Handling Scope Checks

A Comparative Framework for Dynamic Scope Extrusion Checks

MICHAEL LEE, University of Cambridge, United Kingdom
NINGNING XIE, University of Toronto, Canada
OLEG KISELYOV, Tohoku University, Japan
JEREMY YALLOP, University of Cambridge, United Kingdom

Metaprogramming and effect handlers interact in unexpected, and sometimes undesirable, ways. One example is scope extrusion: the generation of ill-scoped code. Scope extrusion can either be preemptively prevented, via static type systems, or retroactively detected, via dynamic checks. Static type systems exist in theory, but struggle with a range of implementation and usability problems in practice. In contrast, dynamic checks exist in practice (e.g. in MetaOCaml), but are understudied in theory. Designers of metaprogramming languages are thus given little guidance regarding the design and implementation of checks. We present the first formal study of dynamic scope extrusion checks, introducing a calculus ($\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$) for describing and evaluating checks. Further, we introduce a novel dynamic check — the "Cause-for-Concern" check — which we prove correct, characterise without reference to its implementation, and argue combines the advantages of existing dynamic checks. Finally, we extend our framework with refined environment classifiers, which statically prevent scope extrusion, and compare their expressivity with the dynamic checks.

CCS Concepts: • **Software and its engineering** → *Control structures.*

Additional Key Words and Phrases: effect handlers, code generation, metaprogramming, scope extrusion

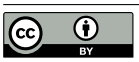
ACM Reference Format:
Michael Lee, Ningning Xie, Oleg Kiselyov, and Jeremy Yallop. 2026. Handling Scope Checks: A Comparative Framework for Dynamic Scope Extrusion Checks. *Proc. ACM Program. Lang.* 10, POPL, Article 39 (January 2026), 30 pages. https://doi.org/10.1145/3776681

1 Introduction

Multi-stage programming languages have been used to write code generators for a wide variety of domains, from database queries and stream processing to geometry, parsing, and differentiable programming [Carette et al. 2011; Kiselyov et al. 2017; Rompf and Amin 2015; Wang et al. 2019; Yallop et al. 2023]. Language constructs for code generation often come with strong guarantees. For example, a well-typed code generator written in the MetaML language [Taha 1999] is guaranteed never to generate ill-typed code. However, these guarantees are weakened when code generation constructs are combined with effects [Calcagno et al. 2000; Isoda et al. 2024; Kameyama et al. 2015, 2011; Kiselyov 2014; Kiselyov et al. 2016; Parreaux 2020].

In particular, the combination of code generation constructs and effects can lead to *scope extrusion*: the inadvertent generation of code with unbound variables. For example, in the MacoCaml program Listing 1, the use of effect handlers in code generation extrudes the variable x beyond its scope:

Authors' Contact Information: Michael Lee, michael.lee@cl.cam.ac.uk, University of Cambridge, United Kingdom; Ningning Xie, ningningxie@cs.toronto.edu, University of Toronto, Canada; Oleg Kiselyov, oleg@okmij.org, Tohoku University, Japan; Jeremy Yallop, jeremy.yallop@cl.cam.ac.uk, University of Cambridge, United Kingdom.



ACM 2475-1421/2026/1-ART39
https://doi.org/10.1145/3776681

```
MacoCaml
1  try << let x = 3 in $(perform (Extrude <<x>>)) >>
2  with effect Extrude y, k → << $y + 1 >>
```

Listing 1. An example of scope extrusion

Line 1 installs a handler whose body `<< let x = 3 in $(...) >>` uses *code quotation* to construct code for a function application. The expression within quotation marks `<< >>` is not evaluated immediately, but constructs a piece of code that may be evaluated in the future. However, the sub-expression prefixed by `$` is evaluated immediately, and performs an effect **`Extrude`**, transferring control to the most recently installed handler. Line 2 shows the handler, which binds the argument `<<x>>` to `y` and the continuation delimited by **`try`** and **`perform`** to `k`. The handler discards the continuation and uses the argument to construct the code `<< x + 1 >>`, in which `x` is unbound.

In Listing 1 the extrusion is simple: `<<x>>` leaves the scope of its binder `<<let x = 3 in ...>>` and never returns. However, handlers that invoke `k` might cause control to re-enter the scope:

```
MacoCaml
2  with effect Extrude y, k → continue k << $y + 1 >>
```

Listing 2. Revising the handler of Listing 1 to bring x back into scope

Here **`continue`** resumes the continuation, returning control to the point where **`perform`** was invoked, so that the program ultimately evaluates to a well-scoped code value `<< let x = 3 in x + 1 >>`.

Scope extrusion is a problem in practice as well as in theory. The strong guarantees attached to multi-stage languages relieve programmers of the burden of debugging type errors in generated code, but scope extrusion reintroduces the burden. Ofenbeck et al. [2016] report an example: refactoring the LMS system to address performance issues used effects to perform code motion optimizations that inadvertently led to scope extrusion errors. These errors had simple causes, but were time-consuming to fix due to the large number of variables involved, and the difficulty of determining which part of the code generator produced the offending code.

To avoid the need for programmers to debug generated code, multi-stage languages with effects often provide help in identifying scope extrusion. The key question is *when* to check for problems. One approach is to track potential extrusion in the type system, rejecting programs that cannot be shown to be safe [Calcagno et al. 2000; Isoda et al. 2024; Kiselyov et al. 2016; Parreaux 2020; Westbrook et al. 2010]. In practice, however, it is difficult to combine expressiveness that allows virtuous interactions between effects and code generation (e.g. code motion optimizations) with strictness that excludes all potential extrusion. Given the choice between such sophisticated type systems and simpler but less safe systems, users tend to prefer the latter [Parreaux 2020].

The other approach is to check dynamically during code generation, allowing potentially unsafe code generators to run, and identifying extrusion as it occurs. This more liberal approach does not have an existing theory, but it is more common in practice, in part because it can be incorporated into existing multi-stage languages such as MetaOCaml, Scala [Stucki et al. 2018] and Typed Template Haskell [Xie et al. 2022] without disruption to their type systems.

There is a range of possible designs for dynamic checks. At one extreme, scope is checked lazily, once code generation is complete. The original MetaOCaml language used lazy checking, since its static type system, *environment classifiers* [Taha and Nielsen 2003], which prevented some forms of extrusion, could not prevent every case. At the other extreme, BER MetaOCaml checks scope eagerly each time a quotation is constructed [Kiselyov 2014, 2024a]. Neither approach is optimal. Lazy checking is *uninformative* [Ofenbeck et al. 2016], producing hard-to-debug errors, *inefficient*, reporting errors much later than eager checking [Kiselyov 2014], and in some systems can bind

variables in *unintended ways* [Kameyama et al. 2015]. On the other hand, eager checking is *incorrect* in a sense that we explicate in §4.3.1, failing to detect occurrences of free variables in certain pathological cases (Listings 8 and 9). Further, eager checking is *not continuation-aware*: for example, it incorrectly rejects the safe code generator in Listing 2 [Kiselyov 2014].

To establish a theory of dynamic checks, we introduce the $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ and $\lambda_{\mathsf{AST(op)}}$ calculi that support multi-stage programming with effects and handlers, and show how they can be used to describe and compare eager and lazy checking. We also describe a new check, the Cause-for-Concern (C4C) check, implemented in MacoCaml, that combines the advantages of eager and lazy checking.

Contributions. §2 presents the eager and lazy approaches informally using a larger example, and introduces our novel C4C check. The subsequent sections present technical contributions:

- Two novel calculi, $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ and $\lambda_{\mathsf{AST(op)}}$, designed for the study of typed multi-stage programming with effects and handlers (§3). $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ is a type safe two-stage calculus, and is the first calculus to support effect handlers at both compile-time and run-time stages.
- A framework based on $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ and $\lambda_{\mathsf{AST(op)}}$ that facilitates formalization and evaluation of different scope extrusion checks as a family of elaborations from $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ to $\lambda_{\mathsf{AST(op)}}$ (§4). We use the framework to study a variety of designs: a lazy check (§4.2), the eager check (§4.3), and our novel C4C check (§4.4).
- An extension of $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ and $\lambda_{\mathsf{AST(op)}}$ with Kiselyov et al.'s [2016] *refined environment classifiers* (§5.1), with a proof of correctness via a logical relation (§5.2), and an evaluation of its expressiveness compared to the dynamic checks (§5.3).
- Implementations of the three dynamic checks in the MacoCaml language (§6). An implementation with the C4C check is available as an artifact [Lee et al. 2025].

Finally, §7 presents related work and §8 concludes.

2 Overview

While there are many metaprogramming languages, our discussion will be grounded in MacoCaml [Chiang et al. 2024; Xie et al. 2023]. The MacoCaml project extends the OCaml programming language with metaprogramming facilities for compile-time program generation: a type constructor α `expr` for code of type α, and quote `<< >>` and splice `$` forms for constructing `expr` values.

At a high level, elements of α `expr` correspond to ASTs of type α. Quotation converts expressions to ASTs, and splices stop the conversion, allowing evaluation during AST creation. As an example, the metaprogram `<<$(print_int (1+2); <<1+2>>) + 0>>` can be thought of as `Plus((print_int (1+2); Plus(Int(1), Int(2))), Int(0))`. This conceptual model will be made precise in §3.

Listing 3 shows our running example, adapted from Kiselyov [2014], which generates code for matrix multiplication. The parameters `a`, `b`, `c` are two-dimensional arrays; `a.(0)` accesses the array representing first row of the matrix. Realistic implementations of matrix multiplication typically employ various sophisticated optimizations, but this simple code will be sufficient to highlight the interaction between code generation and effects that is the focus of this paper.

MacoCaml

```
1  macro iter a body = << for i = 0 to length $a - 1 do $(body <<i>>) done >>
2  macro mmul a b c =
3    iter a @@ fun i →
4    iter <<$a.(0)>> @@ fun k →
5    iter <<$b.(0)>> @@ fun j →
6       << $c.($i).($j) ← $c.($i).($j)
7                       + $a.($i).($k)
8                       * $b.($k).($j) >>
```

Listing 3. Staged code that generates the familiar matrix multiplication code

Line 1 defines a macro (i.e. a compile-time function) that generates code for a `for` loop from the code for a term of array type `a` and the result of calling the `body` function with the loop variable `i`. Lines 3–9 define a second macro `mmul` that uses `iter` to construct a triply-nested loop. The `@@` operator denotes function application, and is used to avoid proliferation of parentheses.

2.1 Effect Handlers in Staging

Effect handlers are a composable and customisable mechanism for simulating effects [Pretnar 2015]. As with metaprogramming, there are many variants of effect handlers, and we ground our discussion in deep, unnamed effect handlers that permit multi-shot continuations (a calculus is presented in §3.1). In this section, our examples use OCaml's deep, unnamed effect handlers, which permit only single-shot continuations [Sivaramakrishnan et al. 2021].

Given the utility of metaprogramming and effect handlers, it is wise to consider how a language that offers both might mediate their interaction. Complete separation may be undesirable, since effects are very useful for relaxing the stack discipline that would otherwise tightly couple the structure of the generated and generating code. Concretely, effects allow the programmer to easily perform *let-insertion* [Kameyama et al. 2011; Yallop and Kiselyov 2019] (Listing 4):

MacoCaml

```
1  type _ Effect.t += Genlet : int expr * int expr → int expr t
2  macro genlet x e = perform (Genlet (x, e))
3  macro handle_genlet body i = try body i
4                               with effect Genlet (x, e), k →
5                                 if x == i then <<let y = $e in $(continue k <<y>>)>>
6                                 else continue k (genlet x e)
```

Listing 4. Effect handlers and quotes and splices combine to perform let-insertion

Lines 1–2 define a new effect constructor `Genlet` and a compile-time function that performs the `Genlet` effect. The `Genlet` effect takes two arguments: the first identifies the insertion point for the new binding, and second the expression to be bound. Lines 3–5 define a handler for `Genlet` that either installs a `let` binding on the stack (line 5) or forwards the effect to an outer handler (line 6). The invocation `handle_genlet body i` wraps the code generated by `body i` with a `let` binding for each call to `genlet i e` that takes place during the execution of `body`.

In the `mmul` example (Listing 3), the expression `$a.($i).($k)` does not depend on `j`, and therefore can be lifted out of the loop, an optimisation known as *loop-invariant code motion*. Effects are a convenient way to perform these types of optimisations in staged programs while maintaining the structure of the generating code. Without effects, the stack discipline couples the structures of the generating and generated code, so that code motion requires updating the generator to lift the expression `$a.($i).($k)` above the `iter <<$b.(0)>>` expression.

Listing 5 shows the example updated to use let-insertion. On Line 1, `handle_genlet` now installs a `Genlet` handler while generating each loop, and Lines 7 and 8 now perform the `genlet` effect.

MacoCaml

```
1  macro iter a body = << for i = 0 to length $a - 1 do $(handle_genlet body <<i>>) done >>
2  macro mmul a b c =
3    iter a @@ fun i →
4    iter <<$a.(0)>> @@ fun k →
5    iter <<$b.(0)>> @@ fun j →
6       << $c.$i.$j ← $c.($i).($j)
7                   + genlet k <<$a.($i).($k)>> (*instead of $a.($i).($k)*)
8                   * genlet j <<$b.($k).($j)>> (*instead of $b.($k).($j)*) >>
```

Listing 5. Effect handlers and quotes and splices combine to perform let-insertion

Unfortunately, it is easy to make mistakes when performing an optimisation of this kind, leading to *scope extrusion*. Assume that, given how arrays are laid out in memory, and the specific design of the cache prefetcher, it is more efficient to interchange the `j` and `k` loops:

```
MacoCaml
3  iter a @@ fun i →
4  iter <<$b.(0)>> @@ fun j →
5  iter <<$a.(0)>> @@ fun k → ...
```

Should the programmer realise this, they may perform this interchange without changing the let-insertion code. But the use of `genlet` in Listing 5 assumes that the `k` loop is above the `j` loop, so this change would result in scope extrusion.

Alternatively, the programmer may identify the wrong let insertion point:

```
MacoCaml
6  << $c.$i.$j ← $c.($i).($j)
7              + genlet i <<$a.($i).($k)>> (*should be k, not i*)
8              * genlet k <<$b.($k).($j)>> (*should be j, not k*) >>
```

This mistake also results in scope extrusion. Detecting such errors requires a scope extrusion check.

2.2 Checking for Scope Extrusion

In theory, it is possible to adopt a **lazy check** (§4.2), which waits until the end of the program generation process to check that the generated program contains no free variables [Kiselyov 2014]. Lazy checking amounts to type checking generated code, an approach used in the original MetaOCaml implementation [Taha and Nielsen 2003], LMS [Ofenbeck et al. 2016], and other systems. The lazy approach has two drawbacks: it is inefficient, since it allows a code generator to run to completion after an error has occurred, and it produces uninformative error messages that refer to the generated code rather than the code generator.

Therefore, BER MetaOCaml instead adopts an **eager check** [Kiselyov 2014, 2024a], which we describe formally in §4.3. By checking at various points during the code generation process, the eager check identifies the error early and raises an informative error message [Kiselyov 2014, §5.1].

While the eager check provides better error reporting than the lazy check, it does not allow effect handlers and metaprogramming to interact as freely as one might desire. For example, a common use-case for effect handlers is *parameterisation*: by choosing different handlers for the same effect, the same piece of code can be specialised in various contexts [Wang et al. 2019]. Parameterisation uses effect handlers in a very simple way; we use it in our example to show that even straightforward uses of effect handlers interact poorly with the eager check.

To extend the matrix multiplication generator with parameterisation, suppose that we wish to generate `e1 + e2 * e3` by default, and generate a call to a fused multiply-add instruction `__fma(e1, e2, e3)` in contexts where performance takes priority over preserving exactly the expected floating point behaviour. Listing 6 shows one way to parameterise over these alternatives, first abstracting the choice as an effect, `FMA` (lines 1-2), then modifying the body of `mmul` to perform the effect (lines 6–8), then defining handlers that can be used to tune the generation process (lines 10–13), and finally installing a handler around the call to `mmul` (line 15). Line 15 makes use of a *top-level splice*, which is used in MacoCaml to insert the code generated by a macro into a larger program.

Unfortunately, the use of `FMA` in Listing 6 is *not* allowed by the eager check, which throws a scope extrusion error when Line 11 or Line 13 is executed. For example, in the body of the `hdl_fma_def` handler, the code template `<<$x + $y * $z>>` is evaluated in a scope where `i`, `j`, and `k` are free. However, as `x` is bound to `<<c.($i).($j)>>`, the eager check reports scope extrusion.

```
MacoCaml
type _ Effect.t += FMA : int expr * int expr * int expr → int expr t
macro fma x y z = perform (FMA (x, y, z))

macro mmul a b c =
  ...
    << $c.($i).($j) ← $(fma <<$c.($i).($j)>>
                            (genlet k <<$a.($i).($k)>>)
                            (genlet j <<$b.($k).($j)>>))>>

macro hdl_fma_def body = try body ()
                        with effect FMA (x, y, z), k → continue k << $x + $y * $z >>
macro hdl_fma_opt body = try body ()
                        with effect FMA (x, y, z), k → continue k << __fma($x, $y, $z) >>

let code = $(hdl_fma_def @@ fun () → mmul <<a>> <<b>> <<c>>)
```

Listing 6. Handlers for selecting a multiply-and-add instruction

To support more flexible interaction between effect handlers and metaprogramming, we introduce a novel continuation-aware **C4C** check, explained in detail in §4.4, which allows code like Listing 6 to run to completion without reporting scope extrusion.

3 Calculus

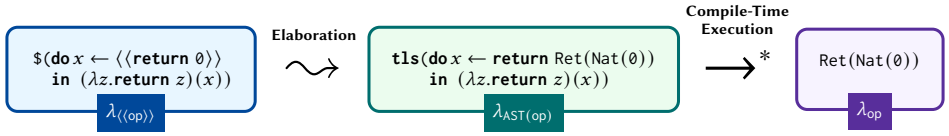


Fig. 1. $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ is first elaborated into $\lambda_{\mathsf{AST}(\mathsf{op})}$, which is then executed **at compile-time** to obtain the AST of a run-time λ_{op} program. **tls** is a marker which tracks the position of the **t**op-**l**evel **s**plice (§3.2)

To ground the discussion of dynamic scope extrusion checks, we introduce two novel calculi for studying the interaction between typed multi-stage programming and effects and handlers: $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ and $\lambda_{\mathsf{AST}(\mathsf{op})}$. $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ (§3.1) offers metaprogramming in the form of quotes and splices, and effect handlers. $\lambda_{\mathsf{AST}(\mathsf{op})}$ offers metaprogramming in the form of AST constructors, and effect handlers. Following Calcagno et al. [2003], $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ has no operational semantics; programs in $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ are instead elaborated into $\lambda_{\mathsf{AST}(\mathsf{op})}$ (§3.3), where they may then be executed, to obtain the AST of a run-time program that has no quotes and splices. This process is summarised in Figure 1. Elaboration simplifies the operational semantics, and is a convenient mechanism for inserting dynamic checks (§4). $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ and $\lambda_{\mathsf{AST}(\mathsf{op})}$ are both type safe (§3.4).

3.1 The Source Language: $\lambda_{\langle\langle \mathbf{op} \rangle\rangle}$

$\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ (Figure 2) is a language which offers both metaprogramming, in the form of quotes $\langle\langle e \rangle\rangle$ and splices $\$e$, as well as effect handlers [Pretnar 2015]. Syntactic $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ terms are divided into values, expressions, and handlers, similar to a fine-grained call-by-value approach [Levy et al. 2003]. Ignoring quotes and splices, and adding a continuation term former, $\kappa x.e$ that cannot be written explicitly but may be generated during reduction, one obtains the syntax of a standard base calculus of effects and handlers [Biernacki et al. 2017; Isoda et al. 2024; Pretnar 2015], which we refer to as λ_{op} (and which is described in an appendix in the extended version). Briefly, **return** v lifts a value into an expression, and **do** $x \leftarrow e_1$ **in** e_2 sequences expressions. **op**(v) performs an effect,

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

Syntax

Values	v	::=	$x \mid m \in \mathbb{N} \mid \lambda x.e$
Expressions	e	::=	$v_1\ v_2 \mid \mathbf{return}\ v \mid \mathbf{do}\ x \leftarrow e_1\ \mathbf{in}\ e_2 \mid \mathbf{op}(v) \mid \mathbf{handle}\ e\ \mathbf{with}\ \{h\} \mid \mathbf{continue}\ v_1\ v_2$ $\mid \langle\langle e \rangle\rangle \mid \$e$
Handlers	h	::=	$\mathbf{return}(x) \mapsto e \mid h; \mathbf{op}(x, k) \mapsto e$

Effect sets

Run-Time $\quad \xi ::= \emptyset \mid \xi \cup \{\mathrm{op}_i^0\}$

Compile-Time $\quad \Delta ::= \emptyset \mid \Delta \cup \{\mathrm{op}_i^{-1}\}$

Typing contexts

$\Gamma ::= \cdot \mid \Gamma, x : T^0 \mid x : T^{-1}$

Types

Level 0

Values $S^0, T^0 ::=$	$\mathbb{N}^0 \mid (S^0 \xrightarrow{\xi} T^0)^0$ $\mid (S^0 \xrightarrow{\xi\!\!\!\!\!\rightarrow} T^0)^0$
Computations	$T^0 \,!\, \xi \mid T^0 \,!\, \Delta \mid T^0 \,!\, \Delta; \xi$ $\mid (S^0 \,!\, \xi_1 \Longrightarrow T^0 \,!\, \xi_2)^0 \,!\, \Delta$
Handlers	$(S^0 \,!\, \xi_1 \Longrightarrow T^0 \,!\, \xi_2)^0$

Level −1

Values $S^{-1}, T^{-1} ::=$	$\mathbb{N}^{-1} \mid (S \xrightarrow{\Delta} T)^{-1}$ $\mid (S \xrightarrow{\Delta\!\!\!\!\!\rightarrow} T)^{-1} \mid \mathsf{Code}(T^0 \,!\, \xi)^{-1}$
Computations	$T^{-1} \,!\, \Delta$
Handlers	$(S^{-1} \,!\, \Delta_1 \Longrightarrow T^{-1} \,!\, \Delta_2)^{-1}$

Fig. 2. $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ syntax and types

suspending the current computation and throwing a value v to be caught by some handler h that was installed using **handle** e **with** $\{h\}$. Within the body of the handler, **continue** $k\ v$ can be used to resume the suspended program (k), inserting the value v in place of the performed effect.

Metaprogramming systems differ along several key dimensions: they can be homogeneous (where the generating and generated languages coincide) or heterogeneous, two-stage or multi-stage, compile-time or run-time [Lilis and Savidis 2019]. $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ offers *homogeneous, two-stage, compile-time* metaprogramming. Many practical systems, like MacoCaml and MetaOCaml, are homogeneous. Many practical use cases of MSP involve only two stages [Inoue and Taha 2012], and scope extrusion is often studied in two stage systems [Isoda et al. 2024; Kiselyov et al. 2016]. Similarly, $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ offers *deep, unnamed handlers* that *permit multi-shot continuations*, modelling OCaml effect handlers, though generalised to multi-shot continuations. Multi-shot continuations, though not supported by OCaml, are useful for if/case insertion [Yallop 2017]. Other effect systems also allow for shallow or sheep handlers, named handlers, or permit only one-shot continuations [Yallop and community contributors 2025]: we do not study these systems.

Following Calcagno et al. [2003], $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ has no operational semantics, but is instead elaborated into $\lambda_{\mathsf{AST}(\mathsf{op})}$. $\lambda_{\mathsf{AST}(\mathsf{op})}$ programs may then be executed, to obtain the AST of a run-time λ_{op} program that has no quotes and splices. This process is summarised in Figure 1. Elaboration simplifies the operational semantics and is a convenient mechanism for inserting dynamic checks (§4).

Only expressions can be quoted (values and handlers 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 $\mathbf{do}\ a \leftarrow \langle\langle \mathbf{return}\ 1 \rangle\rangle\ \mathbf{in}\ \mathbf{op}(a)$ rather than $\mathbf{op}(\ \langle\langle \mathbf{return}\ 1 \rangle\rangle\ )$. However, we 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. Figure 2 summarises the $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ types. To motivate the type system, consider the following running example e in $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ extended with arithmetic:

$$e \triangleq (\lambda x.\mathbf{get}(); \langle\langle \mathbf{do}\ y \leftarrow \$x\ \mathbf{in}\ \mathbf{readInt}() + y \rangle\rangle)$$

Table 1. Stratification of level 0 types

		Run-Time		
Cannot be directly created ↓		**Value**	**Computation**	**Handler**
Compile-Time	**Value**	T^0	$T^0 \,!\, \xi$	$(S^0 \,!\, \xi_1 \Longrightarrow T^0 \,!\, \xi_2)^0$
	Computation	$T^0 \,!\, \Delta$	$T^0 \,!\, \Delta; \xi$	$(S^0 \,!\, \xi_1 \Longrightarrow T^0 \,!\, \xi_2)^0 \,!\, \Delta$
		↑ Type of syntactic level 0 values v	↑ Type of syntactic level 0 expressions e	↑ Type of syntactic level 0 handlers h

Here e is (1) a compile-time function that (2) takes the AST of a run-time computation of type $\mathbb{N} \,!\, \{\texttt{print}\}$, (3) performs a compile-time effect (**get**), and (4) returns the AST of a run-time computation of a different type. The program has the following type:

Compile-time effects ③ — Compile-time function ①

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

② Input: AST of a run-time computation of type $\mathbb{N} \,!\, \{\texttt{print}\}$

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

The type system stratifies types into compile-time (T^{-1}) and run-time (T^0) levels. The function e has a compile-time type $(S \to T)^{-1}$, and cannot be used as a run-time function of type $(S \to T)^0$.

To support compile-time manipulation of run-time programs, the $\text{Code}(T^0 \,!\, \xi)^{-1}$ type makes ASTs of level 0 computations available at level -1. Only *computations*, not values or handlers, can be turned into ASTs.

Effect sets are stratified into Δ (compile-time) and ξ (run-time). In our running example, suppose e is applied at compile-time to some term e' of the right type. The application has a compile-time effect `get`, and returns an AST with two run-time effects, `print` and `readInt`.

$$\Gamma \vdash^{-1}_{\mathbf{s}} e\ e' : \text{Code}(\mathbb{N}^0 \,!\, \{\texttt{print}, \texttt{readInt}\})^{-1} \,!\, \{\texttt{get}\}$$

Splicing the result of application lifts the compile-time AST into a run-time type that has unhandled effects at **both** compile-time and run-time.

$$\Gamma \vdash^{0}_{\mathbf{q}} \$(e\ e') : \mathbb{N}^0 \,!\, \{\texttt{get}\}; \{\texttt{print}, \texttt{readInt}\}$$

We track compile-time and run-time effects in separate sets. Compile-time effects are tracked in $\Delta (= \{\texttt{get}\})$ and run-time effects in $\xi (= \{\texttt{print}, \texttt{readInt}\})$. Distinguishing compile-time and run-time effects stratifies types (Table 1): what is a computation at run-time could have been a value at compile-time, and vice versa.

In $\lambda_{\langle\langle \texttt{op} \rangle\rangle}$, the use of a term typed at level 0 *always* results in compile-time computation (the second row of Table 1). For example, level 0 values are elaborated into compile-time computations ($T^0 \,!\, \Delta$) that evaluate to ASTs of run-time values (§3.3):

$T^0 \,!\, \Delta$	Compile-time computation, run-time value *Inhabitants:* Level 0 values v, e.g. $\lambda x.$**return** x
$T^0 \,!\, \Delta; \xi$	Compile-time computation, run-time computation *Inhabitants:* Level 0 expressions e, e.g. **return** 1
$(S^0 \,!\, \xi_1 \Longrightarrow T^0 \,!\, \xi_2)^0 \,!\, \Delta$	Compile-time computation, run-time handler *Inhabitants:* Level 0 handlers h, e.g. $\{$**return**$(x) \mapsto$ **return** $x\}$

Table 2. The nine $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ typing judgements

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

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

Fig. 3. A metaprogram annotated with compiler modes

Consequently, *syntactic* values (v) at level 0 do not have value *type* (T^0). The relationship between syntax and types is more complicated than in λ_{op}. In contrast, level 0 compile-time value types (the first row in Table 1) have no inhabitants in $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ (but T^0 is used to type *formal parameters* of functions at level 0, like x in $\lambda x.\mathbf{return}\ 0$).

As the stratification is subtle, it is best revisited after covering the typing rules (§3.1.1), core language (§3.2), and elaboration (§3.3).

Selected $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ typing rules are collated in Figure 4. Similar to Xie et al. [2023], typing judgements are indexed by one of three compiler modes: **Compile** (c), **Quote** (q), or **Splice** (s). However, unlike Xie et al. [2023], typing judgements do not need to be indexed by a level: since $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ is a two-level system, each compiler mode uniquely determines a level ($\mathbf{c}|\mathbf{q} \mapsto 0$, $\mathbf{s} \mapsto -1$). For each mode, there are three typing judgements: one for each syntactic category (Table 2).

Modes are useful for elaboration. **c** identifies code that is **ambient** and **inert** (no surrounding quotes or splices). **s** identifies code that **manipulates ASTs** at compile-time (last surrounding annotation is a splice). **q** identifies code whose **ASTs are manipulated** at compile time (last surrounding annotation is a quote). Accordingly, top-level splices transition from **c** to **s**. Quotes transition from **s** to **q**. Splices ($\$e$) transition from **q** to **s**. Figure 3 annotates a metaprogram (that evaluates to the AST of $\lambda x.1 + 2 + 3$) with modes.

The typing judgements for **c** and **q** are identical in almost all cases. To avoid repetition, we introduce the notation $\Gamma \vdash_{\mathbf{c}|\mathbf{q}} e : T$ to stand for the two judgements $\Gamma \vdash_{\mathbf{c}} e : T$ and $\Gamma \vdash_{\mathbf{q}} e : T$. The mode of the conclusion will match the modes of the assumption, unless otherwise stated.

The $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ typing rules are mostly standard for a calculus with effect handlers . In **c** and **q**, compile-time effects Δ are threaded through typing judgements, and only level 0 variables in the context can be accessed. In **s**, only level -1 variables can be accessed. As the levels of types can, in most cases, be inferred: for readability, they too are mostly omitted. The three key rules are **s**-Quote, **q**-Splice, and **c**-Splice, which switch between modes and levels.

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 effect sets: all effects are provably handled, both at compile-time and run-time.

Definition 3.1 (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: $\lambda_{\mathsf{AST}(\mathsf{op})}$

$\lambda_{\mathsf{AST}(\mathsf{op})}$ (Figure 5) is a language which offers AST constructors and effect handlers. Syntax is divided into normal forms, terms, and handlers. The syntax of $\lambda_{\mathsf{AST}(\mathsf{op})}$ combines a standard calculus of effect handlers (λ_{op}) with machinery for AST construction, and primitives for scope extrusion checking.

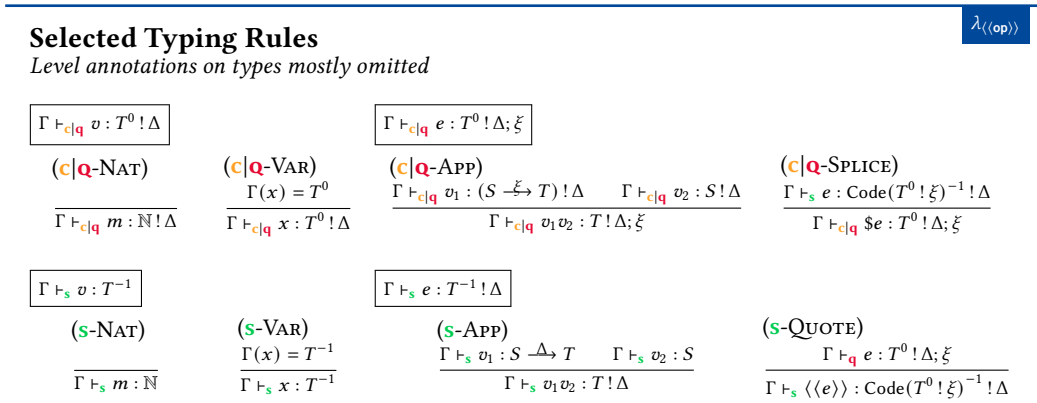


Fig. 4. Selected typing rules for $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$.

$\lambda_{\mathsf{AST}(\mathsf{op})}$'s machinery for AST construction comprises **AST** nodes for each λ_{op} term former that can be written by the user (e.g. `Var` for variables, H_{op} for $\_;\mathbf{op}(\_,\_) \mapsto \_$), as well as *type-annotated* **formal parameters** (α_R, where R is some run-time value pre-type (Figure 5), henceforth simply "type"). Formal parameters represent binding sites, e.g. x in $\lambda x.\mathbf{return}\ 0$. Separating ASTs and formal parameters mirrors the approach by Calcagno et al. [2003], though they use untyped formal parameters. Additionally, $\lambda_{\mathsf{AST}(\mathsf{op})}$ adds **mkvar** R, a primitive for generating fresh formal parameters of type R, α_R, where separate calls to **mkvar** return distinct formal parameters [Taha 1999].

$\lambda_{\mathsf{AST}(\mathsf{op})}$'s machinery for scope extrusion checking comprises:

- **err**, an error state for indicating the presence of scope extrusion,
- **check** and $\mathbf{check}_{\mathsf{M}}$, guarded **return**s that either report scope extrusion or **return** normally,
- **dlet**, a primitive for tracking which variables are well-scoped and which have extruded their scope, and
- **tls**, a marker representing an occurrence of a top-level splice in the source program: at this point, remaining stack frames either introduce a new top-level splice, or construct an AST in an entirely straightforward way, with standard (and thus safe) control flow.

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 operational semantics of $\lambda_{\mathsf{AST}(\mathsf{op})}$ is defined over configurations $\langle t; E; U; M; I \rangle$. At a high level, t are terms and E are evaluation contexts, defined as a stack of evaluation frames, à la Felleisen et al. [1988]. U acts as a source of fresh names. 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 $\emptyset$. Collectively, M and I determine whether to perform the check immediately ($M = \emptyset$), or defer it to a later point (marked by I). Deferring checking in the presence of a continuation that could later be used to recover from scope extrusion is used by the C4C check, making it "continuation-aware". The semantics for the lazy and eager checks can be more simply given as 3-tuple transition systems, which are straightforward projections of the 5-tuple system used to compare the three checks.

The operational semantics is mostly as expected for a calculus with effect handlers. Interesting rules are collated in Figure 5, and full rules in an appendix in the extended version.

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

Syntax

Formal Params	α_R		
Normal Forms	n	::=	$x \mid m \in \mathbb{N} \mid \lambda x.t \mid \kappa x.t \mid \mathsf{Nat}(m) \mid \alpha_R \mid \mathsf{Var}(\alpha_R) \mid \mathsf{Lam}(n_1, n_2) \mid \mathsf{App}(n_1, n_2)$ $\mid \mathsf{Continue}(n_1, n_2) \mid \mathsf{Ret}(n) \mid \mathsf{Do}(n_1, n_2, n_3) \mid \mathsf{Op}(n) \mid \mathsf{Hwith}(n_1, n_2)$ $\mid \mathsf{Hret}(n_1, n_2)(n_1, n_2) \mid \mathsf{Hop}(n_1, n_2, n_3, n_4)$
Terms	t	::=	$n_1\ n_2 \mid \mathbf{return}\ n \mid \mathbf{do}\ x \leftarrow t_1\ \mathbf{in}\ t_2 \mid \mathbf{op}(t) \mid \mathbf{handle}\ n\ \mathbf{with}\ \{h\} \mid \mathbf{continue}\ n_1\ n_2$ $\mid \mathbf{check}\ n \mid \mathbf{check}_{\mathsf{M}}\ n \mid \mathbf{mkvar}\ R \mid \mathbf{dlet}(n, t) \mid \mathbf{tls}(t) \mid \mathbf{err}$
Handlers	h	::=	$\mathbf{return}(x) \mapsto t \mid \mathbf{op}(x, k) \mapsto t$

Typing contexts

$\Gamma ::= \cdot \mid \Gamma, x : T$

Types

Run-time Pre-types		**Types**	
Effects set ξ ::=	$\emptyset \mid \xi \cup \{\mathsf{op}_i\}$	Effects set	$\Delta ::= \emptyset \mid \Delta \cup \{\mathsf{op}_i\}$
Value type Q, R ::=	$\mathbb{N} \mid Q \xrightarrow{\xi} R$	Value type S, T ::=	$\mathbb{N} \mid S \xrightarrow{\Delta} T \mid S \xrightarrow{\Delta}\!\!\!\!\rightarrow T$ $\mid \mathsf{FParam}(R) \mid \mathsf{AST}(R)$
	$\mid Q \xrightarrow{\xi}\!\!\!\!\rightarrow R$		$\mid \mathsf{AST}(R\,!\,\xi) \mid \mathsf{AST}(Q\,!\,\xi_1 \Longrightarrow R\,!\,\xi_2)$
Computation type	$R\,!\,\xi$	Computation type	$T\,!\,\Delta$
Handler type	$Q\,!\,\xi_1 \Longrightarrow R\,!\,\xi_2$	Handler type	$S\,!\,\Delta \Longrightarrow T\,!\,\Delta_2$

Operational Semantics

Selected Rules

Auxiliary Definitions

Evaluation Frame F ::= $\mathbf{do}\ x \leftarrow [-]\ \mathbf{in}\ t_2 \mid \mathbf{handle}\ [-]\ \mathbf{with}\ \{h\} \mid \mathbf{dlet}(\alpha_R, [-]) \mid \mathbf{tls}([-])$

Evaluation Context E ::= $[-] \mid E[F]$

Domain of Handler $\mathrm{dom}(h) \triangleq$
$\mathrm{dom}(\mathbf{return}(x) \mapsto t) = \emptyset,$
$\mathrm{dom}(h; \mathbf{op}(x, k) \mapsto t) = \mathrm{dom}(h) \cup \{\mathbf{op}\}$

Handled Effects $\mathrm{handled}(E) \triangleq$
$\mathrm{handled}([-]) = \emptyset,$
$\mathrm{handled}(E[\mathbf{do}\ x \leftarrow [-]\ \mathbf{in}\ t_2]) = \mathrm{handled}(E),$
$\mathrm{handled}(E[\mathbf{handle}\ [-]\ \mathbf{with}\ \{h\}]) = \mathrm{handled}(E) \cup \mathrm{dom}(h),$
$\mathrm{handled}(E[\mathbf{dlet}(\alpha_R, [-])]) = \mathrm{handled}(E),$
$\mathrm{handled}(E[\mathbf{tls}([-])]) = \mathrm{handled}(E)$

Reduction Rules

*Mechanisms related to muting and unmuting are **highlighted***

(Ast-Gen)
$\langle \mathbf{mkvar}\ R; E; U; M; I\rangle \rightarrow \langle \mathbf{return}\ \alpha_R; E; U \cup \{\alpha\}; M; I\rangle$
where $\alpha = next(U), next(U) \notin U$, *next deterministic*

(Sec-Tls)
$\langle \mathbf{tls}(\mathbf{return}\ n); E; U; M; I\rangle \rightarrow \langle \mathbf{return}\ n; E; U; \emptyset; \top\rangle$

(Sec-Chs)
$\langle \mathbf{check}\ n; E; U; M; I\rangle \rightarrow \langle \mathbf{return}\ n; E; U; M; I\rangle$
if $FV^0(n) \subseteq \pi_{\mathsf{Var}}(E)$

(Sec-Cms)
$\langle \mathbf{check}_{\mathsf{M}}\ n; E; U; M; I\rangle \rightarrow \langle \mathbf{return}\ n; E; U; M; I\rangle$
if $FV^0(n) \setminus M \subseteq \pi_{\mathsf{Var}}(E)$

(Sec-Chf)
$\langle \mathbf{check}\ n; E; U; M; I\rangle \rightarrow \langle \mathbf{err}; E; U; M; I\rangle$
if $FV^0(n) \not\subseteq \pi_{\mathsf{Var}}(E)$

(Sec-Cmf)
$\langle \mathbf{check}_{\mathsf{M}}\ n; E; U; M; I\rangle \rightarrow \langle \mathbf{err}; E; U; M; I\rangle$
if $FV^0(n) \setminus M \not\subseteq \pi_{\mathsf{Var}}(E)$

(Sec-Dlt)
$\langle \mathbf{dlet}(\alpha_R, \mathbf{return}\ n); E; U; M; I\rangle \rightarrow \langle \mathbf{return}\ n; E; U; M'; I'\rangle$
if $len(E) > I$ *then* $M' = M, I' = I$, *else* $M' = \emptyset, I' = \top$

(Eff-Op)
$\langle \mathbf{op}(v); E_1[\mathbf{handle}\ E_2\ \mathbf{with}\ \{h\}]; U; M; I\rangle \rightarrow \langle c[v/x, \mathrm{cont}/k]; E_1; U; M \cup \pi_{\mathsf{Var}}(E_2); I'\rangle$
where $cont = \kappa x.\ \mathbf{handle}\ E_2[\mathbf{return}\ x]\ \mathbf{with}\ \{h\}$ *and* $\mathbf{op}(x, k) \mapsto c \in h$ *and* $\mathbf{op} \notin handled(E_2)$ *and* $I' = min(len(E_1), I)$

Fig. 5. $\lambda_{\mathsf{AST(op)}}$: syntax, types, and operational semantics.

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

Selected Rules

(FPARAM) $\dfrac{}{\Gamma \vdash \alpha_R : \mathsf{FParam}(R)}$

(VAR-AST) $\dfrac{\Gamma \vdash n : \mathsf{FParam}(R)}{\Gamma \vdash \mathtt{Var}(n) : \mathsf{AST}(R)}$

(MKVAR) $\dfrac{}{\Gamma \vdash \mathbf{mkvar}\ R : \mathsf{FParam}(R) \,!\, \Delta}$

(ERR) $\dfrac{}{\Gamma \vdash \mathbf{err} : T \,!\, \Delta}$

(LAMBDA-AST) $\dfrac{\Gamma \vdash n_1 : \mathsf{FParam}(Q) \quad \Gamma \vdash n_2 : \mathsf{AST}(R \,!\, \xi)}{\Gamma \vdash \mathtt{Lam}(n_1, n_2) : \mathsf{AST}(Q \xrightarrow{\xi} R)}$

(TLS) $\dfrac{\Gamma \vdash t : T \,!\, \Delta}{\Gamma \vdash \mathbf{tls}(t) : T \,!\, \Delta}$

(DLET) $\dfrac{\Gamma \vdash n : \mathsf{FParam}(R) \quad \Gamma \vdash t : T \,!\, \Delta}{\Gamma \vdash \mathbf{dlet}(n, t) : T \,!\, \Delta}$

(CHECK) $\dfrac{\Gamma \vdash n : T \quad T \text{ of AST type}}{\Gamma \vdash \mathbf{check}\, n : T \,!\, \Delta}$

Fig. 6. Selected $\lambda_{\mathsf{AST(op)}}$ typing rules

In the AST-GEN rule, U ensures freshness by recording previously generated names. To ensure determinacy of the semantics, fresh names are chosen by some (unspecified) deterministic process.

The **check** primitive acts like a guarded **return**. For some arbitrary normal form n of AST type, either all the free variables of n are properly scoped, so **check** n reduces to **return** n (SEC-CHS), or some free variables of n are not properly scoped, so **check** n reduces to **err** (SEC-CHF). Following Kiselyov [2024a], **dlet**s declare that variables are properly scoped, by placing a frame of the form $\mathbf{dlet}(\alpha_R, [-])$ on the evaluation context E. The notation $\pi_{\mathtt{Var}}(E)$ filters out the variables declared in this manner from E. For example, $\pi_{\mathtt{Var}}(\mathbf{dlet}(\alpha_R, \mathbf{do}\ x \leftarrow [-]\ \mathbf{in}\ t)) = \{\mathtt{Var}(\alpha_R)\}$. Given a term $E[t]$, $\mathtt{Var}(\alpha_R)$ in t is "declared safe" in E if $\mathtt{Var}(\alpha_R) \in \pi_{\mathtt{Var}}(E)$ (Definition 3.2).

DEFINITION 3.2 (DECLARED SAFE). *Given a term $E[t]$, $\mathtt{Var}(\alpha_R)$ in t is* **declared safe** *in E if $\mathtt{Var}(\alpha_R) \in \pi_{\mathtt{Var}}(E)$*

Given a normal form n in some evaluation context E, where n is an AST, n is properly scoped in E (that is, **check** n succeeds) if and only if the free Vars of n, written $\mathsf{FV}^0(n)$, have all been declared safe in E, i.e. $\mathsf{FV}^0(n) \subseteq \pi_{\mathtt{Var}}(E)$. As $\lambda_{\mathsf{AST(op)}}$ is an elaboration target for $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$, it is up to the elaboration to use **dlet** and **check** appropriately.

The $\mathbf{check}_{\mathsf{M}}$ construct is a variant of **check**. As §4.4 explains, $\mathbf{check}_{\mathsf{M}}$ additionally ignores some *muted* variables, treating them as properly scoped ($\mathbf{check}_{\mathsf{M}}\ n$ succeeds if $\mathsf{FV}^0(n) \setminus M \subseteq \pi_{\mathtt{Var}}(E)$).

SEC-TLS, SEC-DLT, and EFF-OP mute or unmute variables. §4.4 explains muting and unmuting. Ignoring muting and unmuting, SEC-TLS and SEC-DLT silently remove a $\mathbf{tls}([-])$ and $\mathbf{dlet}(\alpha_R, [-])$ frame respectively, and EFF-OP gives handlers the expected, standard behaviour.

3.2.2 Type System. $\lambda_{\mathsf{AST(op)}}$ types are mostly standard. The key additions are an FParam type for formal parameters and an AST type for abstract syntax trees (Figure 5).

The $\lambda_{\mathsf{AST(op)}}$ typing rules (Figure 6) are extremely straightforward. Under the typing rules, a well-typed AST can be *ill-scoped*; for example, $\cdot \vdash \mathtt{Var}(\alpha_R) : \mathsf{AST}(R)$ is a valid typing judgement. Scope extrusion checks are effectively invisible to the type system. The only complex case is **err**, which can be assigned any type in any context, similarly to **abort** [Scherer 2017].

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

DEFINITION 3.3 (WELL-TYPED CLOSED TERM). *A closed term t is well-typed if $\cdot \vdash t : T \,!\, \emptyset$*

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 §4, which do insert checks, extend this elaboration.

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

Type Elaboration

Selected Rules

$$\begin{array}{lcl@{\qquad}lcl}
[\![T^0]\!] & = & \mathsf{AST}(\mathsf{erase}(T^0)) & [\![\mathbb{N}^{-1}]\!] & = & \mathbb{N} \\
[\![T^0 \,!\, \xi]\!] & = & \mathsf{AST}(\mathsf{erase}(T^0 \,!\, \xi)) & [\![(S^{-1} \xrightarrow{\Delta} T^{-1})^{-1}]\!] & = & [\![S^{-1}]\!] \overset{[\![\Delta]\!]}{\Rightarrow} [\![T^{-1}]\!] \\
[\![T^0 \,!\, \Delta]\!] & = & \mathsf{AST}(\mathsf{erase}(T^0)) \,!\, [\![\Delta]\!] & [\![(S^{-1} \xrightarrow{\Delta} T^{-1})^{-1}]\!] & = & [\![S^{-1}]\!] \overset{[\![\Delta]\!]}{\Rightarrow} [\![T^{-1}]\!] \\
[\![T^0 \,!\, \Delta; \xi]\!] & = & \mathsf{AST}(\mathsf{erase}(T^0 \,!\, \xi)) \,!\, [\![\Delta]\!] & [\![\mathsf{Code}(T^0 \,!\, \xi)^{-1}]\!] & = & \mathsf{AST}(\mathsf{erase}(T^0 \,!\, \xi))
\end{array}$$

Context Entry Elaboration

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

Term Elaboration

Selected Rules (AST)

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

Selected Rules (Quote/Splice)

$$[\![\$e]\!]_{\mathbf{q}} = [\![e]\!]_{\mathbf{s}} \qquad [\![\$e]\!]_{\mathbf{c}} = \texttt{tls}([\![e]\!]_{\mathbf{s}}) \qquad [\![\langle\langle e \rangle\rangle]\!]_{\mathbf{s}} = [\![e]\!]_{\mathbf{q}}$$

Fig. 7. Selected elaboration rules from $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ to $\lambda_{\mathsf{AST}(\mathsf{op})}$.

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 sets, types, contexts, and terms.

3.3.1 Elaborating Effect Sets and Types. Elaboration of effect sets is the identity. To define the elaboration of types (Figure 7), it is convenient to refer to a helper function, erase. Given a level 0 type, erase *erases* all the level annotations (and elaborates effect sets), e.g. $\mathsf{erase}((S^0 \xrightarrow{\xi} T^0)^0) = S \overset{[\![\xi]\!]}{\Rightarrow} T$. 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.

3.3.2 Elaborating Contexts. Elaboration of contexts is subtle (Figure 7). Level 0 types in the context elaborate 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. To see why level 0 types elaborate 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.

3.3.3 Elaborating Terms. Elaboration of terms (Figure 7) 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**. At a high level, in **c** and **q**-mode, one builds ASTs. To ensure formal parameters are appropriately renamed, the elaboration must use **`mkvar`**.

Elaboration does not differ significantly between **c** and **q**-modes, except in the rule for splice, where **`tls`** is inserted in **c**-mode, but not in **q**-mode. The **c** and **q**-modes become important when building scope extrusion checks. Elaboration in **s**-mode is effectively the identity.

3.3.4 Elaborating Typing Judgements. Elaboration of typing judgements can now be defined compositionally. For example, the typing judgement for lambdas in **c**-mode is elaborated by applying the elaboration component-wise:

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

Letting $Q = \mathsf{erase}(S)$, $R = \mathsf{erase}(T)$, and $[\![e]\!]_{\mathbf{c}} = t$, and applying the elaboration functions defined above, we obtain Typing Derivation 1, which, assuming that the premise is a valid typing derivation, corresponds to a valid $\lambda_{\mathsf{AST(op)}}$ typing derivation.

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

Typing Derivation 1. The elaborated derivation of $\Gamma \vdash_{\mathbf{c}} \lambda x.e : S \xrightarrow{\xi} T$

3.4 Metatheory

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

Theorem 3.1 (Elaboration Preservation). *If* $\Gamma \vdash_\star e : \tau$ *then* $[\![\Gamma]\!] \vdash [\![e]\!]_\star : [\![\tau]\!]$, *where* $\star = \mathbf{c} \mid \mathbf{q} \mid \mathbf{s}$ *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 1 in §3.3.4.

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

Theorem 3.2 (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* $op \in \Delta$, *and* $\mathsf{op} \notin$ *handled*(E)
(3) t *is of the form* **err**
(4) $\exists\, t', E', U', M', I'$ *such that* $\langle t; E; U; M; I\rangle \longrightarrow \langle t'; E'; U'; M'; I'\rangle$

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

The proof of progress is by induction over the typing derivation. Most cases are standard, and have been shown by Bauer and Pretnar [2014]. The proof need only consider the typing rules for AST construction and scope extrusion checking, all of which are straightforward.

Theorem 3.3 (Reduction Preservation). *If* $\cdot \vdash E[t] : T \,!\, \Delta$ *and* $\langle t; E; U; M; I\rangle \longrightarrow \langle t'; E'; U'; M'; I'\rangle$ *then* $\cdot \vdash E'[t'] : T \,!\, \Delta$

The proof is by induction over the operational semantics. Once again, one need only consider the rules for AST construction and scope extrusion checking, which are simple.

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

Corollary 3.4 (Type Safety). *If* $\cdot \vdash_{\mathbf{c}} e : T^0 \,!\, \emptyset; \emptyset$ *then either*

(1) $\langle [\![e]\!]_{\mathbf{c}}; [-]; \emptyset; \emptyset; \top\rangle \longrightarrow^\omega$,
(2) $\langle [\![e]\!]_{\mathbf{c}}; [-]; \emptyset; \emptyset; \top\rangle \longrightarrow^* \langle \mathbf{err}; E; U; M; I\rangle$ *for some* E, U, M, I, *or*
(3) $\langle [\![e]\!]_{\mathbf{c}}; [-]; \emptyset; \emptyset; \top\rangle \longrightarrow^* \langle \mathbf{return}\ n; [-]; U; M; I\rangle$ *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, as would a semantics which never reports scope extrusion. Due to the potential presence of scope extrusion, the third case of Corollary 3.4 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 (Quote-Splice Duality). *Under a top-level splice, quotation and splice are duals:*

$$\$\langle\langle e\rangle\rangle =_{\mathbf{q}} e \qquad\qquad \langle\langle \$e\rangle\rangle =_{\mathbf{s}} e$$

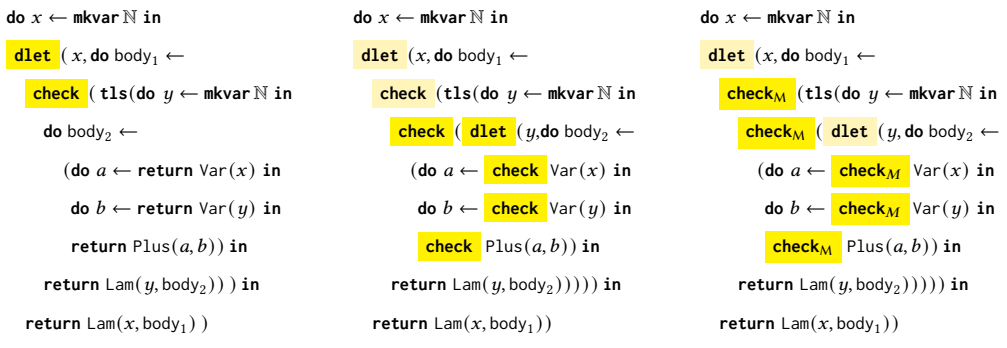


Fig. 8. Elaboration of $\lambda x : \mathbb{N}.\ \$\langle\langle \lambda y : \mathbb{N}.\ x + y \rangle\rangle$ under different checks

where $=_\star$ means "elaborates to contextually equivalent $\lambda_{\mathsf{AST(op)}}$ programs in $\star$ mode". Parameterising by the mode is necessary, since it affects the result of elaboration. It is possible to prove something stronger: they elaborate to the same syntactic $\lambda_{\mathsf{AST(op)}}$ program (contextual equivalence follows from reflexivity). The proof of Theorem 3.4 is by inspection of the definition of elaboration, where:

$$[\![\$\langle\langle e \rangle\rangle]\!]_{\mathbf{q}} = t \iff [\![e]\!]_{\mathbf{q}} = t \qquad\qquad [\![\langle\langle \$e \rangle\rangle]\!]_{\mathbf{s}} = t \iff [\![e]\!]_{\mathbf{s}} = t$$

4 Dynamic Scope Extrusion Checks

This section uses $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ to formulate precise definitions of scope extrusion (including existing approaches [Isoda et al. 2024; Kiselyov 2014]), and properties of scope extrusion checks.

4.1 Properties of Dynamic Scope Extrusion Checks

Since checks are defined as term elaborations, we use $[\![-]\!]^{\mathbf{Check}}$ to indicate an arbitrary check. We refer to the term elaboration in §3.3 as naïve elaboration.

Given a definition of scope extrusion as a predicate Φ on configurations, a check is **correct** if, whenever the naïve elaboration of a well-typed $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ expression e reduces to a configuration exhibiting scope extrusion ($\Phi(\langle t; E; U; M; I \rangle)$), the elaboration of e with the check reduces to **err**. The **permissiveness** of a scope extrusion check refers to the set of well-typed $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ expressions whose elaborations do not reduce to **err**, even if they exhibit scope extrusion.

Definition 4.1 (Correctness of a Dynamic Scope Extrusion Check). *Given a predicate on configurations Φ, a dynamic scope extrusion check $[\![-]\!]^{\mathbf{Check}}$ is* **correct** *with respect to Φ if for all closed, well-typed $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ expressions e, $\langle [\![e]\!]; [-]; \emptyset; \emptyset; \top \rangle \to^* \langle t; E; U; M; I \rangle \wedge \Phi(\langle t; E; U; M; I \rangle) \implies \langle [\![e]\!]^{\mathbf{Check}}; [-]; \emptyset; \emptyset; \top \rangle \to^* \langle \mathbf{err}; E'; U'; M'; I' \rangle$ for some E', U', M', I'.*

Definition 4.2 (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 $\{e \in \mathit{WellTyped} \mid \langle [\![e]\!]^{\mathbf{Check}}; [-]; \emptyset; \emptyset; \top \rangle \not\to^* \langle \mathbf{err}; E; U; M; I \rangle\}$*

4.2 Lazy Check

A $\lambda_{\mathsf{AST(op)}}$ configuration exhibits **lazy scope extrusion** if it is the *result* of compile-time execution and is improperly scoped. This formalises the definition by Kiselyov [2014].

Definition 4.3 (Lazy Scope Extrusion). *A $\lambda_{\mathsf{AST}(\mathsf{op})}$ configuration of the form $\langle t; E; U; M; I\rangle$ exhibits* **lazy scope extrusion** *if $t = \mathtt{return}\ n$ for some n of AST type, $E = E'[\mathtt{tls}([-])]$ for some E', and $FV^0(n) \not\subseteq \pi_{\mathsf{Var}}(E)$.*

The lazy check, $[\![-]\!]^{\mathbf{Lazy}}$, augments the naïve elaboration in two ways (Figure 8a). First, **check**s are performed after top-level splices: $([\![\$e]\!]^{\mathbf{Lazy}}_{\mathbf{c}} \triangleq \mathtt{check}\,(\mathtt{tls}([\![e]\!]^{\mathbf{Lazy}}_{\mathbf{s}})))$. Second, **dlet**s are inserted to ensure variables bound outside top-level splices (in **c**-mode) are declared safe (Definition 3.2) in the context surrounding the top-level splice. Elaboration of formal parameters in **c**-mode (but not **q**-mode) should insert **dlet**s:

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

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 check detects scope extrusion if, and only if, naïve elaboration would exhibit lazy scope extrusion after reduction.

Theorem 4.1 (Correctness and Permissiveness of the Lazy Check). *For all closed, well-typed $\lambda_{\langle\langle \mathsf{op}\rangle\rangle}$ programs e, $\langle [\![e]\!]^{\mathbf{Lazy}}; [-]; \emptyset; \emptyset; \top\rangle \rightarrow^* \langle \mathtt{err}; E; U; M; I\rangle \iff$ For some E', U', M', I', $\langle [\![e]\!]; [-]; \emptyset; \emptyset; \top\rangle \rightarrow^* \langle \mathtt{return}\ n; E'; U'; M'; I'\rangle$, and $\langle \mathtt{return}\ n; E'; U'; M'; I'\rangle$ exhibits lazy scope extrusion*

The lazy check thus characterises the set of $\lambda_{\langle\langle \mathsf{op}\rangle\rangle}$ programs that it is safe to permit. This set is used to define the *expressiveness* of a check, where the lazy check is *maximally* expressive:

Definition 4.4 (Expressiveness of a Dynamic Scope Extrusion Check). *Define the set* $\mathsf{Safe} \triangleq \{e \in \mathit{WellTyped} \mid \langle [\![e]\!]^{\mathbf{Lazy}}; [-]; \emptyset; \emptyset; \top\rangle \not\rightarrow^* \langle \mathtt{err}; E; U; M; I\rangle\}$. *Then the* **expressiveness** *of a dynamic scope extrusion check is defined as* $\{e \in \mathit{Safe} \mid \langle [\![e]\!]^{\mathbf{Check}}; [-]; \emptyset; \emptyset; \top\rangle \not\rightarrow^* \langle \mathtt{err}; E; U; M; I\rangle\}$

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

Definition 4.5 (False Positives of a Dynamic Scope Extrusion Check). *The* **false positives** *of a dynamic scope extrusion check are defined as* $\{e \in \mathit{Safe} \mid \langle [\![e]\!]^{\mathbf{Check}}; [-]; \emptyset; \emptyset; \top\rangle \rightarrow^* \langle \mathtt{err}; E; U; M; I\rangle\}$

However, due again to its simplicity, the lazy check is considered unsuitable for practical use. Ofenbeck et al. [2016], who use the lazy check, report the following:

> *Bugs in our implementation … would manifest in errors such as:*
>
> ```
> forward reference extends over definition of value x1620
> [error] val x1343 = x1232(x1123, x1124, x1180, x1181,
> x1223, x1224, x1223, x1229, x1216, x1120, x1122, x1121)
> ```
>
> *…[A] large piece of code is processed before we hit this error … The root cause of bugs such as this one often proved to be very simple but heavily obfuscated in the code it manifested in.*

The lazy check is uninformative: since it waits until the end of evaluation, errors refer to the generated code rather than the generating program [Kiselyov 2014]. This obfuscation makes debugging difficult for all programs. In addition, the lazy check has to wait for the end of evaluation before reporting errors. This creates an inefficiency in debugging large staged programs, like the ones generated by Ofenbeck et al.. Additionally, Kameyama et al. [2015, §4.1] note that in some systems, the lazy check can result in unintendedly bound variables.

4.3 Eager Check

A configuration exhibits **eager scope extrusion** if it **return**s an improperly scoped AST at any point in the execution. Definition 4.6 thus generalises Definition 4.3.

Definition 4.6 (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 $t = \mathbf{return}\ n$ for some n of AST type, and $FV^0(n) \not\subseteq \pi_{\mathsf{Var}}(E)$*

It is possible to define an eager check by extending the lazy check (Figure 8b). In addition to the top-level splice **check**, and the **c**-mode **dlet**s, the eager check adds **check**s for ASTs constructed in **q**-mode, for example:

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

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 generate **dlet**s:

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

Intuitively, the eager 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 (Listing 7). Kiselyov [2014] observes that the overhead of checking on AST construction is negligible.

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

```
$(do z ← (handle ⟨⟨ λx.$( op(⟨⟨x⟩⟩) ) ⟩⟩
           with {return(u) ↦ ⟨⟨0⟩⟩; op(y, k) ↦ return y})
  in ⟨⟨$z + 1⟩⟩)
```

Listing 7. Extrusion is reported when z is used $\langle\langle \$z + 1\rangle\rangle$ in a context where $\mathsf{Var}(x_{\mathbb{N}})$ is not declared safe

The eager check models the BER MetaOCaml check described by Kiselyov [2024b]. The model can be verified by executing the BER MetaOCaml N153 translations of Listings 7 to 10 in the accompanying artifact [Lee et al. 2025].

4.3.1 Correctness of the Eager Check. The eager check is *not* correct with respect to eager scope extrusion. Evaluation may result in eager scope extrusion that is never detected. For example, the offending AST could be discarded (Listing 8).

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

```
$(handle ⟨⟨ λx.$( op(⟨⟨x⟩⟩) ) ⟩⟩
  with {return(u) ↦ ⟨⟨0⟩⟩; op(y, k) ↦ do w ← return y in ⟨⟨0⟩⟩})
```

Listing 8. The eager check does not report eager scope extrusion when the offending AST is discarded.

A notable property of the eager check is that it allows a program to recover from scope extrusion by *resuming* a continuation. In Listing 9, the program restores the captured evaluation context, which declares $\mathsf{Var}(x_{\mathbb{N}})$ safe. Only then is $\mathsf{Var}(x_{\mathbb{N}})$ used to build an AST, so the checks pass.

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

```
$(handle ⟨⟨ λx. return $( op(⟨⟨x⟩⟩) ) ⟩⟩
  with {return(u) ↦ return u; op(y, k) ↦ do u ← return y in continue k u})
```

Listing 9. The eager check does not report cases where the offending AST is used only in safe ways.

The incorrectness of the eager check (i.e. that it does not report all eager scope extrusion) arises naturally from the definitions. Kiselyov [2014] defines eager scope extrusion as the occurrence of a free variable at any point in the evaluation. The eager *check*, in contrast, is only invoked when the free variable is used, e.g. executed or used to construct larger pieces of code. The incorrectness of the eager check, however, can be desirable. Since Listings 8 and 9 are in Safe, permissiveness makes the eager check more expressive.

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

Fig. 9. The result of elaborating Listing 10 using the eager check

4.3.2 Expressiveness of the Eager Check. In the presence of first-class continuations, the eager check is *not* maximally expressive. It reports false positives, such as Listing 10.

```
$(handle ⟨⟨ λx.$( op(⟨⟨x⟩⟩) ) ⟩⟩                                        λ⟨⟨op⟩⟩
  with {return(u) ↦ return u; op(y, k) ↦ do u ← ⟨⟨$y + 0⟩⟩ in continue k u})
```

Listing 10. A false positive: a safe program that fails the eager check.

In Listing 10, the offending AST ($\texttt{Var}(x_{\mathbb{N}})$) is used in a context where $\texttt{Var}(x_{\mathbb{N}})$ is not declared safe, and thus the eager check reports an error. However, if evaluation had been allowed to proceed, the evaluation context binding $\texttt{Var}(x_{\mathbb{N}})$ and declaring it safe would have been restored, and all variables would have been properly scoped.

Comparing Listing 9, which passes the eager check, with Listing 10, which fails the check, shows that the check is unpredictable: it is difficult to characterise its expressiveness without referring to the operational semantics. Unfortunately, $\langle\langle \$e \rangle\rangle \neq_{\mathsf{s}} \langle\langle \$e + 0 \rangle\rangle$. More generally, for program fragments P and P', $P[e] =_{\mathsf{s}} P'[e] \implies \langle\langle P[\$\langle\langle e \rangle\rangle]\rangle\rangle =_{\mathsf{s}} \langle\langle P'[\$\langle\langle e \rangle\rangle]\rangle\rangle$.

The unpredictability arises from the design of the eager check. Kiselyov [2014, Footnote 10] notes that the eager check can report false positives in the presence of first class continuations, but has not observed such cases in practice. We say that the eager check is not *continuation-aware*.

4.4 Cause-for-Concern (C4C) Check

If the lazy check is too impractical, and the eager check too unpredictable, might it be possible to find a "goldilocks" solution? Such a check should allow the program in Listing 10, and be permissive in a predictable way. A configuration exhibits **inevitable scope extrusion** when it *must* cause lazy scope extrusion.

Definition 4.7 (Inevitable Scope Extrusion). *A $\lambda_{\mathsf{AST}(\mathsf{op})}$ configuration of the form $t; E; U; M; I$ exhibits* **inevitable scope extrusion** *if $\langle t; E; U; M; I\rangle \longrightarrow^* \langle t'; E'; U'; M'; I'\rangle$ and $\langle t'; E'; U'; M'; I'\rangle$ exhibits lazy scope extrusion.*

This section describes a Cause-for-Concern (C4C) check that approximates inevitable scope extrusion, though with false positives. Elaboration for the C4C check is a slight variation of elaboration for the eager check, with $\textbf{check}_{\mathsf{M}}$ replacing **check** (Figure 8c). For example,

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

To understand the C4C check, consider Figure 9, where Listing 10 is elaborated using the eager check into $\lambda_{\mathsf{AST}(\mathsf{op})}$ and simplified for readability (e.g. **check** t rather than **do** $x \leftarrow t$ **in check** x). The failing check is underlined.

The check fails because when **op** is performed, the variable $\texttt{Var}(x_{\mathbb{N}})$ is no longer declared safe in the new evaluation context. Since y is bound to $\texttt{Var}(x_{\mathbb{N}})$, checking $\texttt{Plus}(y, \texttt{Nat}(0))$ reports an error. The problem is that the continuation k can be used to bind $\texttt{Var}(x_{\mathbb{N}})$. It is not clear, when the

```
check_M(dlet(z_N, do b ←
    (do f ← body
     in do a ← return Nat(1)
     in check_M App(f, a))
  in return Lam(z_N, b)))
```

(a) Initial term

```
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(z_N, b)))
```

(b) Reduced term

Fig. 10. (a) A $\lambda_{\mathsf{AST(op)}}$ program that generates the AST of $\lambda z.\,(\lambda x.\,x + \mathtt{0})(\mathtt{1})$. (b) The result of reducing the program in (a) to the point where variables may be unmuted.

`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**$_{\mathsf{M}}$ primitive checks for scope extrusion, but allows a set of muted variables M to temporarily extrude their scope. In our example, we may mute $\mathtt{Var}(x_{\mathbb{N}})$, by adding it to M. The $\lambda_{\mathsf{AST(op)}}$ operational semantics (Figure 5) automates this process, strategically muting and unmuting variables at key points:

- 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).
- Variables are unmuted when there are no bound continuations, and thus no way to resume a continuation k that could bind $\mathtt{Var}(x_{\mathbb{N}})$. This point is identified by tracking the maximal length I of the stack E that was never captured by the handling of an effect.

Intuitively, I is a stack mark (or continuation mark), which tracks the point where effects and exceptions are indistinguishable. The C4C check acts like the lazy check before this point, and like the eager check after it. Stack marks are used in the eager check implementation [Kiselyov 2014, Appendix B], and in the semantics and implementation of languages with continuations [Flatt and Dybvig 2020; Kiselyov 2012]. However, since the eager check is not continuation-aware, stack marks play only a limited role.

As an example, the program in Figure 10a builds the AST of $\lambda z.\,(\lambda x.\,x + \mathtt{0})(\mathtt{1})$. Let body be the program in Figure 9. The surrounding context around body is identified by I: 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.

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 10a eventually reduces to the term in Figure 10b. [−] separates the evaluation context (outside) and the term (inside). At this point, the length of the stack is less than or equal to I. It is safe to unmute all muted variables. When there are no muted variables, **check**$_{\mathsf{M}}$ and **check** have the same behaviour.

However, altering the semantics in such a manner means that any transition could unmute variables. To keep the semantics standard, and to more closely model the implementation of the check, we associate the act of unmuting with **dlet** and **tls**. A transition from **dlet** conditionally unmutes variables (Sec-Dlt, Figure 5). In Figure 10b, the transition from **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, a transition from **tls** *unconditionally* unmutes variables, since the evaluation context beyond **tls** must be inert, and thus can never be captured by a handler (Sec-Tls).

As a $\textbf{check}_{\mathsf{M}}$ can never fail where a **check** succeeds, the C4C check is at least as permissive as the eager check.

4.4.1 Correctness of the C4C check. The C4C check is correct with respect to inevitable scope extrusion. The proof is simple: either one of the non-top-level splice $\textbf{check}_{\mathsf{M}}$s reports an error, or none do. The latter case degenerates to the lazy check, where the top-level splice $\textbf{check}_{\mathsf{M}}$ must report an error.

Theorem 4.2 (Correctness of the C4C Check). *Given a closed, well-typed $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ expression e, if $\langle [\![e]\!]; [-]; \emptyset; \emptyset; \top \rangle$ exhibits inevitable scope extrusion then there exists E, U, M, I such that $\langle [\![e]\!]^{\boldsymbol{BE}}; [-]; \emptyset; \emptyset; \top \rangle \longrightarrow^* \langle \textbf{err}; E; U; M; I \rangle$*

4.4.2 Expressiveness of the C4C Check. The C4C check is not maximally expressive. In particular, it does not allow the program in Listing 11.

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

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

Listing 11. The C4C check reports false positives.

Listing 11 attempts to build the AST $\lambda x.\textbf{return}\ y$, 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 11 eventually reduces to a configuration:

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

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

A Cause-for-Concern property characterises the expressiveness of the C4C check[1]. The property is defined informally as follows: assume the 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 $\textbf{check}_{\mathsf{M}}$s are erased (turned into **return**s). In this counter-factual execution, all ASTs n' that are constructed from n have at least one variable that is not declared safe in its evaluation context. Consequently, in Listing 11, the only way to safely use $\lambda x.\textbf{return}\ y$ is to throw it away.

Theorem 4.3 (Cause-for-Concern Property). *Assuming a closed, well-typed $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ expression e, if $\exists\, E, U, M, I$ such that $\langle [\![e]\!]^{\boldsymbol{BE}}; [-]; \emptyset; \emptyset; \top \rangle \longrightarrow^* \langle \textbf{check}_M\ n; E; U; M; I \rangle$, and $\langle \textbf{check}_M\ n; E; U; M; I \rangle \longrightarrow \langle \textbf{err}; E; U; M; I \rangle$, then, assuming $\langle \textbf{return}\ n; \mathit{erase\text{-}checks}(E); U; M; I \rangle \longrightarrow^* \langle \textbf{return}\ n'; E'; U'; M'; I' \rangle$, and n a subtree of n', it must be that $\mathsf{FV}^0(n') \not\subseteq \pi_{\mathtt{Var}}(E')$.*

The proof of Theorem 4.3 is by contradiction. Informally, if $\mathsf{FV}^0(n') \subseteq \pi_{\mathtt{Var}}(E')$, then all the variables in n must be declared safe. This implies that when the initial $\textbf{check}_{\mathsf{M}}\ n$ failed, there was a continuation on the stack which can declare the variables in n safe. But then M cannot be empty, so the check would not have failed.

The expressiveness of the eager check cannot be characterised by the Cause-for-Concern property, with Listing 10 being a counter-example. Hence, the C4C check is **more** expressive, and more **predictably** expressive, than the eager check.

Like the eager check, the expressiveness of the C4C check can be empirically verified by executing MacoCaml translations of Listings 7 to 11 in the accompanying artifact [Lee et al. 2025].

[1] and gives it its name, which, unlike the lazy and eager check, describes its user-facing behaviour, not its operation

Table 3. Correctness comparison

	Listings				
	8	9	10	11	12
Lazy (§4.2)	Y	Y	Y	Y	Y
Eager (§4.3)	N	N	Y	Y	Y
C4C (§4.4)	Y	Y	Y	Y	Y
Ref. Env. Classifiers (§5)	Y	Y	Y	Y	Y

Table 4. Expressiveness comparison

	Listings				
	8	9	10	11	12
Lazy (§4.2)	Y	Y	Y	Y	Y
Eager (§4.3)	Y	Y	N	N	Y
C4C (§4.4)	Y	Y	Y	N	Y
Ref. Env. Classifiers (§5)	N	N	N	N	Y

4.5 Evaluation of $\lambda_{\langle\langle \mathbf{op} \rangle\rangle}$

We have demonstrated that $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ is an appropriate language for encoding and evaluating scope extrusion checks. Tables 3 and 4 summarize the correctness and expressiveness of the checks (with refined environment classifiers and Listing 12 discussed in the next section). The accompanying artifact [Lee et al. 2025] provides translations of Listings 7 to 11 in both BER MetaOCaml N153 and MacoCaml. The first three rows of Tables 3 and 4 can be verified empirically by executing these translations. Unifying these checks under $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ facilitated comparative evaluation with reference to the same set of programs. Moreover, formalising scope extrusion in $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ aided development of the novel C4C check, which finds a sweet spot between the eager and lazy checks.

It is worth re-iterating that the focus of $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ is on scope extrusion. There are additional interesting questions related to bindings in generated code that this paper does not consider. In particular, $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ does not prevent shadowing: it is possible in $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ to generate programs with multiple binders that use the same formal parameter. For example, using multi-shot continuations, it is possible in $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ to generate the code $\mathtt{Lam}(x, \mathtt{Lam}(x, \mathtt{body}))$, since the binder $\mathtt{Lam}(x, -)$ can be captured in a continuation that is re-instated in a nested manner. None of the three dynamic checks is able to detect every instance of scope shadowing. Restricting $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ to permit only one-shot continuations, as in systems like `MetaOCaml` and `MacoCaml`, would prevent shadowing, but in practice multi-shot continuations are useful in multi-staged programming, e.g. for case-insertion [Yallop 2017, §4.4], and do not compromise type safety.

5 Extension: Refined Environment Classifiers

This section presents $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}{}^{\gamma}$, an extension to $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ with *refined environment classifiers* [Kiselyov et al. 2016] to statically prevent scope extrusion, following Isoda et al. [2024]. §4 illustrates the use of $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ to compare dynamic scope extrusion checks, and $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}{}^{\gamma}$ shows how to extend the framework to describe and evaluate static prevention techniques, too.

5.1 The Calculus

Figure 12 presents the types and selected typing rules of $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}{}^{\gamma}$. The calculus shares its syntax with $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$, extending it with a simplified[2] version of Isoda et al.'s type system.

Intuitively, a *classifier* represents a scope that permits a set of free variables. An AST is considered well-scoped at a given scope if it is well-typed and all its free variables are permitted by the scope.

As an example, consider Figure 11, where there are two classifiers: γ_α is the scope that permits only $\mathtt{Var}(\alpha)$, and $\gamma_\perp$ the scope that permits no variables (the "top-level"). To capture the nesting of scopes, classifiers are related by a partial order $\gamma \sqsubseteq \gamma'$, with γ the outer scope, and γ' the inner scope; in this case, we have $\gamma_\perp \sqsubseteq \gamma_\alpha$.

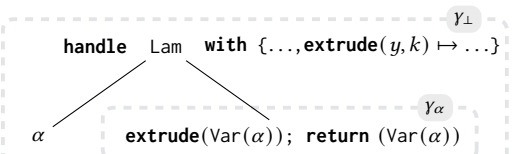


Fig. 11. Refined environment classifiers

[2] Isoda et al.'s typing rules for handlers and continuations are polymorphic over the classifier, to allow for let-insertion.

Classifiers prevent scope extrusion by checking that created ASTs are well-scoped, and moreover, that manipulating ASTs preserves well-scopedness. Specifically, they prevent variables being lifted into scopes where they are not permitted. In Figure 11, the **extrude** effect attempts to lift $\texttt{Var}(\alpha)$ to a handler in the $\gamma_\perp$ scope (where $\texttt{Var}(\alpha)$ is not permitted). Figure 11 cannot be typed, regardless of the body of the handler.

Syntax. We annotate level -1 Code types with a classifier γ. For level 0 types, classifiers are associated in the typing contexts and in the typing judgement. We define *extended* $\lambda_{\langle\langle \mathsf{op} \rangle\rangle^\gamma}$ types, a notion useful for defining the logical relation in §5.2.

Definition 5.1 (Extended $\lambda_{\langle\langle \mathsf{op} \rangle\rangle^\gamma}$ type). *An* extended source type *is either:*

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

The typing context Γ maps terms to their types, and tracks their environment classifiers for level 0 types. Additionally, it tracks classifiers γ and their partial ordering $\gamma \sqsubseteq \gamma'$. A level -1 type T^{-1} is well-formed under a context Γ, written $\Gamma \vdash T^{-1}$, if all its classifiers are in Γ. A context is well-formed if it contains the *least* classifier $\gamma_\perp$, and if all its types are well-formed. We will assume all contexts are well-formed.

Typing. Most typing rules are straightforwardly adapted, with key rules listed in Figure 12. The **c**|**q**-Var rule says that a variable with classifier γ is well-typed under the classifier γ. Of particular interest is the **c**|**q**-Lambda rule. As classifiers formalise the notion of scope, this rule introduces a new scope, represented as a fresh classifier $\gamma' \notin \Gamma$ and associates the variable x with γ'. Moreover, since γ' is created within the scope of γ, we have $\gamma \sqsubseteq \gamma'$.

The **c**|**q**-Sub-Expr and **s**-Sub rules formalise the nesting of scopes: to show a term is well-scoped in some nested scope γ, it suffices to show that it is well-scoped in any of its parents γ', with $\gamma' \sqsubseteq \gamma$.

Following Isoda et al. [2024], operations (**s**-Op), continuations (**s**-Continue), and handlers (**s**-Handle) are restricted to Code types, while it can easily generalise to non-Code types. These rules work in concert to prevent scope extrusion. Specifically, the **handle** construct operates on Code types, and acts at a scope γ. As a result, handlers cannot change the scopes in which the result of computation is permitted, e.g. changing the type from $\text{Code}(\mathbb{N})^\gamma$ to a different classifier $\text{Code}(\mathbb{N})^{\gamma'}$. Notably, each handler clause (and thus handled effect) inherits this scope γ. As a result, the values passed to handled effects, should they be code types, must be tagged with a classifier that may be substituted for γ. Since binders introduce new classifiers γ', where γ' cannot be substituted for γ (since $\gamma' \not\sqsubseteq \gamma$), examples that result in scope extrusion do not type check.

Lastly, we note that since contexts must contain the least classifier $\gamma_\perp$, an expression e is a closed, well-typed expression if $\gamma_\perp \vdash^{\gamma_\perp}_{\mathbf{c}} e : T^0 \,!\, \emptyset; \emptyset$.

Elaboration. 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, where formal parameters are annotated with classifiers (e.g. α^γ_R). Classifiers show up only in the formal parameters, and are invisible to the types. Since elaboration does not require any dynamic scope extrusion checking machinery, $\lambda_{\mathsf{AST}(\mathsf{op})^\gamma}$ does not have **check**, $\textbf{check}_\mathsf{M}$, **dlet**, **tls**, and **err**. Consequently, $\lambda_{\mathsf{AST}(\mathsf{op})^\gamma}$ configurations are of the form $\langle t; E; U \rangle$.

Elaboration is similar to §3.3, except that elaboration of types erases classifiers, elaboration of context entries erases proof-theoretic terms, and elaboration of terms assumes binders have been annotated with an extended source type, and does *not* erase classifiers. Finally, elaboration of top-level splice does not insert **tls**.

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

Typing contexts

$\Gamma ::= \cdot \mid \Gamma, (x : T^0)^\gamma \mid \Gamma, x : T^{-1} \mid \Gamma, \gamma \mid \Gamma, \gamma \sqsubseteq \gamma'$

Types

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

Typing Rules

Selected Rules

(c|q-Var)
$$\frac{(x : T^0)^\gamma \in \Gamma}{\Gamma \vdash^\gamma_{\mathsf{c}|\mathsf{q}} x : T^0 \,!\, \Delta}$$

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

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

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

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

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

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

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

(s-Sub)
$$\frac{\Gamma \vDash \gamma' \sqsubseteq \gamma \qquad \Gamma \vdash_{\mathsf{s}} e : \mathsf{Code}(T \,!\, \xi)^{\gamma'} \,!\, \Delta}{\Gamma \vdash_{\mathsf{s}} e : \mathsf{Code}(T \,!\, \xi)^\gamma \,!\, \Delta}$$

Fig. 12. $\lambda_{\langle\langle \mathsf{op} \rangle\rangle^\gamma}$: types and selected typing rules.

Weakening. We prove a weakening lemma, which is useful for our later proof of correctness. As types are stratified into two levels, and into value, computation, and handler types, there are various sub-lemmas. As an example, we present weakening for level 0 computations:

Lemma 5.2 (Weakening for Level 0 Computations). *If* $\Gamma \vdash^\gamma_{\mathsf{c}|\mathsf{q}} e : T^0 \,!\, \Delta$ *then*

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

5.2 Correctness of Refined Environment Classifiers

In this section, we prove the correctness of refined environment classifiers: every well-typed $\lambda_{\langle\langle \mathsf{op} \rangle\rangle^\gamma}$ term produces a well-scoped AST on termination. However, with an elaboration-based semantics, directly reasoning about $\lambda_{\langle\langle \mathsf{op} \rangle\rangle^\gamma}$ is challenging. As a result, we employ Tait-style logical relations [Tait 1967] to demonstrate that typing guarantees are preserved by elaboration [Benton and Hur 2009], thereby establishing correctness of refined environment classifiers.

Figure 13 presents the logical relation, Scoped, defined on core language ($\lambda_{\mathsf{AST}(\mathsf{op})^\gamma}$) terms. The relation is indexed by a context of proof-theoretic terms Θ and an *extended* $\lambda_{\langle\langle \mathsf{op} \rangle\rangle^\gamma}$ type (Definition 5.1). Given a context Γ, $\pi_\gamma(\Gamma)$ projects out only the proof theoretic terms. For example, given $\Gamma = \gamma_\bot, \gamma_1, \gamma_\bot \sqsubseteq \gamma_1, (x : \mathbb{N}^0)^{\gamma_1}, \gamma_2, \gamma_1 \sqsubseteq \gamma_2, y : (\mathsf{Code}(\mathbb{N}^0 \,!\, \emptyset)^{\gamma_2})^{-1}$, 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.

The two key 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

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

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

Context of Proof Theoretic Terms

$\Theta := \gamma_\perp \mid \Theta, \gamma \mid \Theta, \gamma' \sqsubseteq \gamma$

Normal Forms

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

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

$n \in \mathsf{Scoped}_{\Theta,\tau} \triangleq \cdot \vdash n \in [\![\tau]\!]$ and $\Theta \vdash \mathsf{FV}^0(n) \subseteq \mathsf{permitted}(\gamma)$

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

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

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

Handlers

$h \in \mathsf{Scoped}_{\Theta,(S^{-1}\,!\,\Delta_1 \Longrightarrow T^{-1}\,!\,\Delta_2)^{-1}} \triangleq$ if $h = \mathbf{return}(x) \mapsto t_{\mathrm{ret}}$
$\forall n' \in \mathsf{Scoped}_{\Theta,S^{-1}}, t_{\mathrm{ret}}[n'/x] \in \mathsf{Scoped}_{\Theta,T^{-1}\,!\,\Delta_2}$

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

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 $T^{-1} \,!\, \Delta$
Given a compile-time computation type $\tau \,!\, \Delta$, let τ refer to the corresponding value type. e.g. if $\tau \,!\, \Delta = T^0 \,!\, \Delta; \xi(\gamma)$, then $\tau = T^0 \,!\, \xi(\gamma)$

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

(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 $\langle t; [-]; U\rangle \to^* \langle \mathbf{op}(n); E; U'\rangle$ where $\mathbf{op} \notin \mathsf{handled}(E)$, U' consistent with $E[\mathbf{op}(n)]$, and
   (a) $\mathrm{op} : A^{-1} \to B^{-1}$,
   (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.

Fig. 13. The definition of the Scoped logical relation

AST(erase(T^0)), and the free variables of n need to be permitted within the scope represented by γ. Permissibility assumes some known partial order on classifiers, e.g. $\gamma' \sqsubseteq \gamma$, which is carried by the index Θ. $\mathsf{Scoped}_{\Theta,\tau\,!\,\Delta}$ is defined as a least fixed point, following similar definitions by Plotkin and Xie [2025] and Kuchta [2023], giving rise to the principle of Scoped-Induction:

Induction Principle 5.3 (Scoped-Induction). *For a property Φ on closed terms of type $[\![\tau \,!\, \Delta]\!]$,*
(1) if $\langle t; [-]; U\rangle \to^* \langle \mathbf{return}\ n; [-]; U'\rangle$ implies $\Phi(t)$, and
(2) if $\langle t; [-]; U\rangle \to^* \langle \mathbf{op}(n); E; U'\rangle$ where $\mathbf{op} \notin$ handled(E), $\mathrm{op} : A^{-1} \to B^{-1}$, $n \in \mathsf{Scoped}_{\Theta,A^{-1}}$, 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 [Kuchta 2023] 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 (clause 2 in Definition 5.5).

Lemma 5.4 (Closure under Anti-Reduction). *Assume* $\langle t; E; U\rangle \longrightarrow^* \langle t'; E'; U'\rangle$. *Then* $E'[t'] \in \mathit{Scoped}_{\Theta,\tau\,!\,\Delta} \implies E[t] \in \mathit{Scoped}_{\Theta,\tau\,!\,\Delta}$

Definition 5.5 (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) $() \vDash \gamma_\perp$
(2) If $\rho \vDash \Gamma$, *then for arbitrary* $\gamma \in \Gamma$, $n \in \mathit{Scoped}_{\Theta,\overline{T^0(\gamma)}}$, $(\rho, n/x) \vDash \Gamma, (x : T^0)^\gamma$
(3) If $\rho \vDash \Gamma$, $\Gamma \vdash T^{-1}$, *and* $n \in \mathit{Scoped}_{\Theta,T^{-1}}$, *then* $(\rho, \overline{n/x}) \vDash \Gamma, (x : T^{-1})$
(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* $\gamma, \gamma' \in \Gamma$

Finally, we introduce a Θ-truncation lemma, which allows us to discard proof theoretic terms (γ, $\gamma \sqsubseteq \gamma'$) should they not be necessary for the proof.

Lemma 5.6 (Θ-truncation). *Assume an AST* n. *If* $\mathit{Var}(\alpha_S^{\gamma'})$ *does not occur in* $\mathit{FVs}^0(n)$, *and* $\mathit{Var}(\alpha_S^{\gamma'})$ *is the only variable tagged with classifier* γ', *then* $n \in \mathsf{Scoped}_{(\Theta,\gamma',\gamma\sqsubseteq\gamma'),\tau}$ *implies* $n \in \mathsf{Scoped}_{\Theta,\tau}$

Stratification of types and mode-indexing decomposes the fundamental lemma into many sub-lemmas; here we present one such sub-lemma:

Lemma 5.7 (Fundamental Lemma [c, $T^0\,!\,\Delta; \xi$] of the Scoped Logical Relation). *If* $\Gamma \vdash_{\mathbf{c}}^{\gamma} e : T^0\,!\,\Delta; \xi$ *then for* $\Theta = \pi_\gamma(\Gamma)$, *and for all* ρ *such that* $\rho \vDash \Gamma$, $[\![e]\!]_{\mathbf{c}}(\rho) \in \mathit{Scoped}_{\Theta,T^0\,!\,\Delta;\xi(\gamma)}$

Proof of Lemma 5.7 is by induction on the $\lambda_{\langle\langle \mathsf{op}\rangle\rangle^\gamma}$ typing rules. In the c-Lambda case, it suffices to show that for $\rho \vDash \Gamma$, **do** $x \leftarrow$ **mkvar** $\mathsf{erase}(S^0(\gamma'))$ **in do** body $\leftarrow [\![e]\!]_{\mathbf{c}}(\rho)$ **in return** $\mathsf{Lam}(x, \mathsf{body})$ is in $\mathsf{Scoped}_{\Theta,(S^0 \multimap \xi \to T^0)^0\,!\,\Delta(\gamma)}$. This reduces to **do** body $\leftarrow [\![e]\!]_{\mathbf{c}}(\rho, \alpha_S^{\gamma'}/x)$ **in return** $\mathsf{Lam}(\alpha_S^{\gamma'}, \mathsf{body})$. By anti-reduction (Lemma 5.4) it suffices to show that this term is in the logical relation. By weakening (Lemma 5.2), and the induction hypothesis (IH), $[\![e]\!]_{\mathbf{c}}(\rho, \alpha_S^{\gamma'}/x) \in \mathsf{Scoped}_{\Theta',T^0\,!\,\Delta;\xi(\gamma')}$, where $\Theta' = \Theta, \gamma', \gamma \sqsubseteq \gamma'$. It suffices to show that

$$\forall t \in \mathsf{Scoped}_{\Theta',T^0\,!\,\Delta;\xi(\gamma')},\ \mathbf{do}\ \mathsf{body} \leftarrow t\ \mathbf{in\ return}\ \mathsf{Lam}(\alpha_S^{\gamma'}, \mathsf{body})\ \text{in}\ \mathsf{Scoped}_{\Theta,(S^0 \multimap \xi \to T^0)^0\,!\,\Delta(\gamma)}$$

Applying Scoped-Induction,

(1) $t \in \mathsf{Scoped}_{\Theta',T^0\,!\,\Delta;\xi(\gamma')}$ reduces to some **return** n

do body $\leftarrow$ **return** n **in return** $\mathsf{Lam}(\alpha_S^{\gamma'}, \mathsf{body})$ reduces to **return** $\mathsf{Lam}(\alpha_S^{\gamma'}, n)$, where $\mathsf{Var}(\alpha_S^{\gamma'})$ is bound. By IH, $n \in \mathsf{Scoped}_{\Theta',T^0\,!\,\xi(\gamma')}$. Thus, all the free variables in n are permitted by γ'. By the typing rules, only α is annotated with classifier γ'. Hence, using Lemma 5.6, under Θ, the free variables of $\mathsf{Lam}(\alpha_S^{\gamma'}, n)$ are permitted by γ. The conclusion thus follows from anti-reduction.

(2) $t \in \mathsf{Scoped}_{\Theta',T^0\,!\,\Delta;\xi(\gamma')}$ reduces to $E[\mathbf{op}(n)]$, $\mathbf{op}(n)$ unhandled

As **do** body $\leftarrow [-]$ **in return** $\mathsf{Lam}(\alpha_S^{\gamma'}, \mathsf{body})$ introduces no handlers, the conclusion follows immediately from the Scoped-Induction hypothesis and anti-reduction.

Using the logical relation, and a type safety result identical to Bauer and Pretnar's [2008] Corollary 4.2, we prove the correctness of refined environment classifiers:

Theorem 5.1 (Correctness of Refined Environment Classifiers). *If* $\gamma_\perp \vdash_{\mathbf{c}}^{\gamma_\perp} e : T^0\,!\,\emptyset; \emptyset$, *and* $[\![e]\!]_{\mathbf{c}} = t$, *then for some* U, $\langle t; [-]; \emptyset\rangle \longrightarrow^* \langle \mathbf{return}\ n; [-]; U\rangle$, *and* $\mathit{FV}^0(n) = \emptyset$

5.3 Expressiveness of Refined Environment Classifiers

$\lambda_{\langle\langle \mathsf{op} \rangle\rangle^Y}$ prevents scope extrusion by looking only at the argument to the effect, not at the handler. In a well-typed $\lambda_{\langle\langle \mathsf{op} \rangle\rangle^Y}$ 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 12:

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

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

Listing 12. Refined environment classifiers allow variables to be passed to an effect, so long as the variable can never cause a scope extrusion error (e.g. z may be passed, since it is bound outside the handler definition).

Table 4 summarizes the expressiveness of refined environment classifiers on our set of programs; as shown, refined environment classifiers are less expressive than all the dynamic checks.

6 Implementation

We have implemented the various dynamic checks in the MacoCaml compiler, and made an implementation with the C4C check available as an artifact [Lee et al. 2025]. MacoCaml implements quotation via elaboration in a similar manner to the elaboration of §3.3, albeit targeting a lower-level intermediate language Lambda rather than ASTs. We have extended the elaboration with extrusion checking similarly to the extended elaborations of Sections 4.2 to 4.4.

The MacoCaml implementation closely follows the description in §4. The implementation realises **check**, **dlet**, and **err** as a *mode of use* of effects and handlers:

(1) **check** n is implemented by performing a `FreeVar` effect, passing it the free variables of n. $\mathbf{check}_{\mathsf{M}}\ n$ is similar, except that it additionally relies on a `Mute` effect that is performed within the handlers of effects besides `FreeVar` to implement the Eff-Op rule (Figure 5).
(2) **dlet**(α_R, t) is implemented as a handler of the `FreeVar` effect: it 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 (i.e. if all free variables are declared safe), or
  (b) performs another `FreeVar` effect, to check that the remaining free variables are declared safe. If the check returns successfully, the continuation is resumed.
(3) **err** is implemented as an unhandled `FreeVar` effect.

7 Related Work

Using mutation and control effects for code generation, particularly for let-insertion, has a long history [Lawall and Danvy 1994; Sumii and Kobayashi 2001]; Kameyama et al. [2011, §8] and Kameyama et al. [2015, §5.3] give a thorough overview. The danger of generating code with unbound variables has also become apparent. There are two lines of work dealing with the problem: prevention and detection. Most of the prevention research focuses on designing an appropriate type system, such as closed types [Calcagno et al. 2000] or environment classifiers [Taha and Nielsen 2003] (although the latter prevents scope extrusion arising from `eval` rather than from effects). The majority of type systems aimed at preventing scope extrusion are considerably more complex [Isoda et al. 2024; Kameyama et al. 2015; Kiselyov et al. 2016; Parreaux 2020], and are essentially variations of Nanevski et al.'s [2008] Contextual Modal Type Theory.

Besides types, one may also guarantee the absence of scope extrusion by restricting the scope of mutation (so-called *weak separability* [Westbrook et al. 2010]) or by restricting the scope of control effects by placing an effect handler under every future-stage binder [Kameyama et al. 2011]. Continuation-passing or monadic transformations [Swadi et al. 2006] amount to the same.

Most of the prevention techniques limit, often severely, the expressiveness of the language. Kameyama et al. [2015] proposed a set of benchmarks to evaluate expressiveness of program generation systems; at that time only Kameyama et al. [2015] passed all the benchmarks.

Whereas *prevention* techniques statically reject potentially unsafe code-generating programs when compiling the code generator, *detection* techniques operate when executing the code generator, alerting the metaprogrammer when a code fragment with a scope-extruded variable has been generated. Since generated code must eventually be compiled, the simplest detection technique is to do nothing during code generation, instead relying on the compiler of the generated code to report extrusion. This approach is what §2 calls the *lazy check*; as we stressed, it suffers from severe usability problems in practice. Kiselyov [2014] took efforts to implement the *eager check*, detecting scope extrusion as soon as it occurs, before the complete code is generated, with informative error messages. However, the design and implementation were not formalised. Hence, it was difficult to evaluate the check (or even to tell if it detects errors at the earliest possible point).

Very few of the prevention approaches have been implemented in systems that are used in practice (or, at least, that are used for realistic, larger-scale examples): examples include Mint [Westbrook et al. 2010], Contextual Squid [Parreaux 2020] and StagedHaskell [Kameyama et al. 2015]. Mint is very restrictive, outright prohibiting let-insertion and assert-insertion beyond binders. The other, type-based approaches, permit optimisations such as let-insertion and loop interchange, but they are very complex. Contextual Squid, implemented on top of Scala-2 macros, required access to Scala compiler internals which is no longer available in Scala-3. StagedHaskell relied on tricky Haskell type class programming, where type annotations are often required, and where the types can become quite complex and the error messages incomprehensible. The complexity of the types was unfortunately necessary: §4.1 of Kameyama et al.'s work showed subtle and serious problems that could arise with *unintendedly* bound variables in a version of their system with simpler types. Parreaux [2020] reports that some users found dealing with contextual types to be too much of a burden. Kiselyov [2014] gives more discussion of practical aspects of the scope extrusion check.

Many practical metaprogramming systems such as Template Haskell [Sheard and Jones 2002] rely on generation-time detection of scope extrusion, in particular the lazy check, i.e. offshoring all the detection to the compiler that compiles the generated code. The notable exception is (BER) MetaOCaml [Kiselyov 2014, 2024b], where *eager* scope extrusion detection is the principal feature.

8 Conclusions

We have presented the first formal framework for comparing scope extrusion checks, based on the calculus $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ and its elaboration into the $\lambda_{\mathsf{AST}(\mathsf{op})}$ core language. Using the framework, we have modelled the two main approaches to checking scope extrusion, lazy and eager checking, and developed a new check which combines the best properties of both, and which interacts well with effects and handlers. We have incorporated the new check into the MacoCaml implementation. Our framework also extends to modelling the refined environment classifier system for preventing scope extrusion, and we expect that it could be similarly extended to model other static systems.

Acknowledgments

We thank the anonymous POPL reviewers and Neel Krishnaswami, whose comments helped to improve the work, and Alistair O'Brien and Yulong Huang, whose comments helped to improve the artifact. This work is funded by Jane Street Capital, by Ahrefs, and by the Natural Sciences and Engineering Research Council of Canada.

References

Andrej Bauer and Matija Pretnar. 2014. An Effect System for Algebraic Effects and Handlers. *Logical Methods in Computer Science* Volume 10, Issue 4 (Dec. 2014). doi:10.2168/lmcs-10(4:9)2014

Nick Benton and Chung-Kil Hur. 2009. Biorthogonality, step-indexing and compiler correctness. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming* (Edinburgh, Scotland) *(ICFP '09)*. Association for Computing Machinery, New York, NY, USA, 97–108. doi:10.1145/1596550.1596567

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2017. Handle with care: relational interpretation of algebraic effects and handlers. *Proc. ACM Program. Lang.* 2, POPL, Article 8 (Dec. 2017), 30 pages. doi:10.1145/3158096

Cristiano Calcagno, Eugenio Moggi, and Walid Taha. 2000. Closed Types as a Simple Approach to Safe Imperative Multi-stage Programming. In *Automata, Languages and Programming*, Ugo Montanari, José D. P. Rolim, and Emo Welzl (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 25–36.

Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. 2003. Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection. In *Generative Programming and Component Engineering*, Frank Pfenning and Yannis Smaragdakis (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 57–76.

Jacques Carette, Mustafa Elsheikh, and W. Spencer Smith. 2011. A generative geometric kernel. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2011, Austin, TX, USA, January 24-25, 2011*, Siau-Cheng Khoo and Jeremy G. Siek (Eds.). ACM, 53–62. doi:10.1145/1929501.1929510

Tsung-Ju Chiang, Jeremy Yallop, Leo White, and Ningning Xie. 2024. Staged Compilation with Module Functors. *Proc. ACM Program. Lang.* 8, ICFP, Article 260 (Aug. 2024), 35 pages. doi:10.1145/3674649

Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. 1988. Abstract Continuations: A Mathematical Semantics for Handling Full Jumps. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming, LFP 1988, Snowbird, Utah, USA, July 25-27, 1988*, Jérôme Chailloux (Ed.). ACM, 52–62. doi:10.1145/62678.62684

Matthew Flatt and R. Kent Dybvig. 2020. Compiler and runtime support for continuation marks. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 45–58. doi:10.1145/3385412.3385981

Jun Inoue and Walid Taha. 2012. Reasoning about Multi-stage Programs. In *Programming Languages and Systems*, Helmut Seidl (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 357–376.

Kanaru Isoda, Ayato Yokoyama, and Yukiyoshi Kameyama. 2024. Type-Safe Code Generation with Algebraic Effects and Handlers. In *Proceedings of the 23rd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Pasadena, CA, USA) *(GPCE '24)*. Association for Computing Machinery, New York, NY, USA, 53–65. doi:10.1145/3689484.3690731

Yukiyoshi Kameyama, Oleg Kiselyov, and Chung chieh Shan. 2015. Combinators for impure yet hygienic code generation. *Science of Computer Programming* 112 (2015), 120–144. doi:10.1016/j.scico.2015.08.007

Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2011. Shifting the Stage: Staging with Delimited Control. *Journal of Functional Programming* 21, 6 (2011), 617–662. doi:10.1017/S0956796811000256

Oleg Kiselyov. 2012. Delimited control in OCaml, abstractly and concretely. *Theoretical Computer Science* 435 (2012), 56–76. doi:10.1016/j.tcs.2012.02.025 Functional and Logic Programming.

Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml. In *Functional and Logic Programming*, Michael Codish and Eijiro Sumii (Eds.). Springer International Publishing, Cham, 86–102.

Oleg Kiselyov. 2024a. Generating C: Heterogeneous metaprogramming system description. *Science of Computer Programming* 231 (2024), 103015. doi:10.1016/j.scico.2023.103015

Oleg Kiselyov. 2024b. MetaOCaml: Ten Years Later: System Description. In *Functional and Logic Programming: 17th International Symposium, FLOPS 2024, Kumamoto, Japan, May 15–17, 2024, Proceedings* (Kumamoto, Japan). Springer-Verlag, Berlin, Heidelberg, 219–236. doi:10.1007/978-981-97-2300-3_12

Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream fusion, to completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 285–299. doi:10.1145/3009837.3009880

Oleg Kiselyov, Yukiyoshi Kameyama, and Yuto Sudo. 2016. Refined Environment Classifiers. In *Programming Languages and Systems*, Atsushi Igarashi (Ed.). Springer International Publishing, Cham, 271–291.

Wiktor Kuchta. 2023. A proof of normalization for effect handlers. https://icfp23.sigplan.org/details/hope-2023/4/A-proof-of-normalization-for-effect-handlers Seattle, Washington, United States.

Julia L. Lawall and Olivier Danvy. 1994. Continuation-based partial evaluation. *SIGPLAN Lisp Pointers* VII, 3 (July 1994), 227–238. doi:10.1145/182590.182483

Michael Lee, Ningning Xie, Oleg Kiselyov, and Jeremy Yallop. 2025. Handling Scope Checks: A Comparative Framework for Dynamic Scope Extrusion Checks (Artifact). *Zenodo* (Nov. 2025). doi:10.5281/zenodo.17273738

Paul Blain Levy, John Power, and Hayo Thielecke. 2003. Modelling environments in call-by-value programming languages. *Information and Computation* 185, 2 (2003), 182–210. doi:10.1016/S0890-5401(03)00088-9

Yannis Lilis and Anthony Savidis. 2019. A Survey of Metaprogramming Languages. *ACM Comput. Surv.* 52, 6, Article 113 (Oct. 2019), 39 pages. doi:10.1145/3354584

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual Modal Type Theory. *Transactions on Computational Logic* 9, 3 (June 2008), 23:1–49.

Georg Ofenbeck, Tiark Rompf, and Markus Püschel. 2016. RandIR: differential testing for embedded compilers. In *Proceedings of the 7th ACM SIGPLAN Symposium on Scala, SCALA@SPLASH 2016* (Amsterdam, Netherlands), Aggelos Biboudis, Manohar Jonnalagedda, Sandro Stucki, and Vlad Ureche (Eds.). ACM, 21–30. doi:10.1145/2998392

Lionel Emile Vincent Parreaux. 2020. *Type-Safe Metaprogramming and Compilation Techniques For Designing Efficient Systems in High-Level Languages.* Ph. D. Dissertation. EPFL.

Gordon Plotkin and Ningning Xie. 2025. Handling the Selection Monad. *Proc. ACM Program. Lang.* 9, PLDI (2025). doi:10.1145/3729321

Matija Pretnar. 2015. An Introduction to Algebraic Effects and Handlers. Invited tutorial paper. *Electronic Notes in Theoretical Computer Science* 319 (2015), 19–35. doi:10.1016/j.entcs.2015.12.003 The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI)..

Tiark Rompf and Nada Amin. 2015. Functional pearl: a SQL to C compiler in 500 lines of code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, Kathleen Fisher and John H. Reppy (Eds.). ACM, 2–9. doi:10.1145/2784731.2784760

Gabriel Scherer. 2017. Deciding equivalence with sums and the empty type. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 374–386. doi:10.1145/3009837.3009901

Tim Sheard and Simon Peyton Jones. 2002. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell* (Pittsburgh, Pennsylvania) *(Haskell '02)*. Association for Computing Machinery, New York, NY, USA, 1–16. doi:10.1145/581690.581691

KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 206–221. doi:10.1145/3453483.3454039

Nicolas Stucki, Aggelos Biboudis, and Martin Odersky. 2018. A practical unification of multi-stage programming and macros. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2018, Boston, MA, USA, November 5-6, 2018*, Eric Van Wyk and Tiark Rompf (Eds.). ACM, 14–27. doi:10.1145/3278122.3278139

Eijiro Sumii and Naoki Kobayashi. 2001. A Hybrid Approach to Online and Offline Partial Evaluation. *Higher-Order and Symbolic Computation* 14, 2–3 (Sept. 2001), 101–142.

Kedar Swadi, Walid Taha, Oleg Kiselyov, and Emir Pašalić. 2006. A Monadic Approach for Avoiding Code Duplication When Staging Memoized Functions. In *PEPM* (Charleston, SC). 160–169.

Walid Taha. 1999. *Multi-Stage Programming: Its Theory and Applications.* Ph. D. Dissertation. Halmstad University, Sweden. https://urn.kb.se/resolve?urn=urn:nbn:se:hh:diva-15052

Walid Taha and Michael Florentin Nielsen. 2003. Environment classifiers. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, Alex Aiken and Greg Morrisett (Eds.). ACM, 26–37. doi:10.1145/604131.604134

W. W. Tait. 1967. Intensional Interpretations of Functionals of Finite Type I. *The Journal of Symbolic Logic* 32, 2 (1967), 198–212. http://www.jstor.org/stable/2271658

Fei Wang, Daniel Zheng, James Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. 2019. Demystifying differentiable programming: shift/reset the penultimate backpropagator. *Proc. ACM Program. Lang.* 3, ICFP, Article 96 (July 2019), 31 pages. doi:10.1145/3341700

Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. 2010. Mint: Java multi-stage programming using weak separability. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 400–411. doi:10.1145/1806596.1806642

Ningning Xie, Matthew Pickering, Andres Löh, Nicolas Wu, Jeremy Yallop, and Meng Wang. 2022. Staging with class: a specification for typed template Haskell. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–30. doi:10.1145/3498723

Ningning Xie, Leo White, Olivier Nicole, and Jeremy Yallop. 2023. MacoCaml: Staging Composable and Compilable Macros. *Proc. ACM Program. Lang.* 7, ICFP, Article 209 (Aug. 2023), 45 pages. doi:10.1145/3607851

Jeremy Yallop. 2017. Staged generic programming. *Proc. ACM Program. Lang.* 1, ICFP, Article 29 (Aug. 2017), 29 pages. doi:10.1145/3110273

Jeremy Yallop and community contributors. 2025. effects-bibliography: A collaborative bibliography of work related to the theory and practice of computational effects. GitHub repository. https://github.com/yallop/effects-bibliography

Jeremy Yallop and Oleg Kiselyov. 2019. Generating mutually recursive definitions. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (Cascais, Portugal) *(PEPM 2019)*. Association for Computing Machinery, New York, NY, USA, 75–81. doi:10.1145/3294032.3294078

Jeremy Yallop, Ningning Xie, and Neel Krishnaswami. 2023. flap: A Deterministic Parser with Fused Lexing. *Proc. ACM Program. Lang.* 7, PLDI, Article 155 (June 2023), 24 pages. doi:10.1145/3591269

Received 2025-07-10; accepted 2025-11-06