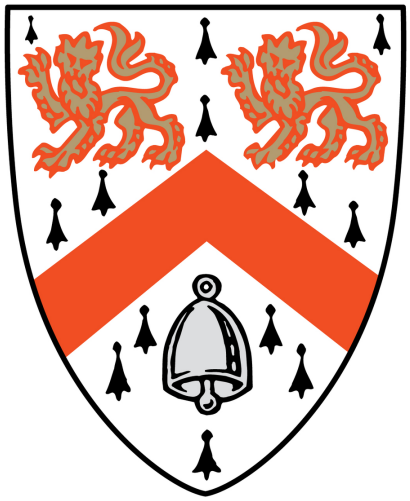
Michael Jing Long Lee



Wolfson College
University of Cambridge

Everything in Scope

A Comprehensive Comparison of Scope Extrusion Checks

This dissertation is submitted in partial fulfilment of the requirements for
Master of Philosophy in Advanced Computer Science

June 9, 2025

Declaration

I, *Michael Jing Long Lee* of *Wolfson College*, being a candidate for the Master of Philosophy in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this report, I adhered to the Department of Computer Science and Technology AI Policy. I am content for my report to be made available to the students and staff of the University.

Signed Michael Jing Long Lee
Date June 9, 2025

Acknowledgements

I am extremely grateful to:

- My supervisor, *Dr. Jeremy Yallop*, for his constant support and guidance (both academic and non-academic) throughout this project.
- The MacoCaml team, *Dr. Ningning Xie*, *Dima Szamozvancev*, *LT Stockmann*, and *Maite Kramarz*, for providing valuable comments and feedback over the course of the project.
- *Dr. Oleg Kiselyov*, for introducing me to the problem of scope extrusion and providing the initial inspiration behind the Best-Effort dynamic check.
- *Dr. Neel Krishnaswami*, for helping me refine an initial version of the Proof of Correctness for Refined Environment Classifiers.
- *Alistair John O'Brien*, *Dima Szamozvancev*, *Jacob Emilio Bennett-Woolf* (Jeb), *Théo Chengkai Wang*, and *Yulong Huang*, for their support, advice, and kinship.

Abstract

Everything in Scope: A Comprehensive Comparison of Scope Extrusion Checks

Metaprogramming and effect handlers interact in unexpected, and sometimes undesirable, ways. One example is scope extrusion: the generation of ill-scoped code. There are many different ways to solve the problem of scope extrusion, but to my knowledge, until now there has been no way to evaluate them against each other. This dissertation introduces such a mechanism, which I use to evaluate the correctness, expressiveness, and efficiency of existing scope extrusion solutions. Additionally, I introduce a novel solution, a dynamic Best-Effort check, that I show is correct, and occupies a goldilocks zone between expressiveness and efficiency.

Contents

1. Introduction

Many compilers perform a wide range of optimisations. Compiler optimisations empower programmers to focus on writing maintainable code, leaving the compiler to optimise said code.

While compiler optimisation is often all one needs, it is difficult for compiler engineers to ensure, *a priori*, that every user has all the optimisations they require [26, 27]. When users discover that their desired optimisation has not been implemented, what can they do, beyond submitting a pull request, or optimising their code by hand?

Metaprogramming (for example, C++ templates) [1] allows the user to "teach the compiler a class of [new] tricks" [30], and is therefore one possible solution. Metaprogramming allows programmers to write **maintainable** code which directs the compiler to generate **efficient** code.

For example, assume a `memcpy` function whose performance is heavily system dependent. How would one write a function that picks, depending on the system, the most performant implementation?

One possibility is via a dynamic conditional:

OCaml
```
let memcpy source target n = if (architecture == AArch64) then
                               (* return this program *)
                             else
                                match vector_extensions with
                                | None -> (* return this program *)
                                | Some(AVX512) -> (* return this program *)
                                | ...
memcpy source target n
```

Since `architecture` and `vector_extensions` are known at compile-time, they may be eliminated by an optimising compiler. If the compiler does not perform this optimisation, programmers might have to duplicate their code, increasing maintenance costs. Alternatively, they may use pre-processing directives, though this may lead to awkward code structures. With metaprogramming, however, they may write:

MacoCaml
```
(* macro and $ can be read as annotations which
   indicate that the code should be run at compile-time *)
macro memcpy source target n = if (architecture == AArch64) then
                               (* generate this program, to be run later *)
                               else
                                 match vector_extensions with
                                 | None -> (* generate this program *)
                                 | Some(AVX512) -> (* generate this program *)
                                 | ...
$(memcpy <<source>> <<target>> <<n>>)
```

Importantly, the `memcpy` macro is executed by the compiler, guaranteeing that the branches

will be eliminated. This process of using metaprogramming to select the most performant `memcpy` on any given system is used in practice, for instance, by the `xine` media player [10].

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 (ASTs). The following code constructs the program $(\lambda x.x)1$.

```OCaml
if (architecture == AArch64) then
  let build_app (f: (int -> int) expr) (v: int expr) = App(f, v) in
  let id_ast: (int -> int) expr = Lam(Var(x), Var(x)) in
  build_app(id_ast, Int(1)) (* int expr *)
else
  (*...*)
```

Metaprogramming is known to interact unexpectedly with effects. This interaction can be extremely desirable, for example, allowing programmers to manually perform loop-invariant code-motion [16] without making major modifications to their existing code [21].

Unfortunately, this interaction can also be undesirable. One problem is *scope extrusion* [16]. Scope extrusion occurs when the programmer accidentally generates code with unbound variables. In the following program, effects and metaprogramming combine to extrude `Var(x)` beyond its scope. The program stores `Var(x)` in a heap cell `(1)`, and subsequently retrieves it outside its scope. The program 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
```

This example illustrates how the state effect could cause scope extrusion. The choice of effect is arbitrary. Scope extrusion could be caused by many other effects, like resumable exceptions.

To parameterise over the specific effect, I turn to effect handlers. Effect handlers, which have recently been added to `OCaml` [31], are a powerful language construct that can simulate many effects (state, I/O, greenthreading) [25]. It is thus strategic, and timely, to study scope extrusion in the context of effect handlers.

The problem of scope extrusion has been widely studied, resulting in multitudinous mechanisms for managing the interaction between metaprogramming and effects. Some solutions adapt the type system (Refined Environment Classifiers [19, 12], Closed Types [5]), others insert dynamic checks into the generated code [16]. However, there are two issues.

First, I am not studying scope extrusion abstractly. The MacoCaml [37] project aims to extend the `OCaml` programming language, which has effect handlers, with compile-time metaprogramming. However, until now, the MacoCaml project has had no clear policy on how metaprogramming and effect handlers should interact. The context adds constraints: type-based approaches require unfeasible modification of the `OCaml` typechecker, 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 occurs, or unpredictable, allowing some programs, but dis-

allowing morally equivalent programs. Further, many of the dynamic solutions have not been proved correct.

Second, and more importantly, I lacked a way to evaluate solutions fairly and holistically. My evaluation criterion 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 own definition, but wrong with respect to another. Further, expressiveness and efficiency are inherently comparative criteria.

This thesis solves both these problems. In Chapter 3, I design a novel, common language for encoding and evaluating different solutions. In Chapter 4, I use the designed language to formally describe and evaluate various checks. This process led to a novel Best-Effort dynamic check (Section 4.4), which I argue lands in a goldilocks zone, and should be adopted by MacoCaml.

1.1. Contributions

Concretely, the contributions of this work are:

1. A novel two-stage calculus, $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$, that allows for effect handlers at both stages: compile-time and run-time (Chapter 3). $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ is well-typed, has the expected metatheoretic properties (Section 3.4), and is designed to facilitate comparative evaluation of different scope extrusion checks (Chapter 4).

2. In $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$, a formal description of the MetaOCaml check described by Kiselyov [16] (Section 2.3.1), as well as an evaluation of its correctness (Section 4.3.1), expressiveness (Section 4.3.2), and efficiency (Section 4.3.3).

3. In $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$, a formal description of a novel Dynamic Best-Effort check (Section 4.4), which I prove correct (Section 4.4.1), and argue, by comparison with other checks, lands in a goldilocks zone between expressiveness (Section 4.4.2) and efficiency (Section 4.4.3).

4. An encoding of refined environment classifiers [19] into $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$, with a proof of correctness via logical relation (Section 4.5.1), and an evaluation of its expressiveness compared to other checks (Section 4.5.2).

5. Implementations of the three dynamic checks in MacoCaml.

2. Background

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

This chapter provides 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? Section 2.1.1 first motivates metaprogramming, by illustrating the challenge of writing code that is both fast and maintainable. Following this, Section 2.1.2 considers the design space of metaprogramming, highlighting decisions made by the MacoCaml designers.

2.1.1. Metaprogramming for Fast and Maintainable Code

Metaprogramming helps programmers write fast and maintainable code. Maintainability and efficiency are in constant tension; Resolving this tension may require more than programmer skill.

To observe this tension, consider an example from the `JAX` machine learning library (adapted to `OCaml`): computing the gradient of a differentiable function. More precisely, assume a type `diff` of differentiable functions (for simplicity, comprising only polynomials and composition)

```OCaml
type diff = Poly of int list
          | Compose of diff * diff
```

For example, the following expression represents $2(x^2 + 2x) + 4$.

```OCaml
Compose(Poly([1 ; 2 ; 0]), Poly([2 ; 4]))
```

The `app: diff -> int -> int` function evaluates elements of type `diff`.

```OCaml
let rec app (f: diff) (x: int) =
  let rec app_poly (cs: int list) (x : int) (acc: int) = match cs with
    | [] -> acc
    | c::cs -> app_poly cs x (c + (x * acc))
  in match f with
    | Poly(cs)      -> match cs with
                       | [] -> 0
                       | c::cs -> app_poly cs x c
    | Compose(g, h) -> app h (app g x)
```

The challenge is to write a function `grad: diff -> int -> int` that computes the gradient of its first argument with respect to its second. The `grad_main` function (Listing 1) is one maintainable way to compute gradients:

```OCaml
let rec grad_main (f: diff) (x: int) =
  let grad_poly (cs: int list) (x: int) =
    let grad_p c (a, b) = (c * b :: a, b + 1)
    let (res, _) = List.fold_right grad_p cs ([], 0) in
    app Poly(remove_last res) x (*remove_last removes the last element of a list*)
  in match f with
    | Poly(cs)      -> grad_poly cs x
    | Compose(g, h) -> grad_main g x * grad_main h (app g x)
```

Listing 1: A maintainable implementation of `grad`

If the function to be differentiated (`f`) is known in advance [1], for example, $\texttt{f} = 2(x^2 + 2x) + 4$, then this approach is inefficient. Instead, `grad` could be specialised to a more efficient `grad_spec` function (Listing 2), whose body is simply a hardcoded equation:

```OCaml
let grad_spec x = 4 * x + 4
```

Listing 2: An implementation of `grad`, specialised to $\texttt{f} = 2(x^2 + 2x) + 4$

What if there were multiple such `f`s? We could use `grad_main` to parameterise over the possible `f`s, relying on one implementation. Abstraction centralises implementations, reducing maintenance costs, but results in inefficiency. Alternatively, we could hardcode one variant of `grad_spec` for each `f`, creating an army of efficient functions. Specialisation applies known arguments in advance, creating opportunities for optimisation, but at the cost of maintainability.

The tension between maintainability (abstraction) and efficiency (specialisation) has been observed in other works on metaprogramming, in domains such as regex matching [33], parsing [40], linking [29], statistical modelling [36], and hardware design [34].

A better approach might therefore be to write maintainable code, delegating the responsibility of optimisation to the compiler. While this suffices for many cases, relying solely on compiler optimisation can be insufficient. The compiler may not implement the desired optimisations. While compiler engineers might have an economic incentive to write optimisations for machine learning, this may not be true for less lucrative domains [27]. Even in machine learning, many libraries are built on top of existing languages, like `Python`, which might not perform the desired optimisations. Further, even if the compiler does implement the optimisation, the programmer has little control over the optimisation process: the compiler may not optimise as frequently as desired, or optimise in ways that do not meet all of the programmer's desiderata (for example, by inflating binary sizes). Moreover, the optimisation process may be sensitive to small changes in the source code.

How does one write maintainable and efficient code, **when one is not certain that the compiler will optimise one's code exactly as desired**?

One answer, and the approach taken by `JAX` [13], is metaprogramming, which gives

[1] e.g. `f` is the computation graph of a neural network

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 Mechanics of Metaprogramming

Metaprogramming allows for code that, when executed, generates code. Thus, it can be utilised to write a function, `grad_gen`, which resembles `grad_main` (inheriting its maintainability), but that generates a program which resembles `grad_spec` (inheriting its efficiency).

Listing 3 presents the metaprogrammed `grad_gen` function. Notice that it is exactly `grad_main`, but extended with annotations: **`macro`**, `<<->>`, and `$-`. Further, the type of `x` (e.g. on line 1) is now **`int expr`**, rather than **`int`**. These changes provide the necessary mechanisms for code generation: on line 20, `grad_gen` is used to generate `(2 + (y*2)) * 2`.

MacoCaml

```
1  macro rec app_gen (f: diff) (x: int expr) =
2    let rec app_poly_gen (cs: int list) (x : int expr) (acc: int expr) = match cs with
3      | [] -> acc
4      | c::cs -> app_poly_gen cs x <<c + ($x * $acc)>>
5    in match f with
6      | Poly(cs)       -> match cs with
7                          | [] -> <<0>>
8                          | c::cs -> app_poly cs x <<c>>
9      | Compose(g, h) -> app_gen h (app_gen g x)
10
11 macro rec grad_gen (f: diff) (x: int expr) =
12   let grad_poly_gen (cs: int list) (x: int expr) =
13     let grad_p c (a, b) = (c * b :: a, b + 1)
14     let (res, _) = List.fold_right grad_p cs ([], 0) in
15     app_gen Poly(remove_last res) x
16   in match f with
17     | Poly(cs)       -> grad_poly_gen cs x
18     | Compose(g, h) -> << $(grad_main g <<x>>) * $(grad_main h (app_gen g <<x>>)) >>
19
20 let grad_spec y = $(grad_gen Compose(Poly([1 ; 2 ; 0]), Poly([2 ; 4])) <<y>>)
21                   (* generates (2 + (y * 2)) * 2 *)
```

Listing 3: A metaprogrammed `grad_gen` function, which resembles `grad_main` but generates a function resembling `grad_spec`

In MacoCaml, the programmer is able to generate code at compile-time, for use at run-time. I separate this into two language features:

1. A type for code (`'a expr`), with mechanisms (`<<->>`, `$-`) for creating and manipulating values of this type. Expressions that return values of code type serve as code generators.

2. A mechanism for executing expressions *at compile-time*. By using the type-system, MacoCaml ensures only code generators can be executed at compile-time. Thus compile-time evaluation will always produce code values that can be evaluated at run-time (as opposed to **`int`**s, for example, which cannot).

First, in MacoCaml, code of type `'a` has type `'a expr`. For example, code of type `int` has type `int expr`. In MacoCaml, the `<<->>` ("quote") annotation converts expressions to code values (similar to how `[-]` converts expressions to list values). For example, `<<1>>` has type `int expr`. Under a quotation, the `$-` ("splice") annotation stops this conversion, allowing for evaluation under a quotation. For example,

```
<<$(print_int 1+2; <<1+2>>) + 0 >>
```

prints `3` and evaluates to `<<1+2+0>>`.

While not an accurate description of MacoCaml, it can be useful to think of elements of type `'a expr` as ASTs. In this conceptual model, quotation creates ASTs, by converting a program into its AST representation. For example,

`<< x + 0 >>` can be thought of as `Plus(Var(x), Int(0))`

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

`<< $x + 0 >>` can be thought of as `Plus(x, Int(0))`

Interleaving quotes and splices executes code to build ASTs

```
<<$(print_int 1+2; <<1+2>>) + 0>>
```
can be thought of as
```
Plus(print_int 1+2; Plus(Int(1), Int(2)), Int(0))
```

Second, to evaluate expressions at compile-time, MacoCaml offers the top-level splice, a splice (`$`) annotation not surrounded by quotes (`<<>>`). Notice that `$` 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 the code type. In Listing 3, there is only one top-level splice, on line 20: `$(grad_gen ...)`. We may now shift expressions of type `'a expr` under top-level splices, to perform generation at compile time.

Note that to access `grad_gen` at compile-time, we must also move it under the top-level splice, resulting in code duplication:

MacoCaml
```
let grad_spec y  = $(let grad_gen = ... in
                     grad_gen Compose(Poly([1 ; 2 ; 0]), Poly([2 ; 4])) <<y>>)
let grad_spec' y = $(let grad_gen = ... in
                     grad_gen Compose(Poly([4; 3 ; 1 ; 7]), Poly([1 ; 8])) <<y>>)
```

To allow compile-time functions, like `grad_gen`, to be re-used across multiple top-level splices, MacoCaml introduces the `macro` keyword:

MacoCaml
```
macro grad_gen = ...
let grad_spec y  = $(grad_gen Compose(Poly([1 ; 2 ; 0]), Poly([2 ; 4])) <<y>>)
let grad_spec' y = $(grad_gen Compose(Poly([4; 3 ; 1 ; 7]), Poly([1 ; 8])) <<y>>)
```

2.1.2. The Design Space of Metalanguages

Different metalanguages provide slightly different variants of metaprogramming to the user. Different variants could interact differently with effect handlers. This thesis focuses on homogenous, compile-time, two-stage metaprogramming:

1. **Homogenous or Heterogenous**
   Do the generated and generating languages agree or differ?

   If the generated and generating languages are the same, this is known as homogenous metaprogramming. Otherwise, it is heterogenous [17].

   The metaprogramming supported by MacoCaml is homogenous, where `OCaml` code generates `OCaml` code. I focus on homogenous metaprogramming.

2. **Run-time or Compile-Time**
   When does the generation take place?

   Code generation could take place at compile-time or at run-time.

   Run-time and compile-time metaprogramming differ non-trivially. For example, with run-time metaprogramming, generated and generating programs may share a heap.

   In MacoCaml, code generation occurs at compile-time, and I 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 shift computation from run-time ("level 0") to compile-time ("level -1"). I describe levels formally in Section 3.1.1 (Definition 3.1.1, Page 27). For now, it suffices to think of a level as a phase of evaluation: everything at level -1 must be fully evaluated before anything at level 0 is evaluated. Might it be possible to shift computation from compile-time to a pre-compile-time ("level -2") phase, for example, via a nested splice?

   ```
   1  $($ grad_gen f <<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 [38]. However, I focus on two-stage metaprogramming.

The restriction from multi-stage to two-stage metaprogramming 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 [7].

2. **Benefit**: In practice, "almost all uses" of multi-stage metaprogramming only use two stages [11]. Further, scope extrusion can be observed, and is often studied, in two-stage systems [12, 19].

2.2. Effect Handlers

What is an effect handler? Section 2.2.1 first motivates effect handlers, as a way to build composable and customisable effects. Section 2.2.2 introduces a calculus for studying the operational behaviour of effect handlers, à la Pretnar [25]. This calculus is useful both for precise description of effect handlers, and as a basis for investigating the interaction between metaprogramming and effect handlers. Finally, since different design decisions for effect handlers could affect the nature of their interaction with metaprogramming, Section 2.2.3 considers the design space of effect handlers.

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. Different effects are typically defined and understood separately from each other, meaning they are not easily composable. They are also typically implemented by compiler engineers rather than programmers, meaning they are not customisable. In contrast, effect handlers provide a unifying (composable) and programmable (customisable) framework that may be instantiated into different effects [15].

Much like how exception handlers allow the programmer to define custom exceptions that can be handled differently, 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 as effect handlers ensures composability by design.

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` [31], making their interaction with metaprogramming a timely problem.

2.2.2. λ_{op}: A Calculus for Effect Handlers

This section presents a calculus, λ_{op}, for reasoning about the operational behaviour of effect handlers. λ_{op} is broadly similar to the calculus described by Pretnar [25] [2], and will be used in later sections to study the interaction between effect handlers and metaprogramming.

Figure 2.1 collates the base syntax of λ_{op}. In this section, I additionally assume λ_{op} additionally supports a unit value $()$, pairs with pattern matching, strings with concatenation and `Python`-style format strings.

For example, treating **do** $x \leftarrow c_1$ **in** c_2 as a let-binding (and ignoring **return**, which is explained shortly), the following code evaluates to `"Revolution 9"`:

[2] Differences will be clarified as they arise

Syntax λ_{op}

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

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

λ_{op}

```
do (x, y) ← return ("Revolution", f"{9}") in x ^ y
```

```
return "Revolution 9"
```

Further, I use $c_1; c_2$ as syntactic sugar for **do** _ ← c_1 **in** c_2. This section explains key language constructs in turn, with reference to a running example: the λ_{op} program in Listing 4.

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")
```

Listing 4: A λ_{op} program that returns `(3, "1;2")`. It is used as a running example throughout this section.

Sequencing computations: `do` and `return`

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

OCaml

```
1 let pure      = (1+0) + (2+0)
2 let effectful = let l = ref 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} follows Pretnar's approach of stratifying terms into distinct syntactic categories, with "inert values" (v) disjoint from "potentially effectful computations" (c). However, while Pretnar treats handlers h as values, in λ_{op}, they are a third syntactic category, disjoint from both values and computations. **return** v lifts values into computations, and is also the result of fully evaluating a computation. **do** $x \leftarrow c_1$ **in** c_2 acts like a let-binding, sequencing computations. This

forces programmers to be explicit the order of evaluation. First, c_1 is fully evaluated to obtain some **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` c_1 c_2, where c_1 and c_2 are compuations that evaluate to naturals? Are both arguments evaluated before application, or are evaluation and application interleaved? The syntax forces programmers to choose explicitly. The programmer can either fully evaluate both arguments before applying them in turn:

do $x \leftarrow c_1$ **in** (**do** $y \leftarrow c_2$ **in** (**do** $f \leftarrow$ `plus` x **in** fy)) λ_{op}

or alternatively, c_1, apply it, then evaluate c_2:

do $x \leftarrow c_1$ **in** (**do** $f \leftarrow$ `plus` x **in** (**do** $y \leftarrow c_2$ **in** fy)) λ_{op}

Both choices are valid, but the programmer must choose. For clarity, where the order cannot affect the result (for example, $c_1 =$ **return** 1, $c_2 =$ **return** 2), I abuse notation and write $c_1 + c_2$. Similarly, for clarity, I implicitly cast values to computations (writing $1 + 2$ rather than **return** $1 +$ **return** 2).

Performing effects: `op`, `handle`, and `continue`

Recall that effect handlers allow users to register custom effects with custom semantics. 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 effect **print** in advance. They may thus write programs which refer to **print**:

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

In the program fragment above, **print** is an effect, but with as-yet-unknown 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. Consider defining an effect handler that accumulates print statements in a string (some "`stdout`"). For example, the aforementioned program should return $(3, \texttt{"1; 2"})$.

First, an effect handler must handle programs that have no calls to **print**. For such a program, the accummulating handler should return both the value and the empty string: $(3, \texttt{""})$. The return handler is written as follows:

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

where c is set to **return** $(x, \texttt{""})$. All effect handlers must specify a return handler. In many cases, the return handler is simply the identity.

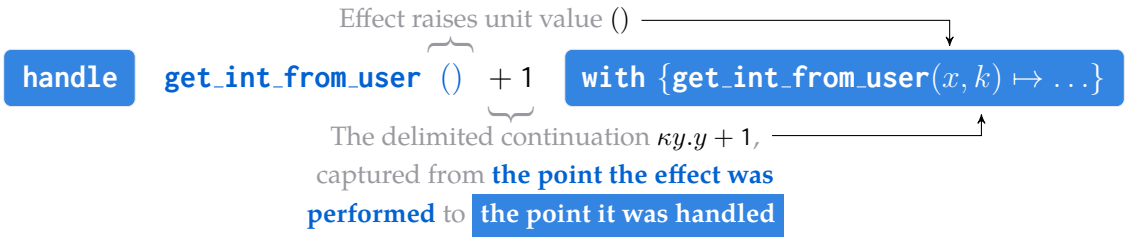
Next, an effect handler must handle programs that perform **print** effects, by specifying an operation handler of the form:

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

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

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

In the definition of c, the programmer may refer to x, the argument passed to **print**, and k, a delimited continuation from the point the effect was performed to the point it was handled. The delimited continuation can be thought of as a suspended program, awaiting a value from its environment. Effects allow programs to receive data from their environments, as in:



where the program that adds one is suspended until the value is received. I write the suspended program as $\kappa y.\ y + 1$, where the variable y indicates the as-yet-unknown value. $\kappa y.\ y + 1$ is the continuation bound to k. The syntax is evocative: continuations can be thought of as being *like* functions $\lambda y.\ y + 1$ (in Pretnar's calculus, functions and continuations share a type). However, for technical reasons relating to scope extrusion (Sections 4.4 and 4.5), in λ_{op}, continuations and functions are disambiguated both syntactically and at the type-level (mirroring the approach by Isoda et al. [12]). The expression

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

is used to resume the suspended program, substiting value v for y. For example,

$$\textbf{handle}\ \texttt{get\_int\_from\_user}() + 1\ \textbf{with}\ \{\texttt{get\_int\_from\_user}(x, k) \mapsto \textbf{continue}\ k\ 2\}$$

steps to $2 + 1$ (and thus evaluates to 3).

In λ_{op}, the continuation to be bound to k is calculated. In contrast, in Pretnar's calculus, it is explicitly written by the programmer, using the syntax

$$\textbf{op}(v; y.c)$$

where $y.c$ is the continuation to be bound to k. Pretnar acknowledges that the approach adopted by λ_{op} is an equivalent formulation.

The concrete operation handler thus resumes the suspended program, supplying a unit value, since **print** effects do not receive values. Evaluating the (now unsuspended) program eventually returns a value v and some partially accumulated `stdout` s. The handler then prepends 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 4.

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

The operational semantics of λ_{op} (Figure 2.2) 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 [8]. Evaluation contexts are represented as a stack of evaluation frames F, à

Operational Semantics

λ_{op}

Auxiliary Definitions

$$
\begin{array}{lrcl}
\text{Evaluation Frame} & F & ::= & \mathbf{do}\ x \leftarrow [-]\ \mathbf{in}\ c_2 \mid \mathbf{handle}\ [-]\ \mathbf{with}\ \{h\} \\
\text{Evaluation Context} & E & ::= & [-] \mid E[F]
\end{array}
$$

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

Operational Semantics

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

Figure 2.2.: The operational semantics of λ_{op}. The semantics is given on configurations of the form $\langle c; E\rangle$. 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}.

la Kiselyov [15]. The two key rules are Eff-Op, the mechanism for giving effects custom semantics, and Eff-Cnt, the mechanism for resuming programs.

The evaluation of Listing 4, beginning with an empty context, illustrates the operaton of Eff-Op and Eff-Cnt. Let h be the handler body

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

Several applications of Cng-Psh produces the configuration

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

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

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

Applying Cng-Psh,

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

Applying Eff-Cnt resumes the program that was suspended

$$\begin{array}{l}\langle\, \textbf{return}\ ()\ \ ;\ \ \textbf{do}(v,s) \leftarrow \\ \quad\quad \textbf{handle} \\ \quad\quad\quad \textbf{do}\, x \leftarrow ([-]; \textbf{return}\ 1)\textbf{in} \\ \quad\quad\quad \textbf{do}\, y \leftarrow (\textbf{print}(2); \textbf{return}\ 2) \\ \quad\quad\quad \textbf{in}\ x + y \\ \quad\quad \textbf{with}\, \{h\} \\ \quad \textbf{in return}\ (v, \mathtt{f"\{1\};"} \mathbin{\hat{}} s)\,\rangle\end{array}$$

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, the program reads 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
```

To be precise, the domain of a handler h, $\mathsf{dom}(h)$, is the set of operations for which it has handlers. For example, the domain of

$$\{\textbf{return}(x) \mapsto \textbf{return}\ x; \textbf{read}(\_, k) \mapsto \textbf{continue}\, k\, 1; \textbf{shirk}(x, k) \mapsto \textbf{shirk}(x)\}$$

is $\{\textbf{read}, \textbf{shirk}\}$. Given an evaluation context, E, the set of effects handled by the context $\mathsf{handled}(E)$, is the union of all of the domains of the handler frames in E. That is,

$$\mathsf{handled}(\textbf{handle}\ (\textbf{handle}\ [-] + 1\ \textbf{with}\ \{h_1\})\ \textbf{with}\ \{h_2\}) = \mathsf{dom}(h_1) \cup \mathsf{dom}(h_2)$$

The latest handler h for an effect **op** in a context E is found by decomposing E into

$$E_1[\textbf{handle}\ E_2\ \textbf{with}\ \{h\}]$$

such that $\textbf{op} \in \mathsf{dom}(h)$ (h handles **op**) and $\textbf{op} \notin \mathsf{handled}(E_2)$ (no handlers for **op** in E_2). If such an h exists, this condition identifies it uniquely.

Type-and-Effect System

Figure 2.3 collates the syntax of λ_{op} types. Types are divided into value types (for example, $\mathbb{N}$), computation types ($\mathbb{N}\,!\,\{\textbf{print}\}$), and handler types ($\mathbb{N}\,!\,\{\textbf{print}\} \Longrightarrow \mathbb{N}\,!\,\emptyset$). Since computations may have effects, computation types track unhandled effects using an effects row (Δ), which in λ_{op} 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:

Types

λ_{op}

Effects row	$\Delta ::= \emptyset \mid \Delta \cup \{\mathsf{op}_i\}$	
Value type	$S, T ::= \mathbb{N}$	
	$\mid S \xrightarrow{\Delta} T$	functions
	$\mid S \overset{\Delta}{\twoheadrightarrow} T$	continuations
Computation type	$T \,!\, \Delta$	
Handler type	$S \,!\, \Delta_1 \Longrightarrow T \,!\, \Delta_2$	

Figure 2.3.: λ_{op} types. Notice that, just as terms are divided into values, computations, and handlers, types are divided into value types (S, T), computation types $(T \,!\, \Delta)$, and handler types $(S \,!\, \Delta_1 \Longrightarrow T \,!\, \Delta_2)$

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

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

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

is a value that accepts a value of type $\mathbb{N}$ and returns a computation of type $\mathbb{N} \,!\, \{\mathbf{print}\}$. Functions thus have suspended effects, which I write $S \xrightarrow{\Delta} T$. In this case, the function has type $\mathbb{N} \xrightarrow{\{\mathbf{print}\}} \mathbb{N}$. Recall that continuations and functions, while similar, need to be distinguished for technical reasons relating to scope extrusion.

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. Second, by modifying the return type of computations. To reflect both abilities, handlers are given a type of the form $S \,!\, \Delta_1 \Longrightarrow T \,!\, \Delta_2$. For example, a handler of the form

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

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.

λ_{op} assumes that the user declares their effects in advance. It additionally assumes that they declare the types of their effects in advance, and that this mapping is stored in Σ. That is,

$$\mathsf{op} : A \to B \in \Sigma$$

In `OCaml`, this would correspond to writing:

```OCaml
type _ Effect.t += Op: A -> B
```

λ_{op} does not impose any restrictions on A and B. In particular, they may be recursive, and refer to op. For example:

$$\mathtt{recursive} : 1 \to (1 \xrightarrow{\{\mathtt{recursive}\}} 1)$$

Thus, one may write programs which do not terminate, by "tying the knot", for example

λ_{op}
```
handle (λ_.recursive())() with {recursive(_, k) ↦ continue k (λ_.recursive())}
```

Note that for the program to be well-typed, the program fragment $\lambda\_.\mathbf{recursive}()$ must refer to **recursive** in its type: $1 \xrightarrow{\{\mathtt{recursive}\}} 1$. Thus, the declared type of the **recursive** effect must be recursive.

The typing rules for terms are collated in Figure 2.4. Since types are stratified, so are typing judgements:

Value Typing Judgment	$\Gamma \vdash v : T$
Computation Typing Judgment	$\Gamma \vdash c : T \,!\, \Delta$
Handler Typing Judgement	$\Gamma \vdash h : S \,!\, \Delta_1 \Longrightarrow T \,!\, \Delta_2$

The key rules are Return, Do, Op, and the handler rules (Ret-Handler, Op-Handler).

The Return rule permits **return** v to be typed with any set of effects. For example:

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

This flexibility is important, because to type **do** $x \leftarrow c_1$ **in** c_2, the Do rule requires c_1 and c_2 to have the same effects. For example, without this flexibility, it would not be possible 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 add explicit subtyping. However, such an approach would no longer be syntax directed.

The Op rule forces the performed operation to be tracked by the effects row ($\mathsf{op} \in \Delta$): flexibility allows the type system to over-approximate the effects in a term, but not to underapproximate them.

Following the approach by Biernacki et al. [4], handlers are typed clause-by-clause, using the Ret-Handler and Op-Handler rules. In contrast, Pretnar types a handler with a single rule that checks all clauses at once. Consider typing the handler:

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

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

(Op-Handler)

$$\frac{\begin{array}{c}(1)\ \mathsf{op} : A \to B \in \Sigma \\ (2)\ \Gamma \vdash h : S \,!\, \Delta_1 \Longrightarrow T \,!\, \Delta_2 \\ (3)\ \Gamma, x : A, k : B \xrightarrow{\Delta_2} T \vdash c : T \,!\, \Delta_2 \qquad (4)\ \Delta_1 \subseteq \Delta_2 \cup \{\mathsf{op}\} \qquad (5)\ \mathbf{op}(x', k') \mapsto c' \notin h\end{array}}{\Gamma \vdash h; \mathbf{op}(x, k) \mapsto c : S \,!\, \Delta_1 \Longrightarrow T \,!\, \Delta_2}$$

Typing Rules

λ_{op}

$\boxed{\Gamma \vdash v : T}$

$$\frac{}{\Gamma \vdash n : \mathbb{N}}\ (\textsc{Nat}) \qquad \frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ (\textsc{Var}) \qquad \frac{\Gamma, x : S \vdash c : T \,!\, \Delta}{\Gamma \vdash \lambda x.c : S \xrightarrow{\Delta} T}\ (\textsc{Lambda}) \qquad \frac{\Gamma, x : S \vdash c : T \,!\, \Delta}{\Gamma \vdash \kappa x.c : S \xrightarrow{\Delta}\!\!\!\!\rightarrow T}\ (\textsc{Continuation})$$

$\boxed{\Gamma \vdash c : T \,!\, \Delta}$

$$\frac{\Gamma \vdash v_1 : S \xrightarrow{\Delta} T \qquad \Gamma \vdash v_2 : S}{\Gamma \vdash v_1\, v_2 : T \,!\, \Delta}\ (\textsc{App}) \qquad \frac{\Gamma \vdash v_1 : S \xrightarrow{\Delta}\!\!\!\!\rightarrow T \qquad \Gamma \vdash v_2 : S}{\Gamma \vdash \textbf{continue}\, v_1\, v_2 : T \,!\, \Delta}\ (\textsc{Continue})$$

$$\frac{\Gamma \vdash v : T}{\Gamma \vdash \textbf{return}\ v : T \,!\, \Delta}\ (\textsc{Return}) \qquad \frac{\Gamma \vdash c_1 : S \,!\, \Delta \qquad \Gamma, x : S \vdash c_2 : T \,!\, \Delta}{\Gamma \vdash \textbf{do}\ x \leftarrow c_1\ \textbf{in}\ c_2 : T \,!\, \Delta}\ (\textsc{Do})$$

$$\frac{\Gamma \vdash v : S \qquad \mathsf{op} : S \to T \in \Sigma \qquad \mathsf{op} \in \Delta}{\Gamma \vdash \textbf{op}(v) : T \,!\, \Delta}\ (\textsc{Op}) \qquad \frac{\Gamma \vdash c : S \,!\, \Delta_1 \qquad \Gamma \vdash h : S \,!\, \Delta_1 \Longrightarrow T \,!\, \Delta_2 \qquad \forall \mathsf{op} \in \Delta_1 \setminus \Delta_2.\, \mathsf{op} \in \mathsf{dom}(h)}{\Gamma \vdash \textbf{handle}\ c\ \textbf{with}\ \{h\} : T \,!\, \Delta_2}\ (\textsc{Handle})$$

$\boxed{\Gamma \vdash h : S \,!\, \Delta_1 \Longrightarrow T \,!\, \Delta_2}$

$$\frac{\Gamma, x : S \vdash c : T \,!\, \Delta_2}{\Gamma \vdash \textbf{return}(x) \mapsto c : S \,!\, \Delta_1 \Longrightarrow T \,!\, \Delta_2}\ (\textsc{Ret-Handler})$$

$$\frac{\begin{array}{c}\mathsf{op} : A \to B \in \Sigma \\ \Gamma \vdash h : S \,!\, \Delta_1 \Longrightarrow T \,!\, \Delta_2 \\ \Gamma, x : A, k : B \xrightarrow{\Delta_2}\!\!\!\!\rightarrow T \vdash c : T \,!\, \Delta_2 \qquad \Delta_1 \subseteq \Delta_2 \cup \{\mathsf{op}\} \qquad \textbf{op}(x', k') \mapsto c' \notin h\end{array}}{\Gamma \vdash h; \textbf{op}(x, k) \mapsto c : S \,!\, \Delta_1 \Longrightarrow T \,!\, \Delta_2}\ (\textsc{Op-Handler})$$

Figure 2.4.: Typing rules for λ_{op} terms. Typing judgements are stratified into value, computation, and handler judgments.

It is sufficient to show that:

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

(2) The rest of the handler $h = \texttt{return}(x) \mapsto \texttt{return}\ (x, \texttt{""})$ 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) The body

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

has type $(\mathbb{N} \times \mathsf{String})\,!\,\emptyset$, assuming x has type $\mathbb{N}$ and k has type $1 \xrightarrow{\emptyset\!\!\to} (\mathbb{N} \times \mathsf{String})$.

(4) The handler removes *at most* **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) There are no other handlers for **print** in h.

Finally, a closed computation is well-typed if it can be typed with an empty effects row.

Definition 2.2.1 (Well-typed closed computation)
c is well-typed if $\cdot \vdash c : T\,!\,\emptyset$

Metatheory

Discussion around the interaction between effect handlers and metaprogramming will build on some metatheoretic properties of λ_{op}, which are proven by Bauer and Pretnar [2].

Theorem 2.2.2 (Progress)
If $\cdot \vdash E[c] : T\,!\,\Delta$ then either

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

Theorem 2.2.3 (Preservation)
If $\cdot \vdash E[c] : T\,!\,\Delta$ and $\langle c; E\rangle \to \langle c'; E'\rangle$, then $\cdot \vdash E'[c'] : T\,!\,\Delta$

Corollary 2.2.4 (Type Safety)
If $\cdot \vdash c : T\,!\,\emptyset$ then either

1. *$\langle c; [-]\rangle \to^{\omega}$ (non-termination)*
2. *$\langle c; [-]\rangle \to^{*} \langle \texttt{return}\ v; [-]\rangle$*

2.2.3. The Design Space of Effect Handlers

The design space of effect handlers is large. Different design decisions could impact how effect handlers interact with metaprogramming. This thesis focuses on unnamed and deep effect handlers that permit multi-shot continuations:

1. **Named or Unnamed Handlers**
   Can an effect invoke a specific handler, rather than the latest?

   In λ_{op}, when there are multiple handlers for the same effect, the latest handler for that effect is invoked. An alternative approach is **named handlers** [37], where, by associating each handler with a *name*, the programmer can more easily specify which handler should be invoked.

   While named handlers can be more ergonomic [37], like other works on scope extrusion [12], this thesis only considers unnamed handlers.

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-Op) 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. These are known as **deep** handlers.

   λ_{op}

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

   ```
   return 5
   ```

   Listing 5: A λ_{op} program which contrasts deep, shallow, and sheep handlers

   An alternative semantics would *not* reinstate the handler, in an approach known as **shallow** handlers [9]. Listing 5 would be stuck, since the second **addn** would not be handled.

   Another alternative would be 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, the programmer could add 1 the first time **addn** is performed, and 2 the second time (Listing 5 would return $2 + 4 = 6$). These handlers behave as a hybrid of shallow and deep handlers, and are thus termed **sheep** handlers [23].

   In keeping with most prior work on scope extrusion, this thesis focuses on deep handlers [12].

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

   In λ_{op}, continuations may be resumed multiple times (Listing 6):

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

```
return 2
```

Listing 6: A λ_{op} program that resumes the same continuation multiple times

Such an effect system is said to permit **multi-shot continuations**. Multi-shot continuations are useful for simulating certain effects, like non-determinism [23].

In `OCaml`, multi-shot continuations are not allowed: continuations are only allowed to be resumed once. These systems permit only **one-shot continuations**.

Although continuations in `OCaml` are one-shot, this thesis studies effect systems with multi-shot continuations.

2.3. Scope Extrusion

Scope extrusion is an undesirable consequence of unrestricted interaction between effect handlers and metaprogramming. To illustrate scope extrusion, I first extend λ_{op} (Section 2.2.2) with quotes $\langle\langle e \rangle\rangle$ and splices $\$e$.

Scope extrusion is observed when programs are evaluated. To evaluate quotes and splices, I rely on the conceptual model introduced in Section 2.1, where quotes convert terms to ASTs, and splices under quotes suspend this conversion (Section 3.3 makes this precise). Additionally, in this section, I assume all computation implicitly takes place under a top-level splice. For example, extending λ_{op} with **`skip`**, the following program:

$$\langle\langle \lambda x : \mathbb{N}.\ \$(\textbf{skip}; \langle\langle x \rangle\rangle) + \texttt{0} \rangle\rangle$$

elaborates into

$$\textbf{return}\ \texttt{Lam}(x_{\mathbb{N}}, \texttt{Plus}(\textbf{skip}; \texttt{Var}(x_{\mathbb{N}}), \texttt{Nat}(\texttt{0})))$$

and thus evaluates to

$$\textbf{return}\ \texttt{Lam}(x_{\mathbb{N}}, \texttt{Plus}(\texttt{Var}(x_{\mathbb{N}}), \texttt{Nat}(\texttt{0})))$$

```
handle
  ⟨⟨ λx : ℕ. $ (extrude( ⟨⟨x⟩⟩ )) ⟩⟩
with
  {return(u) ↦ return ⟨⟨0⟩⟩; extrude(y, k) ↦ return y}
```
λop

```
return Var(x_ℕ)
```

Listing 7: A λ_{op} program that evaluates to $\texttt{Var}(x_{\mathbb{N}})$. The AST is ill-scoped, and thus exhibits scope extrusion. It is used as a running example.

Listing 7 illustrates the problem of scope extrusion. The program performs an effect, **`extrude`**, with type $\mathbb{N}\,\texttt{expr} \to \mathbb{N}\,\texttt{expr}$. The handler for **`extrude`** discards the continuation, returning the value it was given: $\texttt{Var}(x_{\mathbb{N}})$. The program evaluates to $\texttt{Var}(x_{\mathbb{N}})$, and the generated AST is ill-scoped. The result of evaluation demonstrates scope extrusion.

It is difficult to give a precise definition to scope extrusion, because there are multiple competing definitions [16, 19], and many are given informally. For example, is scope extrusion a property of the *result* of evaluation [19], as in Listing 7, or is it a property of *intermediate* configurations [16]? It is possible to build ASTs with extruded variables that are bound at some future point in the execution. The program in Listing 8 produces, during its evaluation, the intermediate AST $\texttt{Var}(x_{\mathbb{N}})$, which is not well scoped. However, the result of evaluation, $\texttt{Lam}(x_{\mathbb{N}}, \texttt{Var}(x_{\mathbb{N}}))$, is well scoped. Listing 8 exhibits scope extrusion under *some*, but not *all*, definitions of scope extrusion.

λ_{op}

```
handle
  ⟨⟨ λx : ℕ. $ (extrude( ⟨⟨x⟩⟩ )) ⟩⟩
with
  {return(u) ↦ return u; extrude(y, k) ↦ continue k y}
```

```
return Lam(x_ℕ, Var(x_ℕ))
```

Listing 8: A λ_{op} program where the result of the program is well-scoped, but not all intermediate results are well-scoped.

Nevertheless, both definitions agree on the example in Listing 7. Making precise the competing definitions of scope extrusion 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 (checks are inserted into code generators, and errors raised when the code generator is executed). This section surveys two dynamic approaches, which I term the lazy and eager checks, and one static approach, the method of refined environment classifiers [19, 12].

Lazy Dynamic Check

Scope extrusion of the kind in Listing 7 may seem trivial to resolve: evaluate the program to completion, and check that the resulting AST is well-scoped [16]. I term this the **lazy dynamic check**. This approach, while clearly correct and maximally expressive, is not ideal for efficiency and error reporting reasons.

λ_{op}

```
do x ← handle
         ⟨⟨ λx : ℕ. $ (extrude( ⟨⟨x⟩⟩ )) ⟩⟩
       with
         {return(u) ↦ return ⟨⟨0⟩⟩; extrude(y, k) ↦ return y}
in some very long program; return x
```

```
return Var(x_ℕ)
```

Listing 9: A λ_{op} program that evaluates to $\texttt{Var}(x_{\mathbb{N}})$. Executing the entire program to determine if it exhibits scope extrusion is inefficient.

To illustrate the inefficiency of this approach, consider a slight variation of Listing 7, Listing 9. In Listing 9, in theory, it is possible to report a warning as soon as the effect is handled. 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, the lazy dynamic check loses information about *which program fragment* was responsible for scope extrusion, reducing the informativeness of reported errors [16].

Eager Dynamic Check

Motivated by the problems with the lazy dynamic check, the **eager dynamic check**, first introduced by Kiselyov [16] and implemented in BER MetaOCaml, adopts a stricter definition of scope extrusion: unbound free variables in intermediate ASTs are defined to be scope extrusion. By this definition, Listing 8 exhibits scope extrusion. To report scope extrusion, checks are inserted at various intermediate points of code generation (in contrast to the lazy dynamic check, where only one check is inserted at the end of code generation). The eager dynamic check offers better efficiency and error reporting guarantees over the lazy dynamic check [16].

However, the eager dynamic check is not without issues. First, even by its own definition, the eager dynamic check does not catch all occurrences of scope extrusion: while the program in Listing 8 is theoretically scope extrusion, scope extrusion is not actually reported by the check.

Second, the check is unpredictable. The problem relates to *when* the checking is performed, which Section 4.3 makes precise. To illustrate the problem, consider Listing 10, a slight variation of Listing 8 in which the program fragment y in $\textbf{continue}\, k\, y$ is replaced with $\textbf{continue}\, k\, \langle\langle \$y + \texttt{0} \rangle\rangle$. In contrast to Listing 8, Listing 10 will fail the check: scope extrusion will be reported.

λ_{op}

$$\begin{array}{l}\textbf{handle} \\ \quad \langle\langle\, \lambda x : \mathbb{N}.\ \$\,(\textbf{extrude}(\,\langle\langle x \rangle\rangle\,))\ \rangle\rangle \\ \textbf{with} \\ \quad \{\textbf{return}(u) \mapsto \textbf{return}\ u;\ \textbf{extrude}(y, k) \mapsto \textbf{continue}\, k\, \langle\langle \$y + \texttt{0} \rangle\rangle\} \\ \hline \\ \texttt{return Lam}(x_{\mathbb{N}}, \texttt{Plus(Var}(x_{\mathbb{N}}), \texttt{Nat(0)}))\end{array}$$

Listing 10: A λ_{op} program that is a slight variation of Listing 8, but that (unlike Listing 8) fails the eager dynamic check.

Without knowledge of the internal operation of the check (Section 4.3), it is difficult to reason about which programs will pass, and which will fail, the eager dynamic check. I conjecture that this behaviour is unintuitive, and exposes too much of the internal operation of the check 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 that:

1. *Created* ASTs are well-scoped

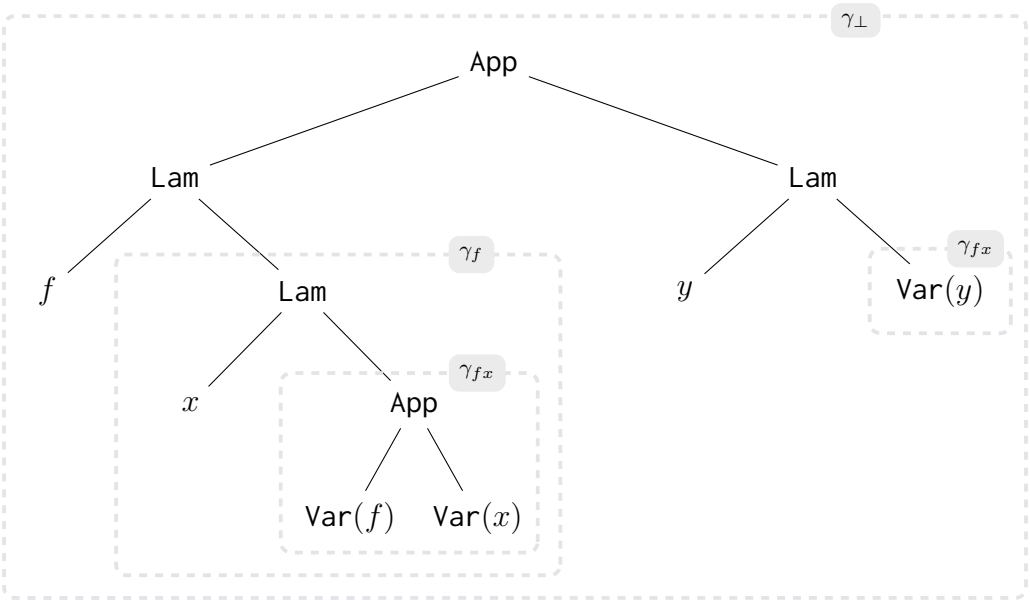


Figure 2.5.: The AST of $(\lambda f.\lambda x.fx)(\lambda y.y)$, where each scope is labelled with the corresponding environment classifier.

2. *Manipulating* ASTs preserves well-scopedness

First, ignoring the ability to manipulate ASTs, how do refined environment classifiers ensure *created* ASTs are well-scoped? What does it mean to be well-scoped? Consider Figure 2.5, the AST of the following term:

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

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 all free variables are 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:

$\gamma_\perp$ 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 the binder $\mathtt{Lam}(\alpha, -)$, $\mathsf{classifier}(\mathtt{Lam}(\alpha, -))$. For example, $\mathsf{classifier}(\mathtt{Lam}(x, -)) = \gamma_{fx}$.

Classifiers make precise "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}. γ_f is compatible with γ_{fx}, written:

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

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

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

anti-symmetry captures that 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 transitivity accounts for nestings within nestings

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

$\gamma_\bot$ acts as the least element of this partial order, and captures the notion of "top-level"

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

A classifier γ thus defines a set of variables which are permitted to be free, written $\mathsf{permitted}(\gamma)$:

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

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

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 $\mathsf{permitted}(\gamma)$, written:

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

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** (this assumption is revisited in Section 3.1.1), has roughly the following shape:

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

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 $\lambda x.c$ is well-scoped at type $T_1 \to T_2$ and scope γ.

(2) Since the function introduces a new variable binder, $\lambda x.\_$, 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) $\mathsf{classifier}(\mathsf{Lam}(x, -)) = \gamma'$ (as a shorthand $(x : T_1)^{\gamma'}$)

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

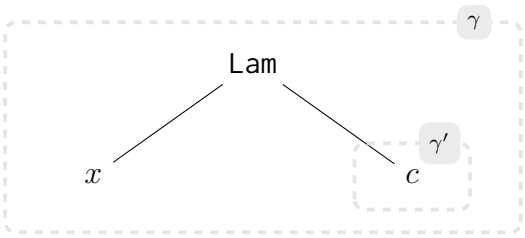


Figure 2.6.: Visual depiction of the C-Abs typing rule.

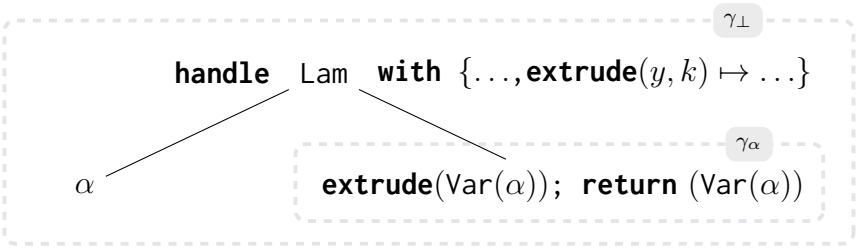


Figure 2.7.: The "AST" of the scope extrusion example, Listing 7. In place of AST nodes, we may now have compile-time executable programs that *evaluate* to AST nodes.

The above example focused on **creating** ASTs, and had no compile-time executable code. Refined environment classifiers additionally ensure well-scopedness is maintained while **manipulating** ASTs. Consider Figure 2.7, the "AST" of the scope extrusion example in Listing 7. In place of AST nodes, we may now have compile-time executable programs that *evaluate* to AST nodes. Thus, both programs and AST nodes reside within scopes. We have two classifiers: $\gamma_\perp$ and γ_α, with classifier$(\lambda\alpha._) = \gamma_\alpha$.

Key to the prevention of scope extrusion is the typing of handlers and operations, like **extrude**, that manipulate ASTs. As these rules are complex, I describe them informally. The handle expression **handle** e **with** $\{h\}$ is in scope $\gamma_\perp$. Therefore, for each operation handled by h (e.g. **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 7.

This analysis was independent of the *body* of the handler. Therefore, the examples in Listings 8 and 10 are *also* not well-typed. Perhaps somewhat surprisingly, so too is Listing 11 (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 that rejects benign programs such as Listing 11.

λ_{op}

```
handle
  ⟨⟨ λx : ℕ. $ (extrude( ⟨⟨x⟩⟩ )) ⟩⟩
  {return(u) ↦ return ⟨⟨1⟩⟩; extrude(y, k) ↦ return ⟨⟨0⟩⟩}
```

```
return Nat(0)
```

Listing 11: A λ_{op} program that passes the eager and lazy dynamic checks, but is not well-typed under the refined environment classifiers type system.

Refined environment classifiers are thus very stringent, and restrict expressiveness.

3. Calculus

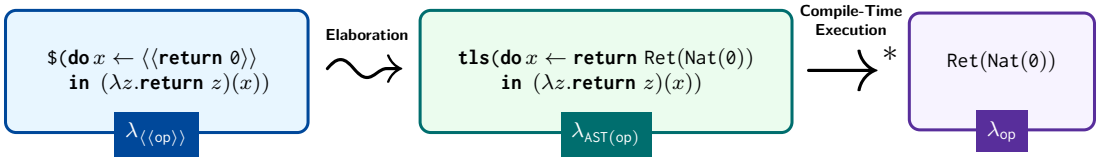


Figure 3.1.: $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ is first elaborated into $\lambda_{\mathsf{AST}(\mathsf{op})}$, which is then executed **at compile-time** to obtain the AST of a run-time λ_{op} program.

This thesis considers the interaction between homogenous, compile-time, two-stage **metaprogramming** (Page 4), and deep, unnamed **effect handlers** with multi-shot continuations (Page 9). This chapter describes a calculus, $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$, for studying said interaction. $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ has both metaprogramming and effect handlers. To the best of my knowledge, this is the first calculus in which both the generating and generated code may use effect handlers. However, $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ does not mediate the interaction between metaprogramming and effects: scope extrusion prevention is not a language feature. Rather, the aim is to extend $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ with various scope extrusion checks, and evaluate these checks 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. [38], one must first elaborate (or compile) from $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ (the "source" language) to $\lambda_{\mathsf{AST}(\mathsf{op})}$ (the "core" language). Programs written in $\lambda_{\mathsf{AST}(\mathsf{op})}$ may then be executed, to obtain the AST of a run-time λ_{op} program. This process, which mirrors the approach by Calcagno et al. [6], is summarised in Figure 3.1. Elaboration is useful, since it simplifies the operational semantics, and is a convenient mechanism for inserting dynamic checks (Chapter 4).

This chapter first introduces $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ (Section 3.1), then $\lambda_{\mathsf{AST}(\mathsf{op})}$ (Section 3.2). Following this, Section 3.3 describes the elaboration from $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ to $\lambda_{\mathsf{AST}(\mathsf{op})}$, and Section 3.4 discusses the metatheoretic properties of $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$.

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. Additionally, the continuation term former $\kappa x.e$ is removed, since it cannot be written explicitly in λ_{op}, only generated during reduction – but $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ has no reduction semantics. Recall that λ_{op} divides terms into three syntactic categories: values (v), computations (c), and handlers (h). $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ is similar, dividing terms into values (v), expressions (e), and handlers (h) (Figure 3.2).

Only expressions can be quoted (values cannot be): thus, quotes must generate run-time computations. For example, $\langle\langle 1 \rangle\rangle$ is not valid syntax, instead, one must write $\langle\langle \mathbf{return}\ 1 \rangle\rangle$. Similarly, $\langle\langle \mathbf{return}\ 1 \rangle\rangle$ is an expression, not a value, so one must write:

$$\mathbf{do}\ a \leftarrow \langle\langle \mathbf{return}\ 1 \rangle\rangle\ \mathbf{in}\ \mathbf{op}(a)$$

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

Values	v	::=	the natural lifting of λ_{op} values, with no term former for continuations
Expressions	e	::=	the natural lifting of λ_{op} expressions $\mid \langle\langle e \rangle\rangle \mid \$e$
Handlers	h	::=	the natural lifting of λ_{op} handlers

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, and the removal of the continuation term former $\kappa x.e$.

rather than $\mathbf{op}(\langle\langle \mathbf{return}\ 1 \rangle\rangle)$. However, I will abuse notation and write $\mathbf{op}(\langle\langle 1 \rangle\rangle)$ in place of $\mathbf{do}\ a \leftarrow \langle\langle \mathbf{return}\ 1 \rangle\rangle\ \mathbf{in}\ \mathbf{op}(a)$.

3.1.1. Type System

The $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ types are summarised in Figure 3.3. There are 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.

First, **types are stratified into two levels, T^0 (run-time), and T^{-1} (compile-time).**

To motivate this stratification, consider the type of the number 3 in $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$. Perhaps surprisingly, the type of 3 is not $\mathbb{N}$. Since $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ is a two-stage system, it carefully disambiguates 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:

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

As is conventional for multi-staged languages, $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ introduces integer levels to enforce separation between compile-time and run-time naturals. While the precise notion of level is slightly more involved (Definition 3.1.1), for $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$, it is sufficient to think of level 0 as run-time (so $\mathbb{N}^0$ is a run-time natural), and -1 as compile-time. Naturals can be annotated with levels, hence, the ill-typed example becomes:

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

and the well-typed example becomes (note that, like naturals, the $+$ operator can be annotated with a level):

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

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.*

Effects Row

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

Run-Time	$\xi ::= \emptyset \mid \xi \cup \{\mathsf{op}_i^0\}$
Compile-Time	$\Delta ::= \emptyset \mid \Delta \cup \{\mathsf{op}_i^{-1}\}$

Types

Level 0	Values	$S^0, T^0 ::= \mathbb{N}^0$	naturals
		$\mid (S^0 \xrightarrow{\xi} T^0)^0$	functions
		$\mid (S^0 \xrightarrow{\xi}\!\!\!\!\rightarrow T^0)^0$	continuations
	Computations	$T^0 \,!\, \xi$	
		$\mid T^0 \,!\, \Delta$	
		$\mid T^0 \,!\, \Delta; \xi$	
	Handlers	$(S^0 \,!\, \xi_1 \Longrightarrow T^0 \,!\, \xi_2)^0 \,!\, \Delta$	
Level -1	Values	$S^{-1}, T^{-1} ::= \mathbb{N}^{-1}$	naturals
		$\mid (S \xrightarrow{\Delta} T)^{-1}$	functions
		$\mid (S \xrightarrow{\Delta}\!\!\!\!\rightarrow T)^{-1}$	continuations
		$\mid \mathsf{Code}(T^0 \,!\, \xi)^{-1}$	run-time code
	Computations	$T^{-1} \,!\, \Delta$	
	Handlers	$(S^{-1} \,!\, \Delta_1 \Longrightarrow T^{-1} \,!\, \Delta_2)^{-1}$	

Figure 3.3.: $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ types. Types are stratified into two levels, 0 and -1. Similarly, effects are stratified into two levels, ξ (run-time) and Δ (compile-time). The Code type allows compile-time programs to manipulate ASTs of run-time code.

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 [1]. In a multi-stage language, separation is even more granular: for example, level 1 and level 0 run-time naturals are distinguished. However, since $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ is a two-stage language, it is sufficient to consider only levels 0 and -1. Definition 3.1.1, 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 writing a run-time program.

It is impossible, without further information, to assign a type to *program fragments*, like `3`. Without knowledge of the wider context, it is impossible to know which level the program fragment is at: in the ill-typed example, `3` occurs under a splice, but no quotes, so it occurs at level -1 and has type $\mathbb{N}^{-1}$. In contrast, in the well-typed example, `3` occurs at level 0 and has type $\mathbb{N}^{0}$. Unless otherwise stated, I always assume program fragments occur at level 0.

Second, **effect rows are stratified into ξ (run-time) and Δ (compile-time)**. The following example prints `1` at compile-time, and `2` at run-time. Further, it reads an integer at run-time.

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

Hence, $\Delta = \{\texttt{print}\}$ and $\xi = \{\texttt{print}, \texttt{readInt}\}$. Disambiguating run-time and compile-time effects stratifies types. Unlike in λ_{op}, where a term is either of value or computation type, in $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$, the stratification of types is thus more granular:

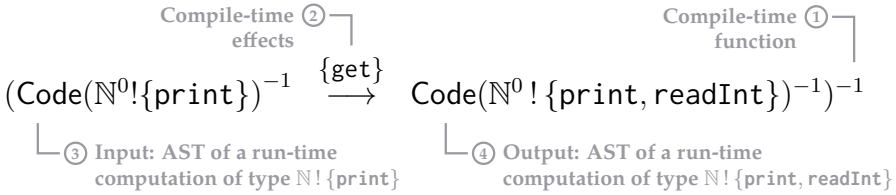
T^0	Compile-time value, run-time value (value types) *Example*: The type of x in $\lambda x.\texttt{return}\ x$
$T^0 \,!\, \xi$	Compile-time value, run-time computation *Example*: The type of x in $\$(\texttt{do}\ x \leftarrow \langle\langle \texttt{return}\ 1 \rangle\rangle\ \texttt{in return}\ x)$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

Further, the relationship between syntax and types is more complicated than in λ_{op}. For example, level 0 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ values (v) do not have value type (T^0). Rather, since level 0 values are elaborated into compile-time computations that produce ASTs of run-time values, they have computation type $T^0 \,!\, \Delta$ (Table 3.1). As the stratification is subtle, it is best revisited after covering the typing rules (Section 3.1.1), core language (Section 3.2), and elaboration (Section 3.3).

Third, **there is a level -1 Code type, representing run-time ASTs**. Stratifying types ensures that run-time (resp. compile-time) terms only interact with run-time (compile-time) terms. However, to enable metaprogramming, 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 the following:

[1] Many existing languages only consider compile-time **or** run-time metaprogramming, not both, and will thus have either non-positive or non-negative levels.



Typing Judgements

	Value (v)	**Expression** (e)	**Handler** (h)
Compile (c)	$\Gamma \vdash^{0}_{\mathsf{c}} v : T^{0} \,!\, \Delta$	$\Gamma \vdash^{0}_{\mathsf{c}} e : T^{0} \,!\, \Delta; \xi$	$\Gamma \vdash^{0}_{\mathsf{c}} h : (S^{0} \,!\, \xi_1 \Longrightarrow T^{0} \,!\, \xi_2)^{0} \,!\, \Delta$
Quote (q)	$\Gamma \vdash^{0}_{\mathsf{q}} v : T^{0} \,!\, \Delta$	$\Gamma \vdash^{0}_{\mathsf{q}} e : T^{0} \,!\, \Delta; \xi$	$\Gamma \vdash^{0}_{\mathsf{q}} h : (S^{0} \,!\, \xi_1 \Longrightarrow T^{0} \,!\, \xi_2)^{0} \,!\, \Delta$
Splice (s)	$\Gamma \vdash^{-1}_{\mathsf{s}} v : T^{-1}$	$\Gamma \vdash^{-1}_{\mathsf{s}} e : T^{-1} \,!\, \Delta$	$\Gamma \vdash^{-1}_{\mathsf{s}} h : (S^{-1} \,!\, \Delta_1 \Longrightarrow T^{-1} \,!\, \Delta_2)^{-1}$

Table 3.1.: The nine $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ typing judgements

The $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ typing rules are collated in Figures 3.6 and 3.7. As in Xie et al. [38], typing judgements are indexed by a level and a compiler mode:

$$\Gamma \vdash^{\textbf{Level}}_{\textbf{Mode}} - : =$$

For each (level, mode) pair, similarly to λ_{op}, there are three typing judgements: one for values (v), expressions (e), and handlers (h). This section shows that each compiler mode uniquely determines a level (once shown, I drop the level from the typing judgement). As there are three compiler modes – **Compile** (c), **Quote** (q), and **Splice** (s) – there are nine typing judgements in total, three for each mode (Table 3.1).

Level. Recall that it is not possible to type a program fragment, like 3, directly. One must also know the *level* (0 or -1), which is attached to the typing judgement.

Modes. During elaboration, it is 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
- **q** Code that **builds ASTs** to be manipulated at compile time.
  Last surrounding annotation is a quote

$$\underset{\mathsf{c}}{\lambda x.}\ \underset{\mathsf{s}}{\$(\textbf{do}\ f \leftarrow (\lambda y.\langle\langle \underset{\mathsf{s}}{\$(y)} + 2 \rangle\rangle)\ \textbf{in do}\ a \leftarrow \langle\langle \underset{\mathsf{q}}{1} \rangle\rangle\ \textbf{in}\ fa)} + \underset{\mathsf{c}}{3}$$

Figure 3.4.: A metaprogram annotated with compiler modes

Figure 3.4 annotates a metaprogram (that evaluates to the AST of $\lambda x.1 + 2 + 3$) with modes. The annotations clarify the purpose of each mode:

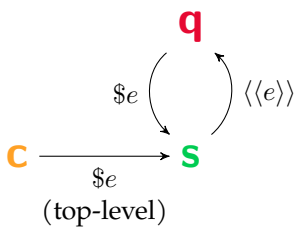


Figure 3.5.: Transitions between modes c, s, and q. Top-level splices transition from c to s, quotes transition from s to q, and splices (under quotes) transition from q to s.

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

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

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.

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

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

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

It is possible to transition between modes (Figure 3.5):

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).

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

Further, the typing judgements for c and q are 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.

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

Figure 3.6 summarises the **c** and **q**-mode typing rules. Figure 3.7 summarises the **s**-mode typing rules. In all modes, rules are extremely similar to the λ_{op} typing rules (Figure 2.4, Page 17). Further, the levels of types can, in most cases, be inferred: for readability, they are mostly omitted. The three key rules are **s**-Quote, **q**-Splice, and **c**-Splice.

Recall that the Code type makes level 0 programs available at compile-time, as ASTs. 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, at compile-time, $\langle\langle e \rangle\rangle$ is a valid AST of type Code, it is sufficient to verify that at run-time, e is a program of the corresponding type.

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

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

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

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

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.

A closed $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ expression is well-typed if, in **c**-mode, it can be typed with empty compile-time and run-time effects rows.

Definition 3.1.2 (Well-typed closed expression)
A closed expression e is well-typed if $\cdot \vdash_{\mathbf{c}} e : T^0 \,!\, \emptyset; \emptyset$

3.2. The Core Language: $\lambda_{\mathsf{AST(op)}}$

The core language, $\lambda_{\mathsf{AST(op)}}$, is a simple extension of λ_{op}. $\lambda_{\mathsf{AST(op)}}$ normal forms (n), terms (t), and handlers (h) are extended versions of λ_{op} values (v), computations (c), and handlers (h) respectively (Figure 3.8).

$\lambda_{\mathsf{AST(op)}}$ extends λ_{op} in two ways. First, it adds machinery for metaprogramming:

1. **AST** nodes (like `Nat` and `Var`) and *type-annotated* **formal parameters** (α_R, where R is some run-time value pre-type[2], henceforth simply "type").

[2] see Figure 3.10

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

c and q-mode Typing Rules

Level annotations on types mostly omitted

$\boxed{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} v : T^0 \,!\, \Delta}$

(Nat)
$$\frac{}{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} m : \mathbb{N} \,!\, \Delta}$$

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

(Lambda)
$$\frac{\Gamma, x : S \vdash_{\mathsf{c}|\mathsf{q}} e : T \,!\, \Delta; \xi}{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} \lambda x.e : (S \xrightarrow{\xi} T) \,!\, \Delta}$$

$\boxed{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} e : T^0 \,!\, \Delta; \xi}$

(App)
$$\frac{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} v_1 : (S \xrightarrow{\xi} T) \,!\, \Delta \qquad \Gamma \vdash_{\mathsf{c}|\mathsf{q}} v_2 : S \,!\, \Delta}{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} v_1 v_2 : T \,!\, \Delta; \xi}$$

(Continue)
$$\frac{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} v_1 : (S \xrightarrow{\xi\!\!\!\rightarrow} T) \,!\, \Delta \qquad \Gamma \vdash_{\mathsf{c}|\mathsf{q}} v_2 : S \,!\, \Delta}{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} \textbf{continue}\ v_1\ v_2 : T \,!\, \Delta; \xi}$$

(Return)
$$\frac{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} v : T \,!\, \Delta}{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} \textbf{return}\ v : T \,!\, \Delta; \xi}$$

(Do)
$$\frac{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} e_1 : S \,!\, \Delta; \xi \qquad \Gamma, x : S \vdash_{\mathsf{c}|\mathsf{q}} e_2 : T \,!\, \Delta; \xi}{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} \textbf{do}\ x \leftarrow e_1\ \textbf{in}\ e_2 : T \,!\, \Delta; \xi}$$

(Op)
$$\frac{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} v : S \,!\, \Delta \qquad \mathsf{op} : S \to T \in \Sigma \qquad \mathsf{op} \in \xi}{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} \textbf{op}(v) : T \,!\, \Delta; \xi}$$

(Handle)
$$\frac{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} e : S \,!\, \Delta; \xi_1 \qquad \Gamma \vdash_{\mathsf{c}|\mathsf{q}} h : S \,!\, \xi_1 \Longrightarrow T \,!\, \xi_2 \,!\, \Delta \qquad \forall \mathsf{op} \in \xi_1 \setminus \xi_2 \,.\, \mathsf{op} \in \mathsf{dom}(h)}{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} \textbf{handle}\ e\ \textbf{with}\ \{h\} : T \,!\, \Delta; \xi_2}$$

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

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

$\boxed{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} h : (S^0 \,!\, \xi_1 \Longrightarrow T^0 \,!\, \xi_2)^0 \,!\, \Delta}$

(Ret-Handler)
$$\frac{\Gamma, x : S \vdash_{\mathsf{c}|\mathsf{q}} e : T \,!\, \Delta; \xi_2}{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} \textbf{return}(x) \mapsto e : (S \,!\, \xi_1 \Longrightarrow T \,!\, \xi_2) \,!\, \Delta}$$

(Op-Handler)
$$\frac{\mathsf{op} : A \to B \in \Sigma \qquad \Gamma \vdash_{\mathsf{c}|\mathsf{q}} h : S \,!\, \xi \Longrightarrow T \,!\, \xi_2 \,!\, \Delta \qquad \Gamma, x : A, k : B \xrightarrow{\xi_2\!\!\!\rightarrow} T \vdash_{\mathsf{c}|\mathsf{q}} e : T \,!\, \Delta; \xi_2 \qquad \xi_1 \subseteq \xi_2 \cup \{\mathsf{op}\} \qquad \textbf{op}(x', k') \mapsto e' \notin h}{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} h; \textbf{op}(x, k) \mapsto e : (S \,!\, \xi_1 \Longrightarrow T \,!\, \xi_2) \,!\, \Delta}$$

Figure 3.6.: 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. Two additional rules, (c-Splice) (top-level splice) and (q-Splice) formalise the transition to s-mode.

s-mode Typing Rules

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

Level annotations on types mostly omitted

$\boxed{\Gamma \vdash_{\mathsf{s}} v : T^{-1}}$

$$\frac{}{\Gamma \vdash_{\mathsf{s}} m : \mathbb{N}}\ (\textsf{s-Nat}) \qquad \frac{\Gamma(x) = T^{-1}}{\Gamma \vdash_{\mathsf{s}} x : T^{-1}}\ (\textsf{s-Var}) \qquad \frac{\Gamma, x : S \vdash_{\mathsf{s}} e : T \,!\, \Delta}{\Gamma \vdash_{\mathsf{s}} \lambda x.e : (S \xrightarrow{\Delta} T)}\ (\textsf{s-Lambda})$$

$$\frac{\Gamma, x : S \vdash_{\mathsf{s}} e : T \,!\, \Delta}{\Gamma \vdash_{\mathsf{s}} \kappa x.e : (S \xrightarrow{\Delta}\!\!\!\!\rightarrow T)}\ (\textsf{s-Continuation})$$

$\boxed{\Gamma \vdash_{\mathsf{s}} e : T^{-1} \,!\, \Delta}$

$$\frac{\Gamma \vdash_{\mathsf{s}} v_1 : (S \xrightarrow{\Delta} T) \qquad \Gamma \vdash_{\mathsf{s}} v_2 : S}{\Gamma \vdash_{\mathsf{s}} v_1\, v_2 : T \,!\, \Delta}\ (\textsf{s-App})$$

$$\frac{\Gamma \vdash_{\mathsf{s}} v_1 : (S \xrightarrow{\Delta}\!\!\!\!\rightarrow T) \qquad \Gamma \vdash_{\mathsf{s}} v_2 : S}{\Gamma \vdash_{\mathsf{s}} \textbf{continue}\, v_1\, v_2 : T \,!\, \Delta}\ (\textsf{s-Continue})$$

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

$$\frac{\Gamma \vdash_{\mathsf{s}} e_1 : S \,!\, \Delta \qquad \Gamma, x : S \vdash_{\mathsf{s}} e_2 : T \,!\, \Delta}{\Gamma \vdash_{\mathsf{s}} \textbf{do}\ x \leftarrow e_1\ \textbf{in}\ e_2 : T \,!\, \Delta}\ (\textsf{s-Do})$$

$$\frac{\Gamma \vdash_{\mathsf{s}} v : S \qquad \mathsf{op} : S \to T \in \Sigma \qquad \mathsf{op} \in \Delta}{\Gamma \vdash_{\mathsf{s}} \textbf{op}(v) : T \,!\, \Delta}\ (\textsf{s-Op})$$

$$\frac{\Gamma \vdash_{\mathsf{s}} e : S \,!\, \Delta_1 \qquad \Gamma \vdash_{\mathsf{s}} h : (S \,!\, \Delta_1 \Longrightarrow T \,!\, \Delta_2) \qquad \forall \mathsf{op} \in \Delta_1 \setminus \Delta_2.\, \mathsf{op} \in \mathsf{dom}(h)}{\Gamma \vdash_{\mathsf{s}} \textbf{handle}\ e\ \textbf{with}\ \{h\} : T \,!\, \Delta_2}\ (\textsf{s-Handle})$$

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

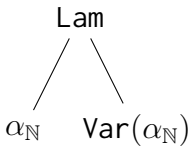
$\boxed{\Gamma \vdash_{\mathsf{s}} h : (S^{-1} \,!\, \Delta_1 \Longrightarrow T^{-1} \,!\, \Delta_2)^{-1}}$

$$\frac{\Gamma, x : S \vdash_{\mathsf{s}} e : T \,!\, \Delta_2}{\Gamma \vdash_{\mathsf{s}} \textbf{return}(x) \mapsto e : S \,!\, \Delta_1 \Longrightarrow T \,!\, \Delta_2}\ (\textsf{s-Ret-Handler})$$

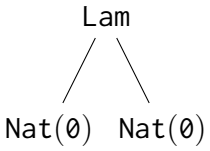
$$\frac{\mathsf{op} : A \to B \in \Sigma \qquad \Gamma \vdash_{\mathsf{s}} h : S \,!\, \Delta_1 \Longrightarrow T \,!\, \Delta_2 \qquad \Gamma, x : A, k : (B \xrightarrow{\Delta_2}\!\!\!\!\rightarrow T) \vdash_{\mathsf{s}} e : T \,!\, \Delta_2 \qquad \Delta_1 \subseteq \Delta_2 \cup \{\mathsf{op}\} \qquad \textbf{op}(x', k') \mapsto e' \notin h}{\Gamma \vdash_{\mathsf{s}} h; \textbf{op}(x, k) \mapsto e : S \,!\, \Delta_1 \Longrightarrow T \,!\, \Delta_2}\ (\textsf{s-Op-Handler})$$

Figure 3.7.: The s-mode typing rules for $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$. The rules (sans levels) are identical to the λ_{op} typing rules. The additional s-Quote rule makes level 0 code available at compile-time.

Similarly to Calcagno et al. [6], the syntax explicitly disambiguates between formal parameters (α_R) and AST nodes (though Calcagno et al. use untyped formal parameters α). For example, consider the AST of $\lambda\alpha{:}\mathbb{N}.\,\alpha$, where the formal parameter (left subtree) is contrasted with its usage (right subtree):



This syntactic distinction is important because it keeps the typing rules syntax directed, and FParams and ASTs need to have different types. If the type system did not distinguish FParams and ASTs, the following malformed AST would be well-typed:



2. **mkvar** R, a primitive for generating fresh formal parameters of type R, α_R, where two different calls to **mkvar** should return two different formal parameters.

   The need for **mkvar** arises from treating $\lambda_{\mathsf{AST(op)}}$ as an elaboration target for $\lambda_{\langle\langle\mathsf{op}\rangle\rangle}$. Consider Listing 12, which should generate the AST of $\lambda\alpha.\lambda\beta.\,\mathbf{return}\ (\alpha, \beta)$.

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

```
$(do mkfun ← λk. ⟨⟨ λx : ℕ⁰. $(k ⟨⟨x⟩⟩) ⟩⟩ in
  mkfun ( λa. mkfun (λb. return (a, b))) )
```

```
return Lam(α_ℕ, Lam(β_ℕ, Ret(Pair(Var(α_ℕ), Var(β_ℕ)))))
```

Listing 12: An example $\lambda_{\langle\langle\mathsf{op}\rangle\rangle}$ program which illustrates the need for **mkvar**

   The compile-time function `mkfun` (Walid's [35] `back`) is a higher-order function that accepts some compile-time function k. `mkfun` creates a binder ($\lambda x._-$) and passes the formal parameter x to k. k uses x to construct the body of a function $\lambda x.\mathsf{body}$. For example, `mkfun` $(\lambda y.\langle\langle\$(y) + 1\rangle\rangle)$ generates the body $x + 1$.

   In Listing 12, k calls `mkfun`. This constructs a *nested* function, whose formal parameters are bound to a and b respectively. If x was not renamed, and simply elaborated into $x_{\mathbb{N}}$, both a and b would be bound to $x_{\mathbb{N}}$, generating the following (incorrect) AST:

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

   Thus, `mkfun` should be elaborated into a function that generates *fresh* names for x each time it is called. This is the purpose of **mkvar**.

Second, $\lambda_{\mathsf{AST(op)}}$ extends λ_{op} with machinery for scope extrusion checking. This machinery comprises:

1. **err**, an error state for indicating the presence of scope extrusion,

Syntax

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

Formal Parameters	α_R		
Normal Forms	n	$::=$	the natural lifting of λ_{op} values
			$\mid \mathtt{Nat}(m) \mid \alpha_R \mid \mathtt{Var}(\alpha_R) \mid \mathtt{Lam}(n_1, n_2)$
			$\mid \mathtt{App}(n_1, n_2) \mid \mathtt{Continue}(n_1, n_2) \mid \mathtt{Ret}(n)$
			$\mid \mathtt{Do}(n_1, n_2, n_3) \mid \mathtt{Op}(n) \mid \mathtt{Hwith}(n_1, n_2)$
			$\mid \mathtt{Hret}(n_1, n_2)(n_1, n_2) \mid \mathtt{Hop}(n_1, n_2, n_3, n_4)$
Terms	t	$:=$	the natural lifting of λ_{op} computations
			$\mid \mathbf{check}\, n \mid \mathbf{check}_M\, n \mid \mathbf{dlet}(n, t) \mid \mathbf{tls}(t) \mid \mathbf{err}$
Handlers	h	$:=$	the natural lifting of λ_{op} handlers

Figure 3.8.: $\lambda_{\mathsf{AST(op)}}$ syntax. The syntax extends λ_{op} with a (countably finite) class of (type-annotated) formal parameters, AST constructors and scope extrusion checking machinery.

2. **check**, a guarded **return** 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 (as Section 4.4 explains) 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 representing an occurrence of a top-level splice in the source program.

Notice that, while the calculus provides 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

The semantics of $\lambda_{\mathsf{AST(op)}}$ are made precise via an operational semantics. Many rules are identical to those of λ_{op} (Figure 2.2): interesting rules are collated in Figure 3.9. Rules are divided into those related to AST construction (Ast-Rule), and those related to **s**cope **e**xtrusion **c**hecking (Sec-Rule).

Configurations

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

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

At a high level, t and E are terms and evaluation contexts respectively. U acts as a source of freshness for name generation. M is a set of muted variables, i.e. those that do not trigger a scope extrusion error, even if they have extruded their scope. I indicates the point at which variables in M should be *unmuted*, by setting M to $\emptyset$.

Operational Semantics

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

Selected Rules

Evaluation Contexts

$$F ::= \ldots \mid \texttt{dlet}(\alpha_R, [-]) \mid \texttt{tls}([-])$$

Operational Semantics

$$\begin{array}{lrcl}
\text{(Ast-Gen)} & \langle \texttt{mkvar}\ R; E; U; M; I\rangle & \to & \langle \texttt{return}\ \alpha_R; E; U \cup \{\alpha\}; M; I\rangle \\
 & & & (\text{where } \alpha = \mathsf{next}(U),\ \mathsf{next}(U) \notin U,\ \text{next deterministic}) \\
\text{(Sec-Chs)} & \langle \texttt{check}\, n; E; U; M; I\rangle & \to & \langle \texttt{return}\ n; E; U; M; I\rangle \\
 & & & (\text{if } \mathsf{FV}^0(n) \subseteq \pi_{\mathsf{Var}}(E)) \\
\text{(Sec-Chf)} & \langle \texttt{check}\, n; E; U; M; I\rangle & \to & \langle \texttt{err}; E; U; M; I\rangle \\
 & & & (\text{if } \mathsf{FV}^0(n) \not\subseteq \pi_{\mathsf{Var}}(E)) \\
\text{(Sec-Cms)} & \langle \texttt{check}_M\, n; E; U; M; I\rangle & \to & \langle \texttt{return}\ n; E; U; M; I\rangle \\
 & & & (\text{if } \mathsf{FV}^0(n) \setminus M \subseteq \pi_{\mathsf{Var}}(E)) \\
\text{(Sec-Cmf)} & \langle \texttt{check}_M\, n; E; U; M; I\rangle & \to & \langle \texttt{err}; E; U; M; I\rangle \\
 & & & (\text{if } \mathsf{FV}^0(n) \setminus M \not\subseteq \pi_{\mathsf{Var}}(E)) \\
\text{(Sec-Tls)} & \langle \texttt{tls}(\texttt{return}\ n); E; U; M; I\rangle & \to & \langle \texttt{return}\ n; E; U; \emptyset; \top\rangle \\
\text{(Sec-Dlt)} & \langle \texttt{dlet}(\alpha_R, \texttt{return}\ n); E; U; M; I\rangle & \to & \langle \texttt{return}\ n; E; U; M'; I'\rangle \\
 & & & (\text{if } \mathsf{len}(E) > I \text{ then } M' = M, I' = I \\
 & & & \text{else } M' = \emptyset, I' = \top) \\
\text{(Eff-Op)} & \langle \texttt{op}(v); E_1[\texttt{handle}\ E_2\ \texttt{with}\ \{h\}]; U; M; I\rangle & \to & \langle c[v/x, \mathrm{cont}/k]; E_1; U; M \cup \pi_{\mathsf{Var}}(E_2); I'\rangle \\
 & & & (\text{where } \mathrm{cont} = \kappa x.\, \texttt{handle}\ E_2[\texttt{return}\ x]\ \texttt{with}\ \{h\} \\
 & & & \text{and } \texttt{op}(x,k) \mapsto c \in h \text{ and } \texttt{op} \notin \mathsf{handled}(E_2) \\
 & & & \text{and } I' = \mathsf{min}(\mathsf{len}(E_1), I)) \\
\end{array}$$

Figure 3.9.: 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). The muting and unmuting of variables is complex, and explained in Section 4.4. For now, these mechanisms are **highlighted**.

AST Rule

The Ast-Gen rule describes the behaviour of **mkvar**: **mkvar** R produces a formal parameter of type R and some name α. Recall that names should be **fresh**: that is, multiple calls to **mkvar** should always return variables with different names. This involves remembering names that have been previously generated, which are collected in U. To ensure determinacy of the semantics, names are chosen *deterministically*.

Scope Extrusion Checking Rules

The **check** primitive acts like a guarded **return**, which can catch occurrences of scope extrusion. For some arbitrary normal form n of AST type, either:

1. All the free variables of n are properly scoped, so **check** n reduces to **return** n (Sec-Chs)

2. Some free variables of n are not properly scoped, so **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:

$$\textbf{do}\ \mathsf{body} \leftarrow \textbf{check}\,\mathtt{Plus}(\mathtt{Var}(\alpha_{\mathbb{N}}), \mathtt{Nat}(\mathtt{0}))\ \textbf{in}\ \textbf{check}\,\mathtt{Lam}(\alpha_{\mathbb{N}}, \mathsf{body})$$

$\mathtt{Var}(\alpha_{\mathbb{N}})$ should be "properly scoped": $\textbf{check}\,\mathtt{Plus}(\mathtt{Var}(\alpha_{\mathbb{N}}), \mathtt{Nat}(\mathtt{0}))$ should succeed. 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*, $\lambda_{\mathsf{AST(op)}}$ defers it to the programmer. Following Kiselyov [17], the programmer must *declare* that a variable is properly scoped through use of the **dlet** keyword.

$$\textbf{dlet}(\alpha_{\mathbb{N}}, \textbf{do}\ \mathsf{body} \leftarrow \textbf{check}\,\mathtt{Plus}(\mathtt{Var}(\alpha_{\mathbb{N}}), \mathtt{Nat}(\mathtt{0}))\ \textbf{in}\ \textbf{check}\,\mathtt{Lam}(\alpha_{\mathbb{N}}, \mathsf{body}))$$

More precisely, **dlet** places a frame of the form $\textbf{dlet}(\alpha_R, [-])$ on the evaluation context E. The notation $\pi_{\mathtt{Var}}(E)$ filters out the variables declared in this manner from E. For example,

$$\pi_{\mathtt{Var}}(\textbf{dlet}(\alpha_R, \textbf{do}\ x \leftarrow [-]\ \textbf{in}\ t)) = \{\mathtt{Var}(\alpha_R)\}$$

Given a term $E[t]$, I say variable $\mathtt{Var}(\alpha_R)$ in t is "declared safe" if $\mathtt{Var}(\alpha_R) \in \pi_{\mathtt{Var}}(E)$ (Definition 3.2.1):

Definition 3.2.1 (Declared Safe)
Given a term $E[t]$, $\mathtt{Var}(\alpha_R)$ in t is declared safe if $\mathtt{Var}(\alpha_R) \in \pi_{\mathtt{Var}}(E)$

Given a term of the form **check** n in some evaluation context E, where n is an AST, **check** thus succeeds if and only if the free Vars of n, written $\mathsf{FV}^0(n)$, have all been declared safe, $\mathsf{FV}^0(n) \subseteq \pi_{\mathtt{Var}}(E)$.

It may seem lazy 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 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$. Therefore, the onus is on the person defining the elaboration to justify that **dlet** and **check** are used appropriately.

$\textbf{check}_M$ is a variant of **check**. As Section 4.4 explains, to design an optimal scope extrusion check, it is helpful to *mute* some variables, pretending that they are properly scoped. $\textbf{check}_M$ behaves exactly like **check**, except that it pretends that the muted variables M are properly scoped ($\textbf{check}_M\ n$ succeeds if $\mathsf{FV}^0(n) \setminus M \subseteq \pi_{\mathtt{Var}}(E)$).

Similarly, when justifying the correctness of scope extrusion checks, it is useful to remember the position of top-level splices in the $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ source program. The **tls** marker indicates the occurrence of a top-level splice in the source program. Beyond unmuting variables (Section 4.4), **tls** has no operational behaviour (Sec-Tls).

The final two rules, Sec-Dlt and Eff-Op, mute or unmute Vars. Muting and unmuting are explained in Section 4.4. For now, the components of transitions corresponding to muting and unmuting are **highlighted**, and can be ignored.

Ignoring muting and unmuting, Sec-Dlt silently removes a $\textbf{dlet}(\alpha_R, [-])$ frame

$$\langle \textbf{dlet}(\alpha_R, \textbf{return}\ n); E; U; M; I\rangle \rightarrow \langle \textbf{return}\ n; E; U; M'; I'\rangle$$

and Eff-Op behaves as it does in λ_{op} (Figure 2.2):

$$\langle \textbf{op}(v); E_1[\textbf{handle}\ E_2\ \textbf{with}\ \{h\}]; U; M; I\rangle \rightarrow \langle c[v/x, \mathsf{cont}/k]; E_1; U; M \cup \pi_{\mathtt{Var}}(E_2); I'\rangle$$
$$(\mathsf{cont} = \kappa x.\, \textbf{handle}\ E_2[\textbf{return}\ x]\ \textbf{with}\ \{h\},\ \textbf{op}(x,k) \mapsto c \in h \text{ and } \textbf{op} \notin \mathsf{handled}(E_2))$$

Types

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

Computation and Handler Types omitted

Run-time Pre-types

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

Value type $\quad Q, R ::= \mathbb{N}$
$\quad\quad \mid Q \xrightarrow{\xi} R$ functions
$\quad\quad \mid Q \xrightarrow{\xi} R$ continuations

Computation type $\quad R \,!\, \xi$

Handler type $\quad Q \,!\, \xi_1 \Longrightarrow R \,!\, \xi_2$

Types

Value type $\quad S, T ::= \ldots$
$\quad\quad \mid \mathsf{FParam}(R)$ formal parameter
$\quad\quad \mid \mathsf{AST}(R)$ AST (value)
$\quad\quad \mid \mathsf{AST}(R \,!\, \xi)$ AST (computation)
$\quad\quad \mid \mathsf{AST}(Q \,!\, \xi_1 \Longrightarrow R \,!\, \xi_2)$ AST (handler)

Figure 3.10.: The types of $\lambda_{\mathsf{AST(op)}}$. $\lambda_{\mathsf{AST(op)}}$ types extend λ_{op} types with an AST type (for ASTs), and an FParam type

3.2.2. Type System

$\lambda_{\mathsf{AST(op)}}$ extends λ_{op} types with an FParam type and an AST type (Figure 3.10).

Typing Rules

The $\lambda_{\mathsf{AST(op)}}$ typing rules (Figure 3.11) are extremely straightforward: for example, α_R is an FParam of type R, and $\mathtt{Var}(\alpha_R)$ is an AST of type R. **`mkvar`** R is a computation that produces FParams of type R, and $\mathtt{Lam}(n_1, n_2)$ is well-typed if n_1 is an FParam 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:

$$\cdot \vdash \mathtt{Var}(\alpha_R) : \mathsf{AST}(R)$$

The typing rules for scope extrusion checks are even more straightforward: they are effectively invisible to the type system. The only complex case is **`err`**, which can be assigned any type in any context. **`err`** thus behaves similarly to **`absurd`** in `Haskell`, or in the λ-calculus extended with the empty type [28].

A closed $\lambda_{\mathsf{AST(op)}}$ term is well-typed if it can be typed with an empty effects row.

> **Definition 3.2.2 (Well-typed closed term)**
> *A closed term t is well-typed if $\cdot \vdash t : T \,!\, \emptyset$*

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

Selected Rules

$$\frac{}{\Gamma \vdash \alpha_R : \mathsf{FParam}(R)}\ (\textsc{Binder}) \qquad \frac{\Gamma \vdash n : \mathsf{FParam}(R)}{\Gamma \vdash \mathbf{return}\ \mathtt{Var}(n) : \mathsf{AST}(R)}\ (\textsc{Variable}) \qquad \frac{}{\Gamma \vdash \mathbf{mkvar}\ R : \mathsf{FParam}(R) \,!\, \Delta}\ (\textsc{Mkvar})$$

$$\frac{\Gamma \vdash n_1 : \mathsf{FParam}(Q) \qquad \Gamma \vdash n_2 : \mathsf{AST}(R \,!\, \xi)}{\Gamma \vdash \mathtt{Lam}(n_1, n_2) : \mathsf{AST}(Q \xrightarrow{\xi} R)}\ (\textsc{Lambda-AST})$$

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

$$\frac{\Gamma \vdash n : T \qquad T = \mathsf{AST}(R) \vee \mathsf{AST}(R \,!\, \xi) \vee \mathsf{AST}(Q \,!\, \xi_1 \Longrightarrow R \,!\, \xi_2)}{\Gamma \vdash \mathbf{check}\, n : T \,!\, \Delta}\ (\textsc{Check})$$

Figure 3.11.: Selected $\lambda_{\mathsf{AST(op)}}$ typing rules

3.2.3. Implementation

The core calculus $\lambda_{\mathsf{AST(op)}}$ corresponds to a concrete `OCaml` implementation. In the concrete `OCaml` implementation, **check**, **dlet**, and **err** are not primitives. Rather, they are encoded as a *mode of use* of effects and handlers.

1. **check** n is implemented by performing a `FreeVar` effect, that are passed the free variables of n
   check$_M$ n is similar, except there are also `Mute` and `Unmute` effects
2. **dlet**(α_R, t) is implemented as a *handler* of the `FreeVar` effect, which subtracts `Var`(α_R) from the set of free variables, and either:
   a) Resumes the continuation, if the set of free variables is now empty (all free variables declared safe)
   b) Performs another `FreeVar` effect, to check that the remaining free variables are declared safe. Following a successful such check, the continuation may be resumed.
3. **err** is an unhandled `FreeVar` effect

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

This section describes an elaboration from $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ to $\lambda_{\mathsf{AST(op)}}$. This elaboration is simple: it does not insert any dynamic scope extrusion checks (other elaborations in Chapter 4, that do insert checks, are built by extending this elaboration).

The elaboration is defined on typing judgements: $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ judgements elaborate to $\lambda_{\mathsf{AST}(\mathsf{op})}$ judgements. This decomposes into four elaborations: on effect rows, types, contexts, and terms. As the specific elaboration is clear from the context, I write $[\![-]\!]$ for all four.

3.3.1. Elaborating Effect Rows

Elaborating effect rows is just the identity, that is

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

3.3.2. Elaborating Types

To define the elaboration of types, it is convenient to refer to a helper function, erase (Appendix A.1), that, given a level 0 type, *erases* all of the level annotations (and elaborates effect rows), producing a run-time pre-type. For example:

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

In a nutshell, level 0 types elaborate into AST types, and level -1 types elaborate into themselves (sans level annotations), except for Code types, which elaborate into AST types ($[\![\mathsf{Code}(T^0 \,!\, \Delta)^{-1}]\!] = [\![T^0 \,!\, \Delta]\!]$).

$$\begin{aligned} [\![T^0]\!] &= \mathsf{AST}(\mathsf{erase}(T^0)) \\ [\![T^0 \,!\, \xi]\!] &= \mathsf{AST}(\mathsf{erase}(T^0 \,!\, \xi)) \\ [\![T^0 \,!\, \Delta]\!] &= \mathsf{AST}(\mathsf{erase}(T^0)) \,!\, [\![\Delta]\!] \\ [\![T^0 \,!\, \Delta; \xi]\!] &= \mathsf{AST}(\mathsf{erase}(T^0 \,!\, \xi)) \,!\, [\![\Delta]\!] \\ [\![(S^0 \,!\, \xi_1 \Longrightarrow T^0 \,!\, \xi_2)^0]\!] &= \mathsf{AST}(\mathsf{erase}((S^0 \,!\, \xi_1 \Longrightarrow T^0 \,!\, \xi_2)^0)) \end{aligned}$$

$$\begin{aligned} [\![\mathbb{N}^{-1}]\!] &= \mathbb{N} \\ [\![(S^{-1} \xrightarrow{\Delta} T^{-1})^{-1}]\!] &= [\![S^{-1}]\!] \xrightarrow{[\![\Delta]\!]} [\![T^{-1}]\!] \\ [\![(S^{-1} \xrightarrow{\Delta}\!\!\!\!\rightarrow T^{-1})^{-1}]\!] &= [\![S^{-1}]\!] \xrightarrow{[\![\Delta]\!]}\!\!\!\!\rightarrow [\![T^{-1}]\!] \\ [\![\mathsf{Code}(T^0 \,!\, \xi)^{-1}]\!] &= \mathsf{AST}(\mathsf{erase}(T^0 \,!\, \xi)) \\ [\![T^{-1} \,!\, \Delta]\!] &= [\![T^{-1}]\!] \,!\, [\![\Delta]\!] \\ [\![(S^{-1} \,!\, \Delta_1 \Longrightarrow T^{-1} \,!\, \Delta_2)^{-1}]\!] &= [\![S^{-1}]\!] \,!\, [\![\Delta_1]\!] \Longrightarrow [\![T^{-1}]\!] \,!\, [\![\Delta_2]\!] \end{aligned}$$

3.3.3. Elaborating Contexts

Elaboration of contexts is subtle. Level 0 types in the context are elaborated into FParam, rather than AST types. Elaboration of contexts thus requires a separate elaboration for context entries, and cannot rely naïvely on the elaboration on types.

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

To see why level 0 types are elaborated into FParam types, notice that the only cases where the context Γ is extended with a level 0 variable occur in **c** or **q**. These modes build ASTs, and thus x must be an FParam.

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

Selected Rules

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

Figure 3.12.: Selected term elaboration rules from $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ to $\lambda_{\mathsf{AST}(\mathsf{op})}$. Term elaboration is very similar to Calcagno et al. [6], adapted for compile-time generation with top-level splices, and modes rather than levels. Elaboration is moderated by the compiler mode. In **c** and **q**, elaboration builds ASTs. In **s**-mode, elaboration is effectively the identity.

3.3.4. Elaborating Terms

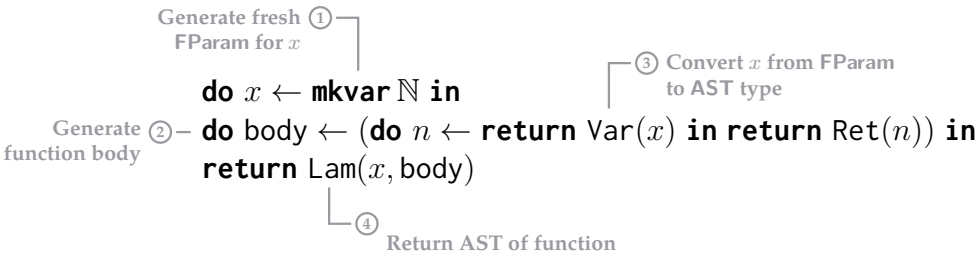
Elaboration of terms assumes that all formal parameters have been annotated with their types, for example:

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

The elaboration for terms is moderated by the **mode**: **c**, **q**, or **s**. Selected rules are collated in Figure 3.12.

At a high level, in **c** and **q**-mode, one builds ASTs. To ensure formal parameters are appropriately renamed (Listing 12), the elaboration must use **mkvar**.

Take, for example, the elaboration of $\lambda x : \mathbb{N}^0.\ \textbf{return}\ x$ in **c** or **q**, where $\mathsf{erase}(\mathbb{N}^0) = \mathbb{N}$:



Elaboration in **c** and **q**-modes do not differ significantly, with the exception of the rule for splice, where in **c**-mode, **tls** is inserted, and in **q**-mode, it is not. The **c** and **q**-modes become important when building scope extrusion checks. Elaboration in **s**-mode is effectively the identity.

3.3.5. Elaborating Typing Judgements

Elaboration of typing judgements can now be defined compositionally. For example, take the typing judgement for lambdas in **c**-mode:

$$\frac{\Gamma, x : S \vdash_{\mathsf{c}} e : T \,!\, \Delta; \xi}{\Gamma \vdash_{\mathsf{c}} \lambda x.e : (S \xrightarrow{\xi} T) \,!\, \Delta}$$

which is elaborated by applying the elaboration component-wise:

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

Letting $R = \mathsf{erase}(S)$, $Q = \mathsf{erase}(T)$, and $[\![e]\!]_{\mathsf{c}} = t$, and applying the elaboration functions defined above, we obtain Typing Derivation 3.1, which, assuming that the premise is valid typing derivation, corresponds to a valid $\lambda_{\mathsf{AST(op)}}$ typing derivation.

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

Typing Derivation 3.1.: The elaborated derivation of $\Gamma \vdash_{\mathsf{c}} \lambda x.e : S \xrightarrow{\xi} T$

Do $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ typing derivations always elaborate into $\lambda_{\mathsf{AST(op)}}$ typing derivations? Yes, but the question begets a larger point: what properties can be claimed about the calculus as defined? What metatheoretic results may be established?

3.4. Metatheory

This section states and proves several metatheoretic results about $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ and $\lambda_{\mathsf{AST(op)}}$.

First, well-typed $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ programs elaborate into well-typed $\lambda_{\mathsf{AST(op)}}$ programs:

Theorem 3.4.1 (Elaboration Preservation)
If $\Gamma \vdash_\star e : \tau$ *then* $[\![\Gamma]\!] \vdash [\![e]\!]_\star : [\![\tau]\!]$, *where* $\star = \mathsf{c} \mid \mathsf{q} \mid \mathsf{s}$ *and* τ *is a level* 0 *or level* -1 *value, computation, or handler type.*

The proof is by induction on the typing rules, e.g. Typing Derivation 3.1 in Section 3.3.5.

Additionally, the core language $\lambda_{\mathsf{AST(op)}}$ satisfies appropriate progress and preservation properties.

Theorem 3.4.2 (Progress)
If $\cdot \vdash E[t] : T \,!\, \Delta$ *then for all* U, M, I *either*

1. t *is of the form* **return** n *and* $E = [-]$,
2. t *is of the form* $\mathbf{op}(v)$ *for some* $\mathsf{op} \in \Delta$, *and* $\mathsf{op} \notin \mathit{handled}(E)$
3. t *is of the form* **err**
4. $\exists t', E', U', M', I'$ *such that* $\langle t; E; U; M; I \rangle \to \langle t'; E'; U'; M'; I' \rangle$

Note the third clause, which may be used by the calculus to report scope extrusion.

The proof of progress by induction over the typing derivation. Since $\lambda_{\mathsf{AST(op)}}$ is built by extending λ_{op}, the proof need only consider the augmented typing rules, all of which are straightforward.

Theorem 3.4.3 (Reduction Preservation)
If $\cdot \vdash E[t] : T \,!\, \Delta$ *and* $\langle t; E; U; M; I\rangle \to \langle t'; E'; U'; M'; I'\rangle$ *then* $\cdot \vdash E'[t'] : T \,!\, \Delta$

The proof of reduction preservation proceeds by induction over the operational semantics. Once again, one need only consider the augmented rules, which are simple.

As a corollary, we obtain a notion of type safety.

Corollary 3.4.4 (Type Safety)
If $\cdot \vdash_{\mathsf{c}} e : T^0 \,!\, \emptyset; \emptyset$ *then either*

1. $\langle [\![e]\!]_{\mathsf{c}}; [-]; \emptyset; \emptyset; \top\rangle \to^{\omega}$,
2. $\langle [\![e]\!]_{\mathsf{c}}; [-]; \emptyset; \emptyset; \top\rangle \to^{*} \langle \mathtt{err}; E; U; M; I\rangle$ *for some* E, U, M, I, *or*
3. $\langle [\![e]\!]_{\mathsf{c}}; [-]; \emptyset; \emptyset; \top\rangle \to^{*} \langle \mathtt{return}\ n; [-]; U; M; I\rangle$ *for some* U, M, I

where the initial configuration comprises an elaborated term, the empty evaluation context, an empty set indicating that no variables have been previously generated, another empty set indicating no variables have been muted, and $\top$*, indicating that there is (currently) no plan to unmute variables.*

Importantly, this notion of type safety is weak. A semantics which always reports a scope extrusion error (`err`) would be type safe under this definition. Further, a semantics which never reports scope extrusion would be type safe as well. Due to the potential presence of scope extrusion, the third case of Corollary 3.4.4 cannot additionally claim that the normal form n represents a well-typed λ_{op} program.

Finally, underneath a top-level splice, quotation and splice are duals (Theorem 3.4.5)

Theorem 3.4.5 (Quote-Splice Duality)
Under a top-level splice, quotation and splice are duals

$$\begin{array}{lll} \$\langle\langle e\rangle\rangle & =_{\mathsf{q}} & e \\ \langle\langle \$e\rangle\rangle & =_{\mathsf{s}} & e \end{array}$$

Where $=_{\star}$ means "elaborates to contextually equivalent $\lambda_{\mathsf{AST(op)}}$ programs in $\star$ mode". Parameterising by the mode is necessary, since the mode affects the result of elaboration. It is possible to prove something stronger: they elaborate to the same $\lambda_{\mathsf{AST(op)}}$ program (contextual equivalence follows from reflexivity). The proof of quote-splice duality is by inspection of the definition of elaboration, where:

$$\begin{array}{lcl} [\![\$\langle\langle e\rangle\rangle]\!]_{\mathsf{q}} = t & \iff & [\![e]\!]_{\mathsf{q}} = t \\ [\![\langle\langle \$e\rangle\rangle]\!]_{\mathsf{s}} = t & \iff & [\![e]\!]_{\mathsf{s}} = t \end{array}$$

4. Scope Extrusion

This chapter uses $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ to formulate precise definitions of scope extrusion, and additionally evaluates four scope extrusion checks:

1. The lazy dynamic check, first described by Kiselyov [16] (Section 4.2).
2. The eager dynamic check, which to the best of my knowledge is a faithful description of the current MetaOCaml check [19] (Section 4.3).
3. A novel best-effort dynamic check, which I argue occupies a goldilocks zone between expressiveness and efficiency (Section 4.4).
4. Refined environment classifiers, a static approach first described by Kiselyov et al. [19], and considered in the context of effect handlers by Isoda et al. [12] (Section 4.5).

I have implemented the three dynamic checks in MacoCaml. The implementations closely mirror the descriptions in this chapter, and were useful for building an understanding of the properties (Section 4.1) of the various checks.

Section 4.6 evaluates $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$'s ability to facilitate comparative evaluation of different scope extrusion checks.

4.1. Properties of Dynamic Scope Extrusion Checks

This chapter refers to the correctness and permissiveness of dynamic scope extrusion checks. Since dynamic checks are defined as term elaborations, I use $[\![-]\!]^{\mathbf{Check}}$ to indicate an arbitrary dynamic check. I refer to the term elaboration in Section 3.3 as naïve elaboration.

Given a definition of scope extrusion Φ (a predicate on configurations), a dynamic check is correct if, for all well-typed $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ expressions e, if elaborating e with the check ($[\![e]\!]^{\mathbf{Check}}$) would detect scope extrusion (transition to $\mathtt{err}$), assuming naïvely elaborating e ($[\![e]\!]$) reduces to a configuration exhibiting scope extrusion ($\Phi(\langle t; E; U; M; I\rangle)$).

> **Definition 4.1.1 (Correctness of a Dynamic Scope Extrusion Check)**
> *Given a predicate on configurations Φ, a dynamic scope extrusion check $[\![-]\!]^{\mathit{Check}}$ is correct with respect to Φ if for all closed, well-type $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ expressions e,*
>
> $$\langle [\![e]\!]; [-]; \emptyset; \emptyset; \top\rangle \to^* \langle t; E; U; M; I\rangle \wedge \Phi(\langle t; E; U; M; I\rangle)$$
> $$\Longrightarrow$$
> $$\langle [\![e]\!]^{\mathit{Check}}; [-]; \emptyset; \emptyset; \top\rangle \to^* \langle \mathtt{err}; E'; U'; M'; I'\rangle$$
>
> *for some E', U', M', I'.*

One has to consider naïve elaboration, because some checks (Section 4.4) can transform a program that exhibits scope extrusion to one that terminates early with an **err**.

The permissiveness of a scope extrusion check refers to the set of well-typed closed $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ expressions that, when elaborated, do not transition to **err** (even if they exhibit scope extrusion).

Definition 4.1.2 (Permissiveness of a Dynamic Scope Extrusion Check)
Let WellTyped be the set of closed, well-typed, $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ expressions. The permissiveness of a dynamic scope extrusion check is defined as

$$\{e \in \mathit{WellTyped} \mid \langle [\![e]\!]^{\mathit{Check}}; [-]; \emptyset; \emptyset; \top\rangle \not\to^* \langle \mathbf{err}; E; U; M; I\rangle\}$$

4.2. Lazy Dynamic Check

Scope extrusion can be defined as a predicate on $\lambda_{\mathsf{AST(op)}}$ configurations ($\langle t; E; U; M; I\rangle$). A configuration exhibits **lazy scope extrusion** when it is the *result* of compile-time execution ($E = E'[\mathbf{tls}([-])]$), and is improperly scoped ($t = \mathbf{return}\ n$, $\mathsf{FV}^0(n) \not\subseteq \pi_{\mathsf{Var}}(E)$). Lazy scope extrusion formalises the definition by Kiselyov [16].

Definition 4.2.1 (Lazy Scope Extrusion)
A $\lambda_{\mathsf{AST(op)}}$ configuration of the form

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

exhibits lazy scope extrusion if all of the following hold:

1. $t = \mathbf{return}\ n$ *for some* n *of AST type*
2. $E = E'[\mathbf{tls}([-])]$ *for some* E'
3. $\mathsf{FV}^0(n) \not\subseteq \pi_{\mathsf{Var}}(E)$

The lazy dynamic check, $[\![-]\!]^{\mathbf{Lazy}}$, augments the naïve elaboration in two ways. First, **check**s are performed after top-level splices:

$$[\![\$e]\!]^{\mathbf{Lazy}}_{\mathbf{c}} \triangleq \mathbf{check}\,(\mathbf{tls}([\![e]\!]^{\mathbf{Lazy}}_{\mathbf{s}}))$$

Second, **dlet**s are inserted to ensure variables bound outside top-level splices are declared safe (Definition 3.2.1), and thus do not cause the **check** to fail. For example, in $\lambda x : \mathbb{N}.\ \$\langle\langle \lambda y : \mathbb{N}.\ x + y\rangle\rangle$, x should be declared safe, and thus allowed to be free, but y should not. Thus, elaboration of formal parameters in **c**-mode (but not **q**-mode) should insert **dlet**s:

$$[\![\lambda x : T^0.\, e]\!]^{\mathbf{Lazy}}_{\mathbf{c}} = \mathbf{do}\ x \leftarrow \mathbf{mkvar}\,\mathsf{erase}(T^0)\ \mathbf{in}\ \mathbf{dlet}(x, \mathbf{do}\ \mathsf{body} \leftarrow [\![e]\!]^{\mathbf{Lazy}}_{\mathbf{c}}\ \mathbf{in}\ \mathbf{return}\ \mathsf{Lam}(x, \mathsf{body}))$$

For example, $\lambda x : \mathbb{N}.\ \$\langle\langle \lambda y : \mathbb{N}.\ x + y\rangle\rangle$ elaborates into (changes from the naïve

elaboration are highlighted):

$$
\begin{array}{l}
\textbf{do}\ x \leftarrow \textbf{mkvar}\ \mathbb{N}\ \textbf{in} \\
\textbf{dlet}(\ x, \textbf{do}\ \mathsf{body}_1 \leftarrow \textbf{check}(\ \textbf{tls}(\textbf{do}\ y \leftarrow \textbf{mkvar}\ \mathbb{N}\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{do}\ \mathsf{body}_2 \leftarrow (\textbf{do}\ a \leftarrow \textbf{return}\ \mathtt{Var}(x)\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{do}\ b \leftarrow \textbf{return}\ \mathtt{Var}(y)\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{return}\ \mathtt{Plus}(a, b))\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{return}\ \mathtt{Lam}(y, \mathsf{body}_2))\ )\ \textbf{in} \\
\qquad\qquad\qquad \textbf{return}\ \mathtt{Lam}(x, \mathsf{body}_1)\ )
\end{array}
$$

Due to the simplicity of the algorithm, verifying the correctness (with respect to lazy scope extrusion) and permissiveness of the check is trivial: the lazy dynamic check detects scope extrusion if, and only if, naïve elaboration would exhibit lazy scope extrusion after reduction.

Theorem 4.2.2 (Correctness and Permissiveness of the Lazy Dynamic Check)
Assuming $\cdot \vdash_c e : T^0\,!\,\emptyset;\emptyset$, *and* $[\![e]\!]_c^{\mathit{Lazy}} = t$,

$$
\begin{array}{c}
\langle t; [-]; \emptyset; \emptyset; \top\rangle \rightarrow^* \langle \textbf{err}; E; U; M; I\rangle \\
\Longleftrightarrow \\
\textit{For some } E', U', M', I', \\
\langle [\![e]\!]; [-]; \emptyset; \emptyset; \top\rangle \rightarrow^* \langle \textbf{return}\ n; E'; U'; M'; I'\rangle, \textit{ and} \\
\langle \textbf{return}\ n; E'; U'; M'; I'\rangle \textit{ exhibits lazy scope extrusion}
\end{array}
$$

The lazy scope extrusion check thus acts as a baseline, characterising the set of $\lambda_{\langle\langle \mathsf{op}\rangle\rangle}$ programs that it is "safe" to permit. This is used to define the *expressiveness* of a check, where the lazy dynamic check is *maximally* expressive:

Definition 4.2.3 (Expressiveness of a Dynamic Scope Extrusion Check)
Let the set Safe *be defined as*

$$\mathit{Safe} \triangleq \{e \in \mathit{WellTyped} \mid \langle [\![e]\!]^{\mathit{Lazy}}; [-]; \emptyset; \emptyset; \top\rangle \not\rightarrow^* \langle \textbf{err}; E; U; M; I\rangle\}$$

The permissiveness of a dynamic scope extrusion check is defined as

$$\{e \in \mathit{Safe} \mid \langle [\![e]\!]^{\mathit{Check}}; [-]; \emptyset; \emptyset; \top\rangle \not\rightarrow^* \langle \textbf{err}; E; U; M; I\rangle\}$$

Given a scope extrusion check, every rejected program that would be permitted by the lazy dynamic check is considered a false positive:

Definition 4.2.4 (False Positives of a Dynamic Scope Extrusion Check)
The false positives of a dynamic scope extrusion check are defined as

$$\{e \in \mathit{Safe} \mid \langle [\![e]\!]^{\mathit{Check}}; [-]; \emptyset; \emptyset; \top\rangle \rightarrow^* \langle \textbf{err}; E; U; M; I\rangle\}$$

However, due again to its simplicity, the lazy dynamic check is inefficient and uninformative, and therefore not suitable for practical use [16] (Section 2.3.1, Page 21).

4.3. Eager Dynamic Check

A configuration exhibits **eager scope extrusion** if it is improperly scoped (Section 2.3.1, Page 22). Notice that the definition drops the requirement that the configuration represents the result of compile-time execution, but is otherwise identical to Definition 4.2.1.

Definition 4.3.1 (Eager Scope Extrusion)
A $\lambda_{\mathsf{AST}(\mathsf{op})}$ configuration of the form

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

exhibits eager scope extrusion if all of the following hold:

1. $t = \textbf{return}\ n$ *for some* n *of AST type*
2. $FV^0(n) \not\subseteq \pi_{\mathsf{Var}}(E)$

Definition 4.3.1 formalises the definition by Kiselyov [16]:

> *At **any point during the evaluation**, an occurrence of an open-code value with a free variable whose name is not dynamically bound.*

The condition $\mathsf{FV}^0(n) \not\subseteq \pi_{\mathsf{Var}}(E)$, formally describes free variables whose "[names are] not dynamically bound".

It is possible to define an eager dynamic check by extending the lazy dynamic check. In addition to the top-level splice **check**, and the **c**-mode **dlet**s, the eager dynamic check adds **check**s for ASTs constructed in **q**-mode, for example

$$[\![v_1 v_2]\!]^{\textbf{Eager}}_{\textbf{q}} = \textbf{do}\ f \leftarrow [\![v_1]\!]^{\textbf{Eager}}_{\textbf{q}}\ \textbf{in do}\ a \leftarrow [\![v_2]\!]^{\textbf{Eager}}_{\textbf{q}}\ \textbf{in check}\ \mathsf{App}(f, a)$$

notice how **return** $\mathsf{App}(f, a)$ is replaced by **check** $\mathsf{App}(f, a)$.

Consequently, to prevent false positives, variables bound in **q**-mode must also be declared safe via **dlet**:

$$[\![\lambda x : T^0.\, e]\!]^{\textbf{Eager}}_{\textbf{q}} = \textbf{do}\ x \leftarrow \textbf{mkvar}\ \mathsf{erase}(T^0)\ \textbf{in check}\ (\textbf{dlet}(x, \textbf{do}\ \mathsf{body} \leftarrow [\![e]\!]^{\textbf{Eager}}_{\textbf{q}}\ \textbf{in return}\ \mathsf{Lam}(x, \mathsf{body})))$$

The elaboration of $\lambda x : \mathbb{N}.\ \$\langle\langle \lambda y : \mathbb{N}.\ x + y \rangle\rangle$ thus changes to (changes from the lazy dynamic check are **highlighted**):

$$\begin{aligned}
&\textbf{do}\ x \leftarrow \textbf{mkvar}\ \mathbb{N}\ \textbf{in}\\
&\textbf{dlet}(x, \textbf{do}\ \mathsf{body}_1 \leftarrow \textbf{check}(\textbf{tls}(\textbf{do}\ y \leftarrow \textbf{mkvar}\ \mathbb{N}\ \textbf{in}\ \textbf{check}(\textbf{dlet}(y,\\
&\qquad\qquad \textbf{do}\ \mathsf{body}_2 \leftarrow (\textbf{do}\ a \leftarrow \textbf{check}\ \mathsf{Var}(x)\ \textbf{in}\\
&\qquad\qquad\qquad \textbf{do}\ b \leftarrow \textbf{check}\ \mathsf{Var}(y)\ \textbf{in}\\
&\qquad\qquad\qquad \textbf{check}\ \mathsf{Plus}(a, b))\ \textbf{in}\\
&\qquad\qquad \textbf{return}\ \mathsf{Lam}(y, \mathsf{body}_2)\ ))\ ))\ \textbf{in}\\
&\qquad \textbf{return}\ \mathsf{Lam}(x, \mathsf{body}_1))
\end{aligned}$$

If **dlet**s were not inserted for binders in **q**-mode (e.g. y), then **check** $\mathsf{Plus}(a, b)$ would fail, as y would not be declared safe.

Intuitively, the eager dynamic check performs a check whenever an AST is built. Hence, assume that evaluation reduces to a configuration that exhibits eager scope extrusion. Let the offending AST be n. The error is detected and reported when, in some

evaluation context E, n is used to build a bigger AST n', and not all free variables in n' are declared safe in E.

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

$$\$(\textbf{do}\, z \leftarrow (\textbf{handle}\, \langle\langle\, \lambda x.\, \$(\, \textbf{extrude}(\langle\langle x \rangle\rangle)\,)\,\rangle\rangle \\ \quad \textbf{with}\, \{\textbf{return}(u) \mapsto \langle\langle \mathtt{0} \rangle\rangle; \textbf{extrude}(y, k) \mapsto \textbf{return}\ y\}) \\ \textbf{in}\, \langle\langle \$z + 1 \rangle\rangle)$$

Listing 13: Eager scope extrusion is caused by the **return** y expression. z is bound to $\mathtt{Var}(x_{\mathbb{N}})$. Scope extrusion is reported when z is used to build a larger AST $\langle\langle \$z + 1 \rangle\rangle$, in a context where $\mathtt{Var}(x_{\mathbb{N}})$ is not declared safe

As an example, consider Listing 13. Eager scope extrusion is caused by the **return** y program fragment, where y refers to the unbound variable $\mathtt{Var}(x_{\mathbb{N}})$[1]. Eager scope extrusion is not caught immediately. Rather, z is bound to $\mathtt{Var}(x_{\mathbb{N}})$, and scope extrusion is caught when z is used to build a larger AST, $\langle\langle \$z + 1 \rangle\rangle$, where $\mathtt{Var}(x_{\mathbb{N}})$ is not declared safe in the evaluation context.

The eager dynamic check also checks at top-level splices, like the lazy dynamic check (Listing 14). Conceptually, a top-level splice builds a larger, ambient and inert, AST (in Listing 14, $\mathtt{Do}(z_{\mathbb{N}}, [-], \mathtt{Plus}(z_{\mathbb{N}}, \mathtt{Nat}(\mathtt{1})))$):

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

$$\textbf{do}\, z \leftarrow \$(\textbf{handle}\, \langle\langle\, \lambda x.\, \$(\, \textbf{extrude}(\langle\langle x \rangle\rangle)\,)\,\rangle\rangle \\ \quad \textbf{with}\, \{\textbf{return}(u) \mapsto \langle\langle \mathtt{0} \rangle\rangle; \textbf{extrude}(y, k) \mapsto \textbf{return}\ y\}) \\ \textbf{in}\, z + 1$$

Listing 14: The eager dynamic check additionally checks at top-level splices.

To the best of my knowledge, the eager dynamic check is a faithful model of the MetaOCaml check, as described by Kiselyov [18]. The model was verified by executing translations of Listings 13, 14, 16 and 17 in BER MetaOCaml N153 (Appendix B.1).

4.3.1. Correctness of the Eager Dynamic Check

The eager dynamic check is incorrect with respect to eager scope extrusion. Evaluation may result in eager scope extrusion, but the offending AST n may never be used in an unsafe way. Thus, eager scope extrusion goes undetected.

For example, recall that $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ permits non-terminating programs. Consider the $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ program in Listing 15, where Ω is some non-terminating program that never refers to z. After translation, the program reduces to a configuration that exhibits eager scope extrusion (**return** y). However, the program immediately enters into a non-terminating loop, and scope extrusion is never reported.

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

$$\$(\textbf{do}\, z \leftarrow (\textbf{handle}\, \langle\langle\, \lambda x.\, \$(\, \textbf{extrude}(\langle\langle x \rangle\rangle)\,)\,\rangle\rangle \\ \quad \textbf{with}\, \{\textbf{return}(u) \mapsto \langle\langle \mathtt{0} \rangle\rangle; \textbf{extrude}(y, k) \mapsto \textbf{return}\ y\}) \\ \textbf{in}\, \Omega)$$

[1] For clarity, I do not rename the variables in this example

Listing 15: The eager dynamic check does not detect scope extrusion, since **return** y is followed by a non-terminating loop which never refers to z.

Non-termination is not the only case in which the eager check does not detect eager scope extrusion. For example, the offending AST could be discarded (Listing 16), and therefore never trigger the check.

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

$\$(\mathbf{handle}\ \langle\langle\, \lambda x.\, \$(\, \mathbf{extrude}(\langle\langle x \rangle\rangle)\,)\,\rangle\rangle$
$\quad \mathbf{with}\, \{\mathbf{return}(u) \mapsto \langle\langle \mathtt{0} \rangle\rangle; \mathbf{extrude}(y, k) \mapsto \mathbf{do}\ w \leftarrow \mathbf{return}\ y\ \mathbf{in}\ \langle\langle \mathtt{0} \rangle\rangle\})$

Listing 16: The eager dynamic check additionally does not report eager scope extrusion in the case where the offending AST is discarded.

Finally, the program may recover from scope extrusion by *resuming* a continuation. In Listing 17, by resuming the continuation, the program restores the captured evaluation context, thus declaring $\mathtt{Var}(x_{\mathbb{N}})$ safe. Only then is $\mathtt{Var}(x_{\mathbb{N}})$ used to build an AST, so the checks pass.

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

$\$(\mathbf{handle}\ \langle\langle\, \lambda x.\, \mathbf{return}\ \$(\, \mathbf{extrude}(\langle\langle x \rangle\rangle)\,)\,\rangle\rangle$
$\quad \mathbf{with}\, \{\mathbf{return}(u) \mapsto \mathbf{return}\ u; \mathbf{extrude}(y, k) \mapsto \mathbf{continue}\, k\, y\})$

Listing 17: The eager dynamic check additionally does not report cases where the offending AST is used, but only in safe ways. In this case, the continuation restores the context that permits $\mathtt{Var}(x_{\mathbb{N}})$ to be used.

Kiselyov [16] acknowledges that the eager dynamic check does not catch every instance of eager scope extrusion, but observes that it makes the check more permissive. Since Listings 15 to 17 are allowed by the lazy dynamic check, permissiveness makes the eager dynamic check more expressive.

4.3.2. Expressiveness of the Eager Dynamic Check

The eager dynamic check, however, is not maximally expressive. It reports false positives (e.g. Listing 18).

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

$\$(\mathbf{handle}\ \langle\langle\, \lambda x.\, \$(\, \mathbf{extrude}(\langle\langle x \rangle\rangle)\,)\,\rangle\rangle$
$\quad \mathbf{with}\, \{\mathbf{return}(u) \mapsto \mathbf{return}\ u; \mathbf{extrude}(y, k) \mapsto \mathbf{continue}\, k\, \langle\langle \$y + \mathtt{0} \rangle\rangle\})$

Listing 18: A program which fails the eager dynamic check. The offending AST ($\mathtt{Var}(x_{\mathbb{N}})$) is used to construct a larger AST in a way that appears to be unsafe $\langle\langle \$y + \mathtt{0} \rangle\rangle$, but the unsafe AST is then only used in a safe way.

In Listing 18, the offending AST ($\mathtt{Var}(x_{\mathbb{N}})$, bound to y) is used in a context where $\mathtt{Var}(x_{\mathbb{N}})$ is not declared safe ($\langle\langle \$y + \mathtt{0} \rangle\rangle$), and thus the eager dynamic check reports an error. However, if evaluation had been allowed to proceed, the evaluation context binding $\mathtt{Var}(x_{\mathbb{N}})$ (and additionally declaring it safe) would have been restored ($\mathbf{continue}\, k\, w$), and all variables would have been properly scoped. The program *recovers* from a state

of eager scope extrusion, and does not exhibit lazy scope extrusion. The eager dynamic check is thus not as expressive as the lazy dynamic check.

More concerningly, the eager dynamic check is unpredictable: it is difficult to characterise its expressiveness without referring to the operational semantics. Compare the program in Listing 17, which passes the check, and Listing 18, which *fails* the check. Unfortunately, the following inequation holds:

$$\langle\langle \$e \rangle\rangle \neq_{\mathbf{s}} \langle\langle \$e + \mathtt{0} \rangle\rangle$$

More generally, for program fragments P and P':

$$P[e] =_{\mathbf{s}} P'[e] \implies \langle\langle P[\$\langle\langle e \rangle\rangle]\rangle\rangle =_{\mathbf{s}} \langle\langle P'[\$\langle\langle e \rangle\rangle]\rangle\rangle$$

One has to appeal to the operational behaviour of the eager dynamic check to explain why these equations do not hold. In my opinion, this exposes too much of the internal operation of the check.

One possible attempt to make the eager dynamic check more predictable is to change the **s**-mode elaboration of `return`, such that each `return` elaborates into a `check`:

$$[\![\mathtt{return}\ v]\!]_{\mathbf{s}} = \mathtt{check}\,[\![v]\!]_{\mathbf{s}}$$

While this restores certain equations (for example, Listing 16 now reports a scope extrusion error), it breaks others. For example, the following equation [22] no longer holds:

$$\mathtt{do}\ \_ \leftarrow \mathtt{return}\ v\ \mathtt{in}\ e =_{\mathbf{s}} e$$

since the `return` is elaborated into a `check`, which v may fail.

4.3.3. Efficiency of the Eager Dynamic Check

Additionally, the eager dynamic check is not, in the worst case, more efficient than the lazy dynamic check, since there exist pathological examples, such as when the offending AST is not used to construct a larger AST, but returned at the top-level splice (Listing 14). However, I hypothesise that in the *common case*, the eager dynamic check detects scope extrusion sufficiently early as to outweigh the checking overhead. This question is an empirical one, and would require further research, outside the scope of the dissertation, to resolve.

4.4. Best-Effort Dynamic Check

If the lazy dynamic check is too inefficient, and the eager dynamic check too unpredictable, might it be possible to find a "goldilocks" solution? Such a check should allow the program in Listing 18, and be permissive in a predictable way. A configuration exhibits **best-effort scope extrusion** when it *must* cause lazy scope extrusion. I call this best-effort scope extrusion:

> **Definition 4.4.1 (Best-Effort Scope Extrusion)**
> *A* $\lambda_{\mathsf{AST(op)}}$ *configuration of the form*
>
> $$\langle t; E; U; M; I \rangle$$

> *exhibits best-effort scope extrusion if*
>
> $$\langle t; E; U; M; I\rangle \to^* \langle t'; E'; U'; M'; I'\rangle$$
>
> *and* $\langle t'; E'; U'; M'; I'\rangle$ *exhibits lazy scope extrusion.*

This section describes a best-effort dynamic check that approximates best-effort scope extrusion, though with false positives.

The best-effort dynamic check is simple: change all **check**s to **check**$_M$s. For example,

$$[\![\lambda x : T^0.\, e]\!]^{\mathbf{BE}}_{\mathbf{q}} = \mathbf{do}\ x \leftarrow \mathbf{mkvar}\ \mathsf{erase}(T^0)\ \mathbf{in}\ \mathbf{check}_M\ (\mathbf{dlet}(x, \mathbf{do}\ \mathsf{body} \leftarrow [\![e]\!]^{\mathbf{BE}}_{\mathbf{q}}\ \mathbf{in}\ \mathbf{return}\ \mathtt{Lam}(x, \mathsf{body})))$$

The $\lambda x : \mathbb{N}.\ \$\langle\langle \lambda y : \mathbb{N}.\ x + y\rangle\rangle$ program elaborates into (changes from the eager dynamic check are **highlighted**):

$$\begin{aligned}
&\mathbf{do}\ x \leftarrow \mathbf{mkvar}\ \mathbb{N}\ \mathbf{in}\\
&\mathbf{dlet}(x, \mathbf{do}\ \mathsf{body}_1 \leftarrow \mathbf{check}_M\ (\mathbf{tls}(\mathbf{do}\ y \leftarrow \mathbf{mkvar}\ \mathbb{N}\ \mathbf{in}\ \mathbf{check}_M\ (\mathbf{dlet}(y,\\
&\qquad\qquad \mathbf{do}\ \mathsf{body}_2 \leftarrow (\mathbf{do}\ a \leftarrow \mathbf{check}_M\ \mathtt{Var}(x)\ \mathbf{in}\\
&\qquad\qquad\qquad \mathbf{do}\ b \leftarrow \mathbf{check}_M\ \mathtt{Var}(y)\ \mathbf{in}\\
&\qquad\qquad\qquad \mathbf{check}_M\ \mathtt{Plus}(a, b))\ \mathbf{in}\\
&\qquad\qquad \mathbf{return}\ \mathtt{Lam}(y, \mathsf{body}_2))))\ \mathbf{in}\\
&\quad \mathbf{return}\ \mathtt{Lam}(x, \mathsf{body}_1))
\end{aligned}$$

To understand the best-effort dynamic check, consider Figure 4.1, where Listing 18 is elaborated using the eager check into $\lambda_{\mathsf{AST(op)}}$ and simplified for readability (e.g. **check** t rather than **do** $x \leftarrow t$ **in check** x). The failing check is underlined.

```
handle
  do x ← mkvar ℕ in
  check(dlet(x, do body ← (do a ← return Var(x) in extrude(a)) in
                  return Lam(x, body))
with
  {return(u) ↦ return u;
   extrude(y, k) ↦ do w ← check Plus(y, Nat(0)) in continue k(w)}
```

However, w is only used in a context where $\mathtt{Var}(x_{\mathbb{N}})$ is declared safe

check fails, transitioning to **err**, since y is bound to $\mathtt{Var}(x_{\mathbb{N}})$, which is not declared via **dlet**

Figure 4.1.: The result of elaborating Listing 18 using the eager check

The check fails because when **extrude** performed, the variable $\mathtt{Var}(x_{\mathbb{N}})$ is no longer declared safe in the new evaluation context. Since y is bound to $\mathtt{Var}(x_{\mathbb{N}})$, checking $\mathtt{Plus}(y, \mathtt{Nat}(0))$ reports an error. The problem is that the continuation k can be used to bind $\mathtt{Var}(x_{\mathbb{N}})$. It is not clear, when the `Plus` AST is constructed and checked, that eager scope extrusion *must* lead to lazy scope extrusion.

To make the check more expressive, therefore, it may be useful to temporarily allow $\mathtt{Var}(x_{\mathbb{N}})$ to extrude its scope, delaying error detection until one *must* have lazy scope extrusion.

The **check**$_M$ primitive allows $\mathtt{Var}(x_{\mathbb{N}})$ to temporarily extrude its scope. **check**$_M$ checks for scope extrusion, but turns a blind eye to some set of muted variables M. It thus

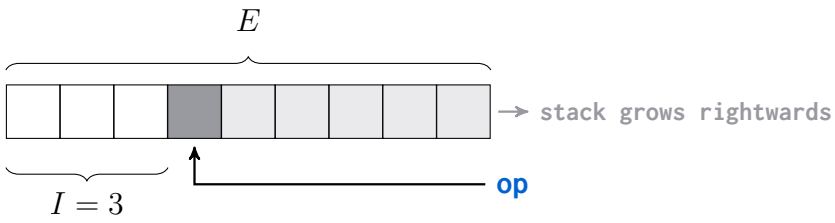


Figure 4.3.: An illustration of when variables are unmuted. The stack, E, grows rightwards. Effects (e.g. **op**, in **blue**) are caught by a handler (**dark grey**), capturing a portion of the stack (light grey). We track the length of the stack, in **white**, that is never captured by an operation in this fashion. Frames in **white** are never able to resume a continuation.

suffices to mute $\mathtt{Var}(x_{\mathbb{N}})$, by adding it to M. The $\lambda_{\mathsf{AST(op)}}$ operational semantics mutes and unmutes variables, like $\mathtt{Var}(x_{\mathbb{N}})$, at key points (Figure 3.9, Page 37).

When effects are performed, the variables which are no longer declared safe in the new evaluation context (like $\mathtt{Var}(x_{\mathbb{N}})$) are added to the set of muted variables (Eff-Op):

$$\langle \mathbf{op}(v); E_1[\mathbf{handle}\ E_2\ \mathbf{with}\ \{h\}]; U; M; I\rangle \to \langle c[v/x, \mathrm{cont}/k]; E_1; U; M \cup \pi_{\mathsf{Var}}(E_2); I'\rangle$$

As an example, consider the reduction of Figure 4.1, but where **check** has been replaced with a $\mathbf{check}_M$ (Figure 4.2).

$$\begin{array}{l}\mathbf{handle}\\ \quad \mathbf{dlet}(x_{\mathbb{N}}, \mathbf{do}\ \mathtt{body} \leftarrow \mathbf{extrude}(\mathtt{Var}(x_{\mathbb{N}}))\ \mathbf{in}\ \mathtt{Lam}(x_{\mathbb{N}}, \mathtt{body}))\\ \mathbf{with}\\ \quad \{\mathbf{return}(u) \mapsto \mathbf{return}\ u;\\ \quad\ \mathbf{extrude}(y, k) \mapsto \mathbf{do}\ w \leftarrow \mathbf{check}_M\, \mathtt{Plus}(y, \mathtt{Nat}(\mathtt{0}))\ \mathbf{in}\ \mathbf{continue}\, k\, w\}\end{array}$$

Figure 4.2.: Figure 4.1, but with $\mathbf{check}_M$ in place of **check**

The first reduction step performs the operation, and the configuration steps to the term:

$$\mathbf{do}\ w \leftarrow \mathbf{check}_M\, \mathtt{Plus}(\mathtt{Var}(x_{\mathbb{N}}), \mathtt{Nat}(\mathtt{0}))\ \mathbf{in}\ \mathbf{continue}\, k\, w$$

Since the $\mathbf{dlet}(x_{\mathbb{N}}, [-])$ frame is no longer on the stack, **check** $\mathtt{Plus}(\mathtt{Var}(x_{\mathbb{N}}), \mathtt{Nat}(\mathtt{0}))$ would throw an error. However, performing the **extrude** operation additionally *mutes* $\mathtt{Var}(x_{\mathbb{N}})$, and hence $\mathbf{check}_M\, \mathtt{Plus}(\mathtt{Var}(x_{\mathbb{N}}), \mathtt{Nat}(\mathtt{0}))$ would not throw an error.

When should a variable like $\mathtt{Var}(x_{\mathbb{N}})$ be *unmuted*? When there **cannot be** any way to resume a continuation k that could bind $\mathtt{Var}(x_{\mathbb{N}})$. A safe approximation is the point where there cannot be *any* bound continuations k. This point is identified by tracking the maximal length I of the stack E that was never captured by the handling of an effect (Figure 4.3).

As an example, consider the program in Figure 4.4, which builds the AST of $\lambda z.\, (\lambda x.\, x + \mathtt{0})(\mathtt{1})$. Note that the body of f is the simplified version of Listing 18. fig. 4.5 is coloured using the convention of Figure 4.3: light grey for the captured stack, **dark grey** for the handler, and **blue** for the effect. The surrounding context (in **black**) is identified by I:

```
check_M(dlet(z_ℕ, do b ←
  (do f ← handle
            dlet(x_ℕ, do body ← extrude(Var(x_ℕ)) in Lam(x_ℕ, body))
          with
            {return(u) ↦ return u;
             extrude(y, k) ↦ do w ← check_M Plus(y, Nat(0)) in continue k w}
   in do a ← return Nat(1)
   in check_M App(f, a))
in return Lam(Var(z_ℕ), b)))
```

Figure 4.4.: A $\lambda_{\mathsf{AST(op)}}$ program that generates the AST of $\lambda z.\,(\lambda x.\,x + 0)(1)$. It is coloured using the convention of Figure 4.3.

it is never captured by the handling of any effect, and thus must have no references to the captured continuation k.

If the stack was never captured by the handling of an effect (for example, no operations were performed), then I is set to $\top$, $\forall n \in \mathbb{N}, \top \geq n$. Performing an effect can thus *decrease* I, but never increase it. This is the side condition on Eff-Op.

$$\langle \mathbf{op}(v); E_1[\mathbf{handle}\ E_2\ \mathbf{with}\ \{h\}]; U; M; I\rangle \rightarrow \langle c[v/x, \mathsf{cont}/k]; E_1; U; M \cup \pi_{\mathsf{Var}}(E_2); I'\rangle \quad (I' = \mathsf{min}(\mathsf{len}(E_1), I))$$

During reduction, when the length of the stack is less than, or equals to, I, there must not be any remaining references to any continuations k, and thus I may be reset to $\top$, and all muted variables may be unmuted. The program in Figure 4.4 eventually reduces to the term in Figure 4.5. $[\ -\ ]$ separates the evaluation context (outside) and the term (inside). At this point, the length of the stack is less than or equals to I. It is safe to unmute all muted variables. When there are no muted variables, $\mathbf{check}_M$ and **check** have the same behaviour.

```
check_M(dlet(z_ℕ, do b ←
  (do f ← [ return Lam(x_ℕ, Plus(Var(x_ℕ), Nat(0))) ]
   in do a ← return Nat(1)
   in check_M App(f, a))
in return Lam(Var(z_ℕ), b)))
```

Figure 4.5.: The result of reducing Figure 4.4. It is now safe to unmute variables.

However, altering the semantics in such a manner means that any transition could potentially have a side effect: unmuting variables. To keep the semantics standard, and to more closely model the implementation of the check, I associate the act of unmuting with **dlet** and **tls**. A transition from **dlet** conditionally unmutes variables (Sec-Dlt rule, Figure 3.9):

$$\langle \mathbf{dlet}(\alpha_R, \mathbf{return}\ n); E; U; M; I\rangle \rightarrow \langle \mathbf{return}\ n; E; U; \emptyset; \top\rangle \quad \text{if } \mathsf{len}(E) \leq I$$
$$\langle \mathbf{dlet}(\alpha_R, \mathbf{return}\ n); E; U; M; I\rangle \rightarrow \langle \mathbf{return}\ n; E; U; M; I\rangle \quad \text{if } \mathsf{len}(E) > I$$

In Figure 4.5, the transition from $\mathbf{dlet}(z_{\mathbb{N}}, \mathbf{return}\ n)$ unmutes variables. Hence, $\mathtt{Var}(x_{\mathbb{N}})$ is still muted when the App constructor is checked, but unmuted when the outer Lam constructor is checked.

Additionally, transitions from **tls** *unconditionally* unmute variables, since the evaluation context beyond **tls** must be inert, and thus can never be captured by a handler (Sec-Tls):

$$\langle \mathbf{tls}(\mathbf{return}\ n); E; U; M; I\rangle \to \langle \mathbf{return}\ n; E; U; \emptyset; \top\rangle$$

Since $\mathbf{check}_M$s are at least as permissive as **check**s, the best-effort dynamic check is at least as expressive as the eager dynamic check.

4.4.1. Correctness of the Best-Effort Dynamic Check

The best-effort dynamic check is correct with respect to best-effort scope extrusion. The proof is simple: either one of the the non-top-level splice $\mathbf{check}_M$s reports an error, or none do. The latter case degenerates to the lazy dynamic check, where the top-level splice $\mathbf{check}_M$ must report an error.

Theorem 4.4.2 (Correctness of the Best-Effort Check)
Given a closed, well-typed $\lambda_{\langle\langle \mathsf{op}\rangle\rangle}$ expression e, if $\langle [\![e]\!]; [-]; \emptyset; \emptyset; \top\rangle$ exhibits best-effort scope extrusion then there exists E, U, M, I such that

$$\langle [\![e]\!]^{\boldsymbol{BE}}; [-]; \emptyset; \emptyset; \top\rangle \to^* \langle \mathbf{err}; E; U; M; I\rangle$$

4.4.2. Expressiveness of the Best-Effort Dynamic Check

The best-effort dynamic check is not maximally expressive. In particular, it does not allow the program in Listing 19.

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

```
$(⟨⟨λx.$(handle ⟨⟨λy.$(op(y); return y)⟩⟩
        with {return(u) ↦ return ⟨⟨0⟩⟩; op(z, k) ↦ return z})⟩⟩;
 ⟨⟨1⟩⟩)
```

Listing 19: The best-effort check reports false positives. This program attempts to build the AST $\lambda x.\mathbf{return}\ y$, where y has extruded its scope, but then throws it away, returning the AST of 1.

Listing 19 attempts to build the AST $\lambda x.\mathbf{return}\ y$, where y has extruded its scope, but then throws it away, returning the AST of 1. Critically, the constructor of the outer lambda, $\lambda x.[-]$, is never captured by any effect. Hence, Listing 19 eventually reduces to a configuration:

$$\langle \mathbf{dlet}(x_{\mathbb{N}}, \mathbf{return}\ \mathtt{Lam}(x_{\mathbb{N}}, \mathtt{Var}(y_{\mathbb{N}}))); E[\mathbf{check}[-]]; U; \{\mathtt{Var}(y_{\mathbb{N}})\}; I\rangle$$

where $\mathsf{len}(E[\mathbf{check}_M\,[-]]) < I$. The subsequent transition unmutes $\mathtt{Var}(y_{\mathbb{N}})$, and the surrounding $\mathbf{check}_M$ fails, as $\mathtt{Var}(y_{\mathbb{N}})$ is free, unmuted, and not declared safe in E.

It is possible to characterise the expressiveness of the best-effort check, via a "Cause for Concern" property. The property is defined informally as follows: assume the best-effort check reports an error, and let the offending AST be n. Now re-wind to the point

of the failing check, and consider an alternative execution where all the $\mathbf{check}_M$s are erased (turned into **return**s). In this counter-factual execution, all ASTs m that are constructed from n have at least one variable that is not declared safe in its evaluation context. Consequently, in Listing 19, the only way to safely use $\lambda x.\mathbf{return}\ y$ is to throw it away.

Theorem 4.4.3 (Cause for Concern Property)
Assuming a closed,well-typed $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ expression e, if

1. $\exists.\ E, U, M, I$ *such that* $\langle [\![e]\!]^{\mathbf{BE}}; [-]; \emptyset; \emptyset; \top \rangle \rightarrow^* \langle \mathbf{check}_M\ n; E; U; M; I \rangle$, *and* $\langle \mathbf{check}_M\ n; E; U; M; I \rangle \rightarrow \langle \mathbf{err}; E; U; M; I \rangle$

Then for all $i \in \mathbb{N}$, if

2. $\langle \mathbf{return}\ n; \mathit{erase\text{-}checks}(E); U; M; I \rangle \rightarrow^i \langle \mathbf{return}\ m; E'; U'; M'; I' \rangle$
3. *and n a subtree of m,*

then $\mathsf{FV}^0(m) \not\subseteq \pi_{\mathsf{Var}}(E')$

The proof of Theorem 4.4.3 is by contradiction. By assumption (1) there must be at least one variable, call it $\mathtt{Var}(x_R)$, that is free in n and not declared safe in E. Assume for the sake of contradiction that $\mathsf{FV}^0(m) \subseteq \pi_{\mathsf{Var}}(E')$. Then, by assumption (3), the frame $\mathbf{dlet}(x_R, [-])$ must be in E'. Note that we only have to consider terms reachable via elaboration. By definition of elaboration, the only way to push the frame $\mathbf{dlet}(x_R, [-])$ on E' is by resuming a continuation containing $\mathbf{dlet}(x_R, [-])$. But then we must have had access to the resumed continuation in E. In turn, this implies $I < \mathsf{len}(E)$, and thus $\mathtt{Var}(x_R)$ would not have been unmuted. Consequently, $\mathbf{check}_M\ n$ would not have failed because of $\mathtt{Var}(x_R)$. This contradicts assumption (1).

The expressiveness of the eager dynamic check cannot be characterised by the Cause for Concern property, with Listing 18 being a counter-example. Hence, the best-effort dynamic check is **more** expressive, and more **predictably** expressive, than the eager dynamic check.

4.4.3. Efficiency of the Best-Effort Dynamic Check

The best-effort dynamic check is no more efficient than the eager dynamic check: whenever a $\mathbf{check}_M$ reports an error, so too must a **check**, and both elaborations insert checks in the same positions. Like the eager dynamic check, I hypothesise that the best-effort check is more efficient in the common case than the lazy check.

4.5. Refined Environment Classifiers

The method of refined environment classifiers (Section 2.3.1, Page 22) presents a static approach to preventing scope extrusion. Isoda et al. [12] introduce a calculus with code combinators (rather than quotes and splices) and algebraic effects, whose interaction is moderated via refined environment classifiers. This section demonstrates that refined environment classifiers may be added to $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$, creating a new language, $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}{}^{\gamma}$.

Types (Refined Environment Classifiers) $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}{}^{\gamma}$

Level -1 Values $T^{-1} ::= \ldots \mid (\mathsf{Code}(T^0 \,!\, \xi)^{\gamma})^{-1}$

Figure 4.6.: $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}{}^{\gamma}$ types. The only change from $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ is that Code types are now annotated with an environment classifier γ, which is **highlighted**.

$\lambda_{\langle\langle \mathsf{op} \rangle\rangle}{}^{\gamma}$ shares a syntax with $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$, but augments the $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ type system with a **simplified** version of Isoda et al.'s type system:

1. $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ types are straightforwardly extended, by annotating all level -1 Code types with a classifier (Figure 4.6).
2. Similarly, level 0 types are associated with a classifier. However, following Isoda et al. [12], for level 0 types, classifiers are associated with the judgement rather than annotated onto the type (Figure 4.7).

$\lambda_{\langle\langle \mathsf{op} \rangle\rangle}{}^{\gamma}$ elaborates into $\lambda_{\mathsf{AST}(\mathsf{op})}{}^{\gamma}$, which is a simple extension of $\lambda_{\mathsf{AST}(\mathsf{op})}$. Classifiers show up only in the formal parameters, and are invisible to the types:

1. $\lambda_{\mathsf{AST}(\mathsf{op})}{}^{\gamma}$ formal parameters are annotated with classifiers (e.g. α_R^{γ})
2. Since elaboration does not require any dynamic scope extrusion checking machinery, $\lambda_{\mathsf{AST}(\mathsf{op})}{}^{\gamma}$ does not have **check** (and $\mathbf{check}_M$), **dlet**, **tls**, and **err**. Consequently, $\lambda_{\mathsf{AST}(\mathsf{op})}{}^{\gamma}$ configurations are of the form $\langle t; E; U \rangle$.

For ease of reasoning, it is helpful to define the notion of an **extended** $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}{}^{\gamma}$ type.

Definition 4.5.1 (Extended $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}{}^{\gamma}$ type)
*An **extended** source type is either:*

1. *A level -1 type, for example, $(\mathsf{Code}(\mathbb{N}^0)^{\gamma})^{-1}$ or*
2. *A level 0 type annotated with a classifier, for example, $\mathbb{N}^0(\gamma)$.*
3. *A level 0 formal parameter type, which is a level 0 value type (like $\mathbb{N}^0$) annotated with a classifier γ, and an underline, to indicate that it is elaborated into an FParam type, $\underline{(\mathbb{N}^0(\gamma))}$*

Contexts Γ must be well-formed [12], and a well-formed context must contain the least classifier $\gamma_\perp$. The definition of closed well-typed expression is adapted to reflect this:

Definition 4.5.2 (Well-typed closed expression)
e is a well-typed expression if $\gamma_\perp \vdash_{\mathsf{c}}^{\gamma_\perp} e : T^0 \,!\, \emptyset; \emptyset$

Since level -1 types can refer to classifiers γ, level -1 types should be well-formed under a context Γ (only refer to classifiers in Γ):

Definition 4.5.3 (Well-Formed Level -1 Type)
T^{-1} is well-formed under context Γ written

$$\Gamma \vdash T^{-1}$$

if for all classifiers $\gamma \in T^{-1}$, $\gamma \in \Gamma$

Most typing rules are straightforwardly adapted, with key rules listed in Figure 4.7.

The **c**|**q**-Lambda rule corresponds to C-Abs in the refined environment classifier literature. Recall that classifiers formalise the notion of scope (Section 4.5) The $\gamma' \notin \Gamma$ constraint ensures that introducing a binder $\lambda x.\_$ introduces a new scope, via a **fresh** classifier. This is necessary for correctness. The **c**|**q**-Sub and **s**-Sub subtyping rules formalise the nesting of scopes: to show a term is well-scoped in some nested scope (γ, where $\gamma' \sqsubseteq \gamma$), it suffices to show that it is well-scoped in any of its parents (γ'), see Figure 2.5, Page 60.

Following Isoda et al., handlers and continuations are restricted to Code types; However, types and typing rules are further simplified by eliminating polymorphism.

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

$$\$(\textbf{handle do}\ x \leftarrow \textbf{genlet}(\langle\langle e \rangle\rangle)\ \textbf{in}\ \langle\langle \$x + \$x \rangle\rangle$$
$$\textbf{with}\ \{\textbf{return}(u) \mapsto u; \textbf{genlet}(y, k) \mapsto \langle\langle\ (\lambda z.\, \$(\textbf{continue}\ k\ \langle\langle z \rangle\rangle))\ y\ \rangle\rangle\})$$

Listing 20: The program uses handlers to perform let-insertion, generating $(\lambda z.z+z)(e)$ rather than $e + e$. It is hard to type this in ${\lambda_{\langle\langle \mathsf{op} \rangle\rangle}}^{\gamma}$ without polymorphism.

Isoda et al.'s typing rules for handlers and continuations are polymorphic over the classifier, to allow for *let-insertion* [39]. Let-insertion aims to reduce duplication in the generated code. Rather than generating the same AST multiple times (e.g. $\mathtt{Plus}(n, n)$), let-insertion generates a let-binding, duplicating the variable rather than the AST (e.g. $\mathtt{Do}(x_{\mathbb{N}}, n, \mathtt{Plus}(\mathtt{Var}(x_{\mathbb{N}}), \mathtt{Var}(x_{\mathbb{N}})))$). This generates more compact code. Let-insertion can be easily implemented with effects. For example, in Listing 20, the **genlet** effect is used to generate $(\lambda z.z + z)e$, rather than $e + e$. Since λ_{op} is a call-by-value calculus (Figure 2.2), let-insertion ensures e is only evaluated once. Polymorphism helps to prove that Listing 20 is safe. In Listing 20, the continuation k is resumed under a binder $\lambda z.\_$ (which introduces a classifier). Thus, the continuation k should be polymorphic over classifiers. For similar reasons, Isoda et al.'s type system demands that handlers are polymorphic over classifiers. For example, a more faithful transcription of Isoda et al.'s handler type would have the form:

$$\forall\gamma.((\mathsf{Code}(S\,!\,\xi_1)^{\gamma})\,!\,\Delta_1 \Longrightarrow (\mathsf{Code}(T\,!\,\xi_2)^{\gamma})\,!\,\Delta_2)$$

Reasoning about the correctness of polymorphic typing rules, however, is complex. It is even more complex in ${\lambda_{\langle\langle \mathsf{op} \rangle\rangle}}^{\gamma}$. Like $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$, ${\lambda_{\langle\langle \mathsf{op} \rangle\rangle}}^{\gamma}$ does not have an operational semantics, but is elaborated into ${\lambda_{\mathsf{AST}(\mathsf{op})}}^{\gamma}$ terms. Thus, it is difficult to reason directly about the ${\lambda_{\langle\langle \mathsf{op} \rangle\rangle}}^{\gamma}$ type system via progress and preservation. One has to appeal to alternative techniques, like Tait-style logical relations [32], increasing the complexity of reasoning.

This evaluation focuses on extending $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$, not on refined environment classifiers. Thus, I choose to simplify the type system, and minimise the complexity of reasoning.

It is unclear whether Isoda et al.'s system allows for non-termination. Once again, to simplify reasoning, I force effect signatures to be well-founded (not recursive). Thus,

all well-typed programs must thus terminate [14].

Proof of correctness relies on a weakening lemma. As types are stratified into two levels, and into value, computation, and handler types, there are various sub-lemmas, for example:

Lemma 4.5.4 (Weakening for Level 0 Values)
If $\Gamma \vdash^{\gamma}_{c|q} e : T^0$ *then*

1. $\Gamma, (x : S^0)^{\gamma'} \vdash^{\gamma}_{c|q} e : T^0$, *for arbitrary* $\gamma' \in \Gamma$
2. $\Gamma, (x : S^{-1}) \vdash^{\gamma}_{c|q} e : T^0$, *where* $\Gamma \vdash S^{-1}$
3. $\Gamma, \gamma' \vdash^{\gamma}_{c|q} e : T^0$, *for arbitrary* $\gamma' \notin \Gamma$
4. $\Gamma, \gamma' \sqsubseteq \gamma'' \vdash^{\gamma}_{c|q} e : T^0$, *for arbitrary* $\gamma', \gamma'' \in \Gamma$

The side conditions ensure well-formed contexts are weakened into well-formed contexts

The proof of the weakening lemma is by induction on the typing derivation.

$\lambda_{\langle\langle \mathsf{op} \rangle\rangle}{}^{\gamma}$ elaborates into $\lambda_{\mathsf{AST}(\mathsf{op})}{}^{\gamma}$. Elaboration is similar to Section 3.3, with the following changes:

1. Elaboration of types erases classifiers. For example, $[\![(\mathsf{Code}(\mathbb{N}^0)^{-1})^{\gamma}]\!] = \mathsf{AST}(\mathbb{N})$.
2. Elaboration of context entries erases proof-theoretic terms e.g. γ and $\gamma \sqsubseteq \gamma'$.
3. Elaboration of terms assumes binders have been annotated with an extended source type, and does *not* erase classifiers.
   For example, $[\![\lambda x : \mathbb{N}^0(\gamma).\ e]\!]_c = \mathbf{do}\ x \leftarrow \mathbf{mkvar}\ \mathbb{N}^{\gamma}\ \mathbf{in}\ (\mathbf{do}\ \mathsf{b} \leftarrow [\![e]\!]\ \mathbf{in}\ \mathbf{return}\ \mathsf{Lam}(x, \mathsf{b}))$
4. Elaboration of top-level splice $[\![\$e]\!]_c$ does not insert **tls**.

4.5.1. Correctness of Refined Environment Classifiers

This section shows that $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}{}^{\gamma}$ can be used to formally reason about correctness, by proving that every well-typed $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}{}^{\gamma}$ term returns a well-scoped AST on termination (Theorem 4.5.5).

Theorem 4.5.5 (Correctness of Refined Environment Classifiers)
If $\gamma_\bot \vdash^{\gamma_\bot}_{c} e : T^0\ !\ \emptyset; \emptyset$, *and* $[\![e]\!]_c = t$,
then for some U, $\langle t; [-]; \emptyset\rangle \rightarrow^* \langle \mathbf{return}\ n; [-]; U\rangle$, *and* $FV^0(n) = \emptyset$

The proof of Theorem 4.5.5 is via a Tait-style logical relation. Logical relations help to show that typing guarantees are maintained by elaboration [3].

Figure 4.8 defines the logical relation, Scoped. Scoped is defined on core language ($\lambda_{\mathsf{AST}(\mathsf{op})}$) terms, and is indexed by:

1. A context of proof-theoretic terms Θ. Given a context Γ, one can project out only the proof theoretic terms $\pi_{\gamma}(\Gamma)$. For example, given

$$\Gamma = \gamma_\bot, \gamma_1, \gamma_\bot \sqsubseteq \gamma_1, (x : \mathbb{N}^0)^{\gamma_1}, \gamma_2, \gamma_1 \sqsubseteq \gamma_2, y : (\mathsf{Code}(\mathbb{N}^0\ !\ \emptyset)^{\gamma_2})^{-1}$$

Refined Environment Classifiers Typing Rules

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

Selected Rules

$$\frac{(x : T^0)^\gamma \in \Gamma}{\Gamma \vdash^{\gamma}_{\mathsf{c|q}} x : T^0 \,!\, \Delta}\ (\mathsf{c|q}\text{-Var}) \qquad \frac{\Gamma, \gamma', \gamma \sqsubseteq \gamma', (x : S)^{\gamma'} \vdash^{\gamma'}_{\mathsf{c|q}} e : T \,!\, \Delta; \xi \qquad \gamma \in \Gamma \qquad \gamma' \notin \Gamma}{\Gamma \vdash^{\gamma}_{\mathsf{c|q}} \lambda x.e : (S \xrightarrow{\xi} T) \,!\, \Delta}\ (\mathsf{c|q}\text{-Lambda})$$

$$\frac{\Gamma \vdash_{\mathsf{s}} v : S \qquad \mathsf{op} : S \to \mathsf{Code}(T \,!\, \xi)^\gamma \in \Sigma \qquad \mathsf{op} \in \Delta}{\Gamma \vdash_{\mathsf{s}} \mathbf{op}(v) : \mathsf{Code}(T \,!\, \xi)^\gamma \,!\, \Delta}\ (\mathsf{s}\text{-Op})$$

$$\frac{\Gamma \vdash_{\mathsf{s}} v_1 : \mathsf{Code}(S \,!\, \xi_1)^\gamma \xrightarrow{\Delta\!\!\!\!\!\rightarrow} \mathsf{Code}(T \,!\, \xi_2)^{\gamma'} \qquad \Gamma \vdash_{\mathsf{s}} v_2 : \mathsf{Code}(S \,!\, \xi_1)^\gamma}{\Gamma \vdash_{\mathsf{s}} \mathbf{continue}\ v_1\ v_2 : \mathsf{Code}(T \,!\, \xi_2)^{\gamma'} \,!\, \Delta}\ (\mathsf{s}\text{-Continue})$$

$$\frac{\Gamma \vdash_{\mathsf{s}} h : (\mathsf{Code}(S \,!\, \xi_1)^\gamma) \,!\, \Delta_1 \Longrightarrow (\mathsf{Code}(T \,!\, \xi_2)^\gamma) \,!\, \Delta_2 \qquad \Gamma \vdash_{\mathsf{s}} e : \mathsf{Code}(S \,!\, \xi)^\gamma \,!\, \Delta \qquad \forall \mathsf{op} \in \Delta_1 \setminus \Delta_2.\, \mathsf{op} \in \mathsf{dom}(h)}{\Gamma \vdash_{\mathsf{s}} \mathbf{handle}\ e\ \mathbf{with}\ \{h\} : \mathsf{Code}(T \,!\, \xi_2)^\gamma \,!\, \Delta_2}\ (\mathsf{s}\text{-Handle})$$

$$\frac{\Gamma \vdash_{\mathsf{s}} e : \mathsf{Code}(T \,!\, \xi)^\gamma \,!\, \Delta}{\Gamma \vdash^{\gamma}_{\mathsf{c|q}} \$e : T \,!\, \Delta; \xi}\ (\mathsf{c|q}\text{-Splice}) \qquad \frac{\Gamma \vdash^{\gamma}_{\mathsf{q}} e : T \,!\, \Delta; \xi}{\Gamma \vdash_{\mathsf{s}} \langle\langle e \rangle\rangle : \mathsf{Code}(T \,!\, \xi)^\gamma \,!\, \Delta}\ (\mathsf{s}\text{-Quote})$$

$$\frac{\Gamma \vdash^{\gamma}_{\mathsf{c|q}} \$e : T \,!\, \Delta; \xi \qquad \Gamma \vDash \gamma \sqsubseteq \gamma'}{\Gamma \vdash^{\gamma'}_{\mathsf{c|q}} e : T \,!\, \Delta; \xi}\ (\mathsf{c|q}\text{-Sub}) \qquad \frac{\Gamma \vdash_{\mathsf{s}} e : \mathsf{Code}(T \,!\, \xi)^\gamma \,!\, \Delta \qquad \Gamma \vDash \gamma \sqsubseteq \gamma'}{\Gamma \vdash_{\mathsf{s}} e : \mathsf{Code}(T \,!\, \xi)^{\gamma'} \,!\, \Delta}\ (\mathsf{s}\text{-Sub})$$

Figure 4.7.: Selected typing rules for refined environment classifiers. The c|q-Lambda rule corresponds to C-Abs in the refined environment classifier literature. Following Isoda et al., handlers and continuations are restricted to Code types; However, unlike Isoda et al.'s system, typing rules are not polymorphic over classifiers.

The $\mathsf{Scoped}_{\Theta,T}$ Logical Relation

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

Normal Forms

$n \in \mathsf{Scoped}_{\Theta,\mathbb{N}^{-1}} \triangleq n \in \mathbb{N}$

$n \in \mathsf{Scoped}_{\Theta,T^0(\gamma)} \triangleq \cdot \vdash [\![n]\!] \in [\![T^0(\gamma)]\!]$ and $\Theta \vdash \mathsf{FV}^0(n) \subseteq \mathsf{permitted}(\gamma)$
(while defined for $T^0(\gamma)$, this definition applies for all types that elaborate to some $\mathsf{AST}(\text{value} \mid \text{computation} \mid \text{handler})$ *type, e.g. $(\mathit{Code}(T^0)^\gamma)^{-1}$, $T^0 \,!\, \xi(\gamma)$, etc)*

$n \in \mathsf{Scoped}_{\Theta,\underline{T^0(\gamma)}} \triangleq \mathtt{Var}(n) \in \mathsf{Scoped}_{\Theta,T^0(\gamma)}$

$n \in \mathsf{Scoped}_{\Theta,(S^{-1} \underline{\to} T^{-1})^{-1}} \triangleq \forall n' \in \mathsf{Scoped}_{\Theta,S^{-1}}, n\, n' \in \mathsf{Scoped}_{\Theta,T^{-1}\,!\,\Delta}$

$n \in \mathsf{Scoped}_{\Theta,(S^{-1} \underline{\underline{\to}} T^{-1})^{-1}} \triangleq \forall n' \in \mathsf{Scoped}_{\Theta,S^{-1}}, \mathbf{continue}\, n\, n' \in \mathsf{Scoped}_{\Theta,T^{-1}\,!\,\Delta}$

Handlers

$h \in \mathsf{Scoped}_{\Theta,(S^{-1}\,!\,\Delta_1 \Longrightarrow T^{-1}\,!\,\Delta_2)^{-1}} \triangleq$ if $h = \mathbf{return}(x) \mapsto t_{\text{ret}}$
$\forall n' \in \mathsf{Scoped}_{\Theta,S^{-1}}, t_{\text{ret}}[n'/x] \in \mathsf{Scoped}_{\Theta,T^{-1}\,!\,\Delta_2}$
else $h = h'; \mathbf{op}(x,k) \mapsto t_{\text{op}}, \mathsf{op} : A^{-1} \to B^{-1}$
$h' \in \mathsf{Scoped}_{\Theta,(S^{-1}\,!\,\Delta_1 \Longrightarrow T^{-1}\,!\,\Delta_2)^{-1}}$ and
$\forall n \in \mathsf{Scoped}_{\Theta,A^{-1}}, n' \in \mathsf{Scoped}_{\Theta,B^{-1} \underline{\to} T^{-1}}$,
$t_{\text{op}}[n/x, n'/k] \in \mathsf{Scoped}_{\Theta,T^{-1}\,!\,\Delta_2}$

Terms

In the following, let $\tau \,!\, \Delta$ be shorthand for any of $T^0 \,!\, \Delta(\gamma)$, $T^0 \,!\, \Delta; \xi(\gamma)$, $(S^0 \,!\, \xi_1 \Longrightarrow T^0 \,!\, \xi_2)^0 \,!\, \Delta(\gamma)$, or $T^{-1} \,!\, \Delta$

$\mathsf{Scoped}_{\Theta,\tau\,!\,\Delta} \triangleq$ The smallest property on terms t such that

1. For arbitrary U consistent with t, exists U' such that $\langle t; [-]; U\rangle \to^* \langle \mathbf{return}\ n; [-]; U'\rangle$, such that U' consistent with n, and $n \in \mathsf{Scoped}_{\Theta,\tau}$
2. For arbitrary U consistent with t, exists U' such that $\langle t; [-]; U\rangle \to^* \langle \mathbf{op}(n); E; U'\rangle \not\to$, U' consistent with $E[\mathbf{op}(n)]$, and
   a) $\mathsf{op} : A^{-1} \to B^{-1}$,
   b) $n \in \mathsf{Scoped}_{\Theta,A^{-1}}$, and
   c) for all $n' \in \mathsf{Scoped}_{\Theta,B^{-1}}$, $E[n'] \in \mathsf{Scoped}_{\Theta,\tau\,!\,\Delta}$

Where, in this context, consistent with t means that for all $\mathtt{Var}(\alpha^\gamma_R)$ or $\alpha^\gamma_R \in t$, $\alpha \in U$. This side condition ensures that we use **mkvar** correctly.

Figure 4.8.: The definition of the Scoped logical relation. Most definitions are standard. The logical relation on terms is defined as a least fixed point, following the definitions by Plotkin and Xie [24] and Kuchta [20].

the proof theoretic part of the context is

$$\pi_\gamma(\Gamma) = \gamma_\bot, \gamma_1, \gamma_\bot \sqsubseteq \gamma_1, \gamma_2, \gamma_1 \sqsubseteq \gamma_2$$

which is an instance of Θ.

2. An **extended** $\lambda_{\langle\langle \mathsf{op} \rangle\rangle^\gamma}$ type (Definition 4.5.1).

The two important definitions are the relation on the $T^0(\gamma)$ value type ($\mathsf{Scoped}_{\Theta,T^0(\gamma)}$), and the relation on terms ($\mathsf{Scoped}_{\Theta,\tau\,!\,\Delta}$).

For a normal form n to be in $\mathsf{Scoped}_{\Theta,T^0(\gamma)}$, n must be of type $\mathsf{AST}(\mathsf{erase}(T^0))$ **and** the free variables of n need to be permitted by γ (permissibility was defined in Section 2.3.1, Page 22). Recall that the definition of permissibility assumes some known partial order on classifiers, e.g. $\gamma' \sqsubseteq \gamma$. The partial order is carried by the index Θ.

$\mathsf{Scoped}_{\Theta,\tau\,!\,\Delta}$ is defined as a least fixed point, following the definitions by Plotkin and Xie [24] and Kuchta [20]. Defining the logical relation as a fixed point gives rise to the principle of Scoped-Induction:

Definition 4.5.6 (Scoped-Induction)
For some property Φ on closed terms of type $[\![\tau\,!\,\Delta]\!]$, if

1. $\langle t; [-]; U\rangle \to^* \langle \mathtt{return}\ n; [-]; U'\rangle$ *implies* $\Phi(t)$
2. $\langle t; [-]; U\rangle \to^* \langle \mathsf{op}(n); E; U'\rangle \not\to$, *with* $\mathsf{op} : A^{-1} \to B^{-1}$, $n \in \mathsf{Scoped}_{\Theta,A^{-1}}$, *and for arbitrary* $n' \in \mathsf{Scoped}_{\Theta,B^{-1}}$, $\Phi(E[n'])$ *implies* $\Phi(t)$

Then for all $t \in \mathsf{Scoped}_{\Theta,\tau\,!\,\Delta}$, $\Phi(t)$

The proof additionally relies on a closure lemma [20], and a notion of closed substitution $\rho \vDash \Gamma$. Care must be taken with substitution of level-0 variables, since these should be in the logical relation for FParams rather than ASTs (note the second clause in Definition 4.5.8).

Lemma 4.5.7 (Closure under Anti-Reduction)
If $\langle t; E; U\rangle \to^* \langle t'; E'; U'\rangle$ *and* $E'[t'] \in \mathsf{Scoped}_{\Theta,\tau\,!\,\Delta}$ *then* $E[t] \in \mathsf{Scoped}_{\Theta,\tau\,!\,\Delta}$

Definition 4.5.8 (Closed substitution)
Given a context Γ, and assuming $\Theta = \pi_\gamma(\Gamma)$, the set of closed substitutions $\rho \vDash \Gamma$ are defined inductively as follows:

1. $() \vDash \gamma_\bot$
2. *If* $\rho \vDash \Gamma$, *then for arbitrary* $\gamma \in \Gamma$, $n \in \mathsf{Scoped}_{\Theta,\underline{T^0(\gamma)}}$, $(\rho, n/x) \vDash \Gamma, (x : T^0)^\gamma$
3. *If* $\rho \vDash \Gamma$, $\Gamma \vdash T^{-1}$, *and* $n \in \mathsf{Scoped}_{\Theta,T^{-1}}$, *then* $(\rho, n/x) \vDash \Gamma, (x : T^{-1})$
4. *If* $\rho \vDash \Gamma$ *then* $\rho \vDash \Gamma, \gamma$, *for arbitrary* $\gamma \notin \Gamma$
5. *If* $\rho \vDash \Gamma$ *then* $\rho \vDash \Gamma, \gamma \sqsubseteq \gamma'$, *for arbitrary* $\gamma, \gamma' \in \Gamma$

Stratification of types and mode-indexing decomposes the fundamental lemma into many sub-lemmas, e.g. Lemma 4.5.9:

Lemma 4.5.9 (Fundamental Lemma [c, $T^0 \,!\, \Delta; \xi$] of the Scoped Logical Relation) *If* $\Gamma \vdash^{\gamma}_{c} e : T^0 \,!\, \Delta; \xi$ *then for* $\Theta = \pi_\gamma(\Gamma)$, *and for all* ρ *such that* $\rho \vDash \Gamma$,

$$[\![e]\!]_c(\rho) \in \mathit{Scoped}_{\Theta, T^0 \,!\, \Delta; \xi(\gamma)}$$

Proof of Lemma 4.5.9 is by induction on the $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}{}^{\gamma}$ typing rules. I focus on the c-Lambda (C-Abs) case, where (handwaving the side-condition on U for clarity) it suffices to show that for some arbitrary ρ, $\rho \vDash \Gamma$,

$$\textbf{do}\ x \leftarrow \textbf{mkvar}\ \mathsf{erase}(S^0(\gamma'))\ \textbf{in do}\ \mathsf{body} \leftarrow [\![e]\!]_c(\rho)\ \textbf{in return}\ \mathsf{Lam}(x, \mathsf{body})$$

in $\mathsf{Scoped}_{\Theta, (S^0 \multimap T^0)^0 \,!\, \Delta(\gamma)}$. It is clear that this reduces to

$$\textbf{do}\ \mathsf{body} \leftarrow [\![e]\!]_c(\rho, \alpha_S^{\gamma'}/x)\ \textbf{in return}\ \mathsf{Lam}(\alpha_S^{\gamma'}, \mathsf{body})$$

By anti-reduction (Lemma 4.5.7) it suffices to show that this term is in the logical relation. By weakening, and the induction hypothesis (IH), $[\![e]\!]_c(\rho, \alpha_S^{\gamma'}/x) \in \mathsf{Scoped}_{\Theta', T^0 \,!\, \Delta; \xi(\gamma')}$, where $\Theta' =, \Theta, \gamma', \gamma \sqsubseteq \gamma'$. Applying Scoped-Induction on $[\![e]\!]_c(\rho, \alpha_S^{\gamma'})$:

1. $[\![e]\!]_c(\rho, \alpha_S^{\gamma'}/x) \in \mathsf{Scoped}_{\Theta', T^0 \,!\, \Delta; \xi(\gamma')}$ reduces to some $\textbf{return}\ n$
   $\textbf{do}\ \mathsf{body} \leftarrow \textbf{return}\ n\ \textbf{in}\ \mathsf{Lam}(\alpha_S^{\gamma'}, \mathsf{body})$ reduces to $\mathsf{Lam}(\alpha_S^{\gamma'}, n)$, where $\mathsf{Var}(\alpha_S^{\gamma'})$ is bound. By IH, all the free variables in n are permitted by γ'. By the typing rules, only α is annotated with classifier γ'. Hence, under Θ, the free variables of $\mathsf{Lam}(\alpha_S^{\gamma'}, n)$ are permitted by γ. The conclusion thus follows from anti-reduction.

2. $[\![e]\!]_c(\rho, \alpha_S^{\gamma'}/x) \in \mathsf{Scoped}_{\Theta', T^0 \,!\, \Delta; \xi(\gamma')}$ reduces to $E[\textbf{op}(n)]$
   As $\textbf{do}\ \mathsf{body} \leftarrow [-]\ \textbf{in return}\ \mathsf{Lam}(\alpha_S^{\gamma'}, \mathsf{body})$ introduces no handlers, the conclusion follows immediately from IH.

4.5.2. Expressiveness of Refined Environment Classifiers

$\lambda_{\langle\langle \mathsf{op} \rangle\rangle}{}^{\gamma}$ prevents scope extrusion by looking only at the argument to the effect, not the handler. In a well-typed $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}{}^{\gamma}$ program, the only variables that may be passed to an effect **op** are those that are in scope when the handler for **op** is defined, for example, the variable z in Listing 21.

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

```
λz.$(handle ⟨⟨ λx. return $( extrude(⟨⟨z⟩⟩) ) ⟩⟩
     with {return(u) ↦ return u; extrude(y, k) ↦ continue k ()})
```

Listing 21: Refined environment classifiers allow variables to be passed to an effects, so long as the variable can never cause a scope extrusion error (e.g. z).

Consequently, Listing 22 is never well-typed. By extension, neither are Listings 15 to 18. Refined environment classifiers are less expressive than the eager dynamic check.

	Listings						
	15	16	17	18	19	20	21
Lazy	Y	Y	Y	Y	Y	Y	Y
Eager	Y	Y	Y	N	N	Y	Y
Best-Effort	Y	Y	Y	Y	N	Y	Y
Ref. Env. Classifiers	N	N	N	N	N	?	Y

Table 4.1.: Summarising expressiveness w.r.t litmus tests. Isoda et al.'s system should be able to express Listing 20, but $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}{}^{\gamma}$ cannot, and thus it is marked with a ?.

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

$\$(\textbf{handle}\ \langle\langle\, \lambda x.\, \textbf{return}\ \$(\,\textbf{extrude}(\langle\langle x \rangle\rangle)\,)\,\rangle\rangle$
$\quad \textbf{with}\,\{\textbf{return}(u) \mapsto \textbf{return}\ u; \textbf{extrude}(y, k) \mapsto \underline{\text{any arbitrary program}}\})$

Listing 22: The typing rules for refined environment classifiers forbid any program which attempts to extrude some potentially unsafe variable x to a handler.

4.6. Evaluation of $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$

The results in this chapter suggest that $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ is an appropriate language in which to encode, and evaluate, scope extrusion checks. Formalising scope extrusion in $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ aided development of a novel best-effort check, which finds a sweet spot between the eager and lazy checks. Unifying checks under $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ facilitated comparative evaluation with reference to a set of $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ programs (Table 4.1).

The cost of encoding static and dynamic checks into the same language is that reasoning about the correctness of static checks becomes more complicated (Section 4.5). It is difficult to reduce the complexity of reasoning, since dynamic checks are defined via elaboration. I hypothesise that the added complexity of reasoning is a reasonable cost to pay for a comprehensive and comparative evaluation.

5. Conclusion

This thesis makes progress on four fronts.

1. It introduces a novel two-stage calculus, $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$, that allows for effect handlers at both stages: compile-time and run-time. $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ was designed to facilitate comparison of scope extrusion checks.
2. It formally describes and evaluates a range of scope extrusion solutions in $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$, including the existing MetaOCaml check [16] and refined environment classifiers [19]. This comparison was facilitated by $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$, validating its design.
3. It formally describes and evaluates a novel best-effort dynamic check, which finds a sweet spot between expressiveness and efficiency.
4. It has provided a basis for implementations of all three dynamic checks (lazy, eager, and best-effort) in MacoCaml.

5.1. Limitations and Future Work

Let-Insertion This thesis was primarily concerned with the *undesirable* interaction of metaprogramming and effects, rather than their *desirable* interaction: the previously described let-insertion [39].

I hypothesise that $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ is a good target in which to study let-insertion. Let-insertion has typically been studied in calculi where only pure programs can be generated [12]. Let-insertion is more interesting when one can generate effectful programs. Since the order of operation becomes more important, it can be more challenging to describe an "optimal" insertion point.

Empirical Studies I hypothesised that the dynamic checks differ in efficency in the common case. I would like to verify this with empirical studies.

Formalisation The proofs in this thesis are pen-and-paper proofs. However, several are quite intricate (for example, the fundamental lemma of the Scoped logical relation). To provide greater confidence in present and future results, these proofs ought to be mechanised in a proof assistant.

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] N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, page 97–108, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583327. doi: 10.1145/1596550.1596567. URL https://doi.org/10.1145/1596550.1596567.

[4] D. Biernacki, M. Piróg, P. Polesiuk, and F. Sieczkowski. Handle with care: relational interpretation of algebraic effects and handlers. *Proc. ACM Program. Lang.*, 2(POPL), Dec. 2017. doi: 10.1145/3158096. URL https://doi.org/10.1145/3158096.

[5] 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.

[6] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In F. Pfenning and Y. Smaragdakis, editors, *Generative Programming and Component Engineering*, pages 57–76, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-39815-8.

[7] 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.

[8] 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.

[9] 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.

[10] huceke. memcpy.c. https://github.com/huceke/xine-lib-vaapi/blob/master/src/xine-utils/memcpy.c. Accessed: 2025-05-26.

[11] 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.

[12] 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.

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

[14] O. Kammar, S. Lindley, and N. Oury. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, page 145–158, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323260. doi: 10.1145/2500365.2500590. URL https://doi.org/10.1145/2500365.2500590.

[15] 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.

[16] 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.

[17] 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.

[18] O. Kiselyov. MetaOCaml: Ten years later: System description. In *Functional and Logic Programming: 17th International Symposium, FLOPS 2024, Kumamoto, Japan, May 15–17, 2024, Proceedings*, page 219–236, Berlin, Heidelberg, 2024. Springer-Verlag. ISBN 978-981-97-2299-0. doi: 10.1007/978-981-97-2300-3_12. URL https://doi.org/10.1007/978-981-97-2300-3_12.

[19] 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.

[20] W. Kuchta. A proof of normalization for effect handlers, Sept. 2023. URL https://icfp23.sigplan.org/details/hope-2023/4/A-proof-of-normalization-for-effect-handlers. Seattle, Washington, United States.

[21] J. L. Lawall and O. Danvy. Continuation-based partial evaluation. *SIGPLAN Lisp Pointers*, VII(3):227–238, July 1994. ISSN 1045-3563. doi: 10.1145/182590.182483. URL https://doi.org/10.1145/182590.182483.

[22] 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.

[23] 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.

[24] G. Plotkin and N. Xie. Handling the selection monad (full version), 2025. URL https://arxiv.org/abs/2504.03890.

[25] 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).

[26] 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.

[27] 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.

[28] G. Scherer. Deciding equivalence with sums and the empty type. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, page 374–386, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346603. doi: 10.1145/3009837.3009901. URL https://doi.org/10.1145/3009837.3009901.

[29] 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.

[30] T. Sheard and S. P. Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, page 1–16, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581136056. doi: 10.1145/581690.581691. URL https://doi.org/10.1145/581690.581691.

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

[32] W. W. Tait. Intensional interpretations of functionals of finite type i. *The Journal of Symbolic Logic*, 32(2):198–212, 1967. ISSN 00224812. URL http://www.jstor.org/stable/2271658.

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

[34] 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.

[35] T. Walid. Multi-stage programming: Its theory and applications. Technical report, 1999.

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

[37] 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.

[38] 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.

[39] J. Yallop and O. Kiselyov. Generating mutually recursive definitions. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM 2019, page 75–81, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362269. doi: 10.1145/3294032.3294078. URL https://doi.org/10.1145/3294032.3294078.

[40] 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.

A. Auxiliary Definitions

A.1. Erase

The erase function takes a level 0 type and erases all level annotations, and elaborates effect rows. It is defined by straightforward induction on $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ types.

$$
\begin{array}{rcl}
\mathsf{erase}(\mathbb{N}^0) & = & \mathbb{N} \\
\mathsf{erase}((S^0 \xrightarrow{\xi} T^0)^0) & = & \mathsf{erase}(S^0) \xrightarrow{[\![\xi]\!]} \mathsf{erase}(T^0) \\
\mathsf{erase}((S^0 \xrightarrow{\xi}\!\!\!\!\rightarrow T^0)^0) & = & \mathsf{erase}(S^0) \xrightarrow{[\![\xi]\!]}\!\!\!\!\rightarrow \mathsf{erase}(T^0) \\
\\
\mathsf{erase}(T^0 \,!\, \xi) & = & \mathsf{erase}(T^0) \,!\, [\![\xi]\!] \\
\mathsf{erase}(T^0 \,!\, \Delta) & = & \mathsf{erase}(T^0) \,!\, [\![\Delta]\!] \\
\mathsf{erase}(T^0 \,!\, \Delta; \xi) & = & \mathsf{erase}(T^0) \,!\, [\![\Delta]\!]; [\![\xi]\!] \\
\\
\mathsf{erase}((S^0 \,!\, \xi_1 \Longrightarrow T^0 \,!\, \xi_2)^0 \,!\, \Delta) & = & (\mathsf{erase}(S^0) \,!\, [\![\xi_1]\!] \Longrightarrow \mathsf{erase}(T^0) \,!\, [\![\xi_2]\!]) \,!\, [\![\Delta]\!]
\end{array}
$$

B. Litmus Tests

B.1. MetaOCaml

This section transcribes Listings 13, 14, 16 and 17 in BER MetaOCaml N153.

To test these programs, as of 05 June 2025, one must first set up MetaOCaml to perform scope extrusion checking.

```MetaOCaml
let () =  Trx.set_with_stack_mark
  {Trx.stackmark_region_fn =
      fun body ->
        let module M = struct type _ Effect.t += E: unit t end in
        try body (fun () -> try perform M.E; true with Unhandled _ -> false)
        with effect M.E, k -> continue k ()}
```

```MetaOCaml
run (let z  = match .<fun x -> .~(perform (Extrude .<x>.) ) >.
             with u -> .<0>.
                | effect (Extrude y), k -> y
     in .< .~z + 1 >.)
```

Listing 23: Listing 13, transcribed in MetaOCaml

```MetaOCaml
let z = run (match .<fun x -> .~(perform (Extrude .<x>.) ) >.
             with u -> .<0>.
                | effect (Extrude y), k -> y)
in z + 1
```

Listing 24: Listing 14, transcribed in MetaOCaml

```MetaOCaml
run (match .<fun x -> .~(perform (Extrude .<x>.) ) >.
             with u -> .<0>.
                | effect (Extrude y), k -> let _ = y in .<0>. )
```

Listing 25: Listing 16, transcribed in MetaOCaml

```MetaOCaml
run (match .<fun x -> .~(perform (Extrude .<x>.) ) >.
             with u -> .<0>.
                | effect (Extrude y), k -> continue k y )
```

Listing 26: Listing 17, transcribed in MetaOCaml

MetaOCaml

```
run (match .<fun x -> .~(perform (Extrude .<x>.) ) >.
             with u -> .<0>.
                | effect (Extrude y), k -> continue k .< .~y + 1>. )
```

Listing 27: Listing 18, transcribed in MetaOCaml