ \frac{\Gamma \vdash n : \mathsf{Binder}(A) \qquad \Gamma \vdash t : T \,!\, \Delta}{\Gamma \vdash \mathbf{dlet}(n, t) : T \,!\, \Delta}$$

$$\text{(Check)}\ \frac{\Gamma \vdash n : T \qquad T = \mathsf{AST}(A) \vee \mathsf{AST}(A \,!\, \xi) \vee \mathsf{AST}(A_1 \,!\, \xi \Longrightarrow A_2 \,!\, \xi')}{\Gamma \vdash \mathbf{check}\, n : T \,!\, \Delta}$$

Figure 3.10: Selected $\lambda_{\mathsf{AST(op)}}$ typing rules

3.3 Elaboration from $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ to $\lambda_{\mathsf{AST(op)}}$

Having described both $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ and $\lambda_{\mathsf{AST(op)}}$, I will now describe a simple elaboration from $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ to $\lambda_{\mathsf{AST(op)}}$. This elaboration will be *simple* – it will not insert any dynamic scope extrusion checks.

The elaboration is defined on typing judgements: I elaborate a $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ judgement to a $\lambda_{\mathsf{AST(op)}}$ judgement. To do so, I define four elaborations: on effect rows, types, contexts, and terms. As it will be clear from context which elaboration is being referred to, I will abuse notation and write $[\![-]\!]$ for all four.

Elaborating effect rows will just be the identity, that is

$$\begin{aligned} [\![\Delta]\!] &= \Delta \\ [\![\xi]\!] &= \xi \end{aligned}$$

and I will not touch on them any further.

3.3.1 Elaborating Types

To define elaboration of types, it will be convenient to refer to a helper function, erase, that *erases* all of the level annotations (and elaborates effect rows). It is easy to define inductively, and I do not give a formal definition, but rather an example.

$$\mathsf{erase}((T_1^0 \xrightarrow{\xi} T_2^0)^0) = T_1 \xrightarrow{[\![\xi]\!]} T_2$$

I can now define elaboration of types easily. In a nutshell, level 0 types elaborate into AST types. Level -1 types elaborate into themselves (sans level annotations), except for Code types, which elaborate into AST types.

$$
\begin{aligned}
[\![T^0]\!] &= \mathsf{AST}(\mathsf{erase}(T^0)) \\
[\![T^0 \,!\, \xi]\!] &= \mathsf{AST}(\mathsf{erase}(T^0 \,!\, \xi)) \\
[\![T^0 \,!\, \Delta]\!] &= \mathsf{AST}(\mathsf{erase}(T^0)) \,!\, [\![\Delta]\!] \\
[\![T^0 \,!\, \Delta; \xi]\!] &= \mathsf{AST}(\mathsf{erase}(T^0 \,!\, \xi)) \,!\, [\![\Delta]\!] \\
[\![(T_1^0 \,!\, \xi \Longrightarrow T_2^0 \,!\, \xi')^0]\!] &= \mathsf{AST}(\mathsf{erase}((T_1^0 \,!\, \xi \Longrightarrow T_2^0 \,!\, \xi'))^0)
\end{aligned}
$$

$$
\begin{aligned}
[\![T^{-1}]\!] &= \text{if}\, T \neq \mathsf{Code}(T^0 \,!\, \xi) \,\text{then}\, \mathsf{erase}(T^{-1}) \\
&\ \text{else}\, \mathsf{AST}(\mathsf{erase}(T^0 \,!\, \xi)) \\
[\![T^{-1} \,!\, \Delta]\!] &= [\![T^{-1}]\!] \,!\, [\![\Delta]\!]
\end{aligned}
$$

3.3.2 Elaborating Contexts

Elaborating contexts is *slightly* subtle. Rather than mapping the elaboration function over all types in the context, we treat the level 0 types slightly differently, elaborating in Binder, rather than AST types.

$$
\begin{aligned}
[\![\cdot]\!] &= \cdot \\
[\![\Gamma, x : T^0]\!] &= [\![\Gamma]\!], x : \mathsf{Binder}(\mathsf{erase}(T^0)) \\
[\![\Gamma, x : T^{-1}]\!] &= [\![\Gamma]\!], x : [\![T^{-1}]\!]
\end{aligned}
$$

To see why this is the case, notice that the only cases where the context Γ is extended with a level 0 variable $x : T^0$ occur in **c** or **q**. In these modes, we are building ASTs, and thus x must be an AST Binder.

3.3.3 Elaborating Terms

Elaborating terms is slightly more involved. To start, we will assume that we have annotated all binders with their types, for example

$$\lambda x : \mathbb{N}^0 .\, e$$

The elaboration for terms is moderated by the **mode**: **c**, **q**, or **s**. Selected rules are collated in Figure 3.11.

At a high level, in **c** and **q**-mode, one builds ASTs, calling **mkvar** when binders are encountered (see earlier discussion on the necessity of **mkvar**). Elaboration in **c** and **q**-modes do not differ particularly significantly, with the exception of the rule for splice, where in **c**, I insert the marker for the top-level splice, and in **q**, I do not. Elaboration in **c** and **q**-modes will be further distinguished when I extend the elaboration to insert scope extrusion checks. Elaboration in **s** is effectively the identity.

3.3.4 Elaborating Typing Judgements

We may now elaborate full typing judgements, by elaborating each component in the judgement. For example, take the typing judgement for lambdas in **c**-mode.

$$\frac{\Gamma, x : T_1^0 \vdash_{\mathsf{c}} e : {T_2}^0 \,!\, \Delta; \xi}{\Gamma \vdash_{\mathsf{c}} \lambda x.e : (T_1^0 \xrightarrow{\xi} T_2^0)^0 \,!\, \Delta}$$

Term Elaboration $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$

Selected Rules

$$
\begin{array}{lcl}
[\![x]\!]_{\mathsf{c}|\mathsf{q}} & = & \mathtt{binderToAST}\ x \\
[\![\lambda x : T^0 .\ e]\!]_{\mathsf{c}|\mathsf{q}} & = & \mathtt{do}\ x \leftarrow \mathtt{mkvar}\ \mathsf{erase}(T^0)\ \mathtt{in}\ \mathtt{do}\ \mathsf{body} \leftarrow [\![e]\!]_{\mathsf{c}|\mathsf{q}}\ \mathtt{in}\ \mathtt{return}\ \mathsf{Lam}(x, \mathsf{body}) \\
\\
[\![\$e]\!]_{\mathsf{c}} & = & \mathtt{tls}([\![e]\!]_{\mathsf{s}}) \\
[\![\$e]\!]_{\mathsf{q}} & = & [\![e]\!]_{\mathsf{s}} \\
\\
[\![x]\!]_{\mathsf{s}} & = & x \\
[\![\lambda x : T^0 .\ e]\!]_{\mathsf{s}} & = & \lambda x . [\![e]\!]_{\mathsf{s}} \\
[\![\langle\langle e \rangle\rangle]\!] & = & [\![e]\!]_{\mathsf{q}}
\end{array}
$$

Figure 3.11: Selected term elaboration rules from $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ to $\lambda_{\mathsf{AST}(\mathsf{op})}$. Elaboration is moderated by the compiler mode. In **c** and **q**, elaboration builds ASTs. While mostly identical, they differ in the elaboration for $\$e$. In **c**, one additionally inserts a marker representing the position of the top-level splice. **c** and **q** will be further distinguished when elaboration is extended to insert scope extrusion checks. In **s**, elaboration is effectively the identity.

which we elaborate by applying the elaboration component-wise

$$
\frac{[\![\Gamma, x : T_1^0]\!] \vdash [\![e]\!]_{\mathsf{c}} : [\![{T_2}^0 \,!\, \Delta; \xi]\!]}{[\![\Gamma]\!] \vdash [\![\lambda x . e]\!]_{\mathsf{c}} : [\![(T_1^0 \xrightarrow{\xi} T_2^0)^0 \,!\, \Delta]\!]}
$$

Letting $A_i = \mathsf{erase}(T_i^0)$, and $[\![e]\!]_{\mathsf{c}} = t$, and applying the elaboration functions defined above, we obtain

$$
\frac{[\![\Gamma]\!], x : \mathsf{Binder}(A_1) \vdash t : \mathsf{AST}(A_2 \,!\, \xi) \,!\, \Delta}{[\![\Gamma]\!] \vdash \mathtt{do}\ x \leftarrow \mathtt{mkvar}\ \mathsf{erase}(T^0)\ \mathtt{in}\ \mathtt{do}\ \mathsf{body} \leftarrow t\ \mathtt{in}\ \mathtt{return}\ \mathsf{Lam}(x, \mathsf{body}) : \mathsf{AST}(A_1 \xrightarrow{\xi} A_2) \,!\, \Delta}
$$

which one can verify corresponds to a valid $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ typing derivation.

Is this true in general? Do $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ typing derivations always elaborate into $\lambda_{\mathsf{AST}(\mathsf{op})}$ typing derivations? Yes, but the question begets a larger point: what properties can we claim (and not claim) about the calculus as defined? What metatheoretic results may we establish?

3.4 Metatheory

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] A. Bauer and M. Pretnar. An effect system for algebraic effects and handlers. *Logical Methods in Computer Science*, Volume 10, Issue 4, Dec. 2014. ISSN 1860-5974. doi: 10.2168/lmcs-10(4:9)2014. URL http://dx.doi.org/10.2168/LMCS-10(4:9)2014.

[3] 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.

[4] T.-J. Chiang, J. Yallop, L. White, and N. Xie. Staged compilation with module functors. *Proc. ACM Program. Lang.*, 8(ICFP), Aug. 2024. doi: 10.1145/3674649. URL https://doi.org/10.1145/3674649.

[5] M. Felleisen and D. P. Friedman. Control operators, the secd-machine, and the λ-calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III, Ebberup, Denmark, 25-28 August 1986*, pages 193–222. North-Holland, 1987.

[6] D. Hillerström and S. Lindley. Shallow effect handlers. In S. Ryu, editor, *Programming Languages and Systems*, pages 415–435, Cham, 2018. Springer International Publishing. ISBN 978-3-030-02768-1.

[7] J. Inoue and W. Taha. Reasoning about multi-stage programs. In H. Seidl, editor, *Programming Languages and Systems*, pages 357–376, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-28869-2.

[8] 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.

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

[10] O. Kiselyov. Delimited control in ocaml, abstractly and concretely. *Theoretical Computer Science*, 435:56–76, 2012. ISSN 0304-3975. doi: https://doi.org/10.1016/j.tcs.2012.02.025. URL https://www.sciencedirect.com/science/article/pii/S0304397512001661. Functional and Logic Programming.

[11] 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.

[12] O. Kiselyov. Generating c: Heterogeneous metaprogramming system description. *Science of Computer Programming*, 231:103015, 2024. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2023.103015. URL https://www.sciencedirect.com/science/article/pii/S0167642323000977.

[13] 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.

[14] P. Levy, J. Power, and H. Thielecke. Modelling environments in call-by-value programming languages. *Information and Computation*, 185(2):182–210, 2003. ISSN 0890-5401. doi: https://doi.org/10.1016/S0890-5401(03)00088-9. URL https://www.sciencedirect.com/science/article/pii/S0890540103000889.

[15] G. Mainland. Explicitly heterogeneous metaprogramming with metahaskell. *SIGPLAN Not.*, 47(9):311–322, Sept. 2012. ISSN 0362-1340. doi: 10.1145/2398856.2364572. URL https://doi.org/10.1145/2398856.2364572.

[16] L. Phipps-Costin, A. Rossberg, A. Guha, D. Leijen, D. Hillerström, K. Sivaramakrishnan, M. Pretnar, and S. Lindley. Continuing webassembly with effect handlers. *Proc. ACM Program. Lang.*, 7(OOPSLA2), Oct. 2023. doi: 10.1145/3622814. URL https://doi.org/10.1145/3622814.

[17] 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).

[18] 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.

[19] 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.

[20] 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.

[21] 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 Com-

puting Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454039. URL https://doi.org/10.1145/3453483.3454039.

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

[23] 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.

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

[25] N. Xie, Y. Cong, K. Ikemori, and D. Leijen. First-class names for effect handlers. *Proc. ACM Program. Lang.*, 6(OOPSLA2), Oct. 2022. doi: 10.1145/3563289. URL https://doi.org/10.1145/3563289.

[26] N. Xie, L. White, O. Nicole, and J. Yallop. Macocaml: Staging composable and compilable macros. *Proc. ACM Program. Lang.*, 7(ICFP), Aug. 2023. doi: 10.1145/3607851. URL https://doi.org/10.1145/3607851.

[27] 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.