arXiv:2601.18793v1 [cs.PL] 26 Jan 2026

处理作用域检查（扩展版）

动态作用域外逸检查的比较框架

MICHAEL LEE，剑桥大学，英国NINGNING XIE，多伦多大学，加拿大OLEG KISELYOV，东北大学，日本JEREMY YALLOP，剑桥大学，英国

元编程与效应处理器的交互方式出人意料，有时甚至会带来不良影响。作用域外逸就是一个例子：生成作用域不良代码。作用域外逸既可以通过静态类型系统提前防止，也可以通过动态检查事后检测。静态类型系统在理论上是存在的，但在实践中面临诸多实现和易用性问题。相比之下，动态检查在实践中已有应用（例如 MetaOCaml 中），但在理论上尚未得到充分研究。因此，元编程语言的设计者在检查机制的设计与实现方面缺乏指导。我们首次对动态作用域外逸检查进行了形式化研究，并引入一种演算($\lambda_{\langle\langle \mathrm{op} \rangle\rangle}$)，用于描述和评估检查。此外，我们提出了一种新颖的动态检查——“值得关注原因”检查（Cause-for-Concern check），并证明其正确性，在不涉及其实现的情况下对其进行刻画，同时论证它结合了现有动态检查的优点。最后，我们使用精化环境分类器扩展了框架；这种分类器能够静态地防止作用域外逸。我们还将其表达能力与动态检查进行了比较。

CCS概念：• **软件及其工程** → 控制结构。

附加关键词 与短语：效应处理器、代码生成、元编程、作用域外逸

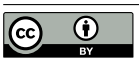
ACM参考文献格式：
Michael Lee、Ningning Xie、Oleg Kiselyov 和 Jeremy Yallop。2026年。处理作用域检查（扩展版）：动态作用域外逸检查的比较框架。*ACM 程序设计语言会议论文集*10，POPL，第 39 篇文章（2026年1月），42 页。
https://doi.org/10.1145/3776681

1 引言

多阶段编程语言已被用于为各种领域编写代码生成器，涵盖数据库查询、流处理、几何、解析和可微编程 [Carette 等人 2011；Kiselyov 等人 2017；Rompf 和 Amin 2015；Wang 等人 2019；Yallop 等人 2023]。用于代码生成的语言构造通常带有强有力的保证。例如，用 MetaML 语言编写的类型正确的代码生成器 [Taha 1999] 保证永远不会生成类型错误的代码。然而，当代码生成构造与效应结合时，这些保证会减弱 [Calcagno 等人 2000；Isoda 等人 2024；Kameyama 等人 2015，2011；Kiselyov 2014；Kiselyov 等人 2016；Parreaux 2020]。

特别是，代码生成构造与效应的结合可能导致作用域外逸：意外生成包含未绑定变量的代码。例如，在 MacoCaml 程序列表 1中，代码生成中效应处理器的使用使变量x逸出其作用域：

作者联系信息：Michael Lee，michael.lee@cl.cam.ac.uk，剑桥大学，英国；Ningning Xie，ningningxie@cs.toronto.edu，多伦多大学，加拿大；Oleg Kiselyov，oleg@okmij.org，东北大学，日本；Jeremy Yallop，jeremy.yallop@cl.cam.ac.uk，剑桥大学，英国。



本作品采用知识共享署名 4 .0 国际许可协议授权 .
©2026年 版权归所有者/作者所有。ACM
2475-1421/2026/1-第 39 篇文章
https://doi.org/10.1145/3776681

```
MacoCaml
1  尝试 <<令x = 3 在 $(执行 (Extrude <<x>>)) >>
2  带效应 Extrudey, k → <<$y + 1 >>
```

列表 1。作用域外逸示例

第 1 行安装了一个处理器，其主体为 `<< let x = 3 in $(...) >>` 使用代码引用来构造函数应用的代码。引号内的表达式 `<< >>` 不会立即求值，而是构造一段将来可能被求值的代码。然而，以`$` 为前缀的子表达式会立即求值，并执行效应`Extrude`，将控制权转移给最近安装的处理器。第 2 行展示了该处理器：它将参数`<<x>>` 绑定到`y` ，并将由`try` 和 `perform` to `k`界定的延续。该处理器丢弃延续，并使用参数构造代码 `<< x + 1 >>`，其中`x` 未绑定。

在列表 1中，外逸很简单： `<<x>>` 离开了其绑定器的作用域 `<<令 x = 3 在 ...>>`，并且再也不会返回。然而，调用 `k`的处理器可能会使控制流重新进入该作用域：

```
MacoCaml
2  使用效应外逸y, k → 继续k <<$y + 1 >>
```

列表 2。修订处理器列表 1 以将 `x` 重新纳入作用域

这里，`继续`恢复延续，并将控制权返回到`perform` 被调用的位置，使得程序最终求值得到一个作用域正确的代码值 `<< let x = 3 in x + 1 >>`。

作用域外逸在实践和理论中都是一个问题。多阶段语言所提供的强保证，减轻了程序员调试生成代码中类型错误的负担，但作用域外逸又让这一负担卷土重来。Ofenbeck 等人 [2016] 报告了一个例子：在重构 LMS 系统以解决性能问题时，使用效应来执行代码移动优化。这些效应无意间导致了作用域外泄错误。尽管这些错误的原因很简单，但由于涉及的变量数量众多，而且难以确定代码生成器的哪一部分生成了问题代码，因此修复起来耗时很长。

为了避免程序员需要调试生成代码，带有效应的多阶段语言通常会提供帮助，以识别作用域外逸。关键问题是何时检查问题。一种方法是在类型系统中跟踪潜在的外逸，并拒绝无法证明安全的程序 [Calcagno 等人 2000；Isoda 等人 2024；Kiselyov 等人 2016；Parreaux 2020；Westbrook 等人 2010]。然而在实践中，要兼顾表达能力（使效应与代码生成能够良性地交互，例如代码移动优化）和严格性（排除所有潜在的外逸）并不容易。在复杂的类型系统与较简单但安全性较低的系统之间做选择时，用户往往更倾向于后者 [Parreaux 2020]。

另一种方法是在代码生成期间进行动态检查，允许可能不安全的代码生成器运行，并在作用域外逸发生时识别出来。这种更宽松的方法尚无现成的理论基础，但在实践中更为常见，部分原因是它可以融入现有的多阶段语言——例如 MetaOCaml、Scala [Stucki 等人 2018] 和 Typed Template Haskell [Xie 等人 2022] ——而不会影响它们的类型系统。

动态检查有多种可能的设计。一种极端做法是惰性地检查作用域，待代码生成完成后再进行。最初的 MetaOCaml 语言采用惰性检查，因为其静态类型系统、环境分类器 [Taha 和 Nielsen 2003],能够阻止某些形式的外逸，但无法阻止所有情况。另一种极端做法是，BER MetaOCaml 每次构造代码引述时都会急切地检查作用域 [Kiselyov2014、2024a]。这两种做法都不是最优的。惰性检查无法提供有用信息，会产生难以调试的错误 [Ofenbeck 等人 2016], 效率低下，发现错误的时间比急切检查晚得多 [Kiselyov 2014],，而且在某些系统中还可能将变量绑定

到非预期的方式 [Kameyama 等人 2015]。另一方面，急切检查在某种意义上并不正确，具体将在§4.3.1中阐明：在某些病态情形下，它无法检测自由变量的出现（代码清单 8和9）。此外，急切检查无法感知延续：例如，它会错误地拒绝代码清单 2 [Kiselyov 2014]中的安全代码生成器。

为了建立动态检查理论，我们引入支持效应与处理器多阶段编程的 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ and $\lambda_{\mathsf{AST}(\mathsf{op})}$ 演算，并展示如何用它们描述和比较急切检查与惰性检查。我们还介绍一种在 MacoCaml 中实现的新型检查——值得关注（C4C）检查，它结合了急切检查和惰性检查的优点。

贡献。 第2节 通过一个更大的示例非正式地介绍急切与惰性方法，并引入我们新颖的 C4C 检查。后续各节将介绍以下技术贡献：

- 两个新颖的演算： $\lambda_{\langle\langle \text{操作} \rangle\rangle}$ 和 $\lambda_{\text{抽象语法树(操作)}}$，旨在研究带有效应与处理器的类型化多阶段编程（第3节）。 $\lambda_{\langle\langle \text{操作} \rangle\rangle}$ 是一种类型安全的两阶段演算，也是首个支持在编译时和运行时阶段均使用效应处理器的演算。
- 一个基于 $\lambda_{\langle\langle \text{操作} \rangle\rangle}$ 和 $\lambda_{\text{抽象语法树(操作)}}$ 的框架，可将不同的作用域外逸检查形式化为一系列从 $\lambda_{\langle\langle \text{操作} \rangle\rangle}$ 到 $\lambda_{\text{抽象语法树(操作)}}$ (§4) 的 elaboration，并据此研究了多种设计：惰性检查（§4.2）、急切检查（§4.3）以及我们新提出的 C4C 检查（§4.4）。
- 对 $\lambda_{\langle\langle \text{操作} \rangle\rangle}$ 和 $\lambda_{\text{抽象语法树(操作)}}$ 的扩展，加入了 Kiselyov 等人的 [2016] 精化环境分类器(§5.1)，并通过逻辑关系证明了其正确性（§5.2），还将其表达能力与动态检查进行了比较评估（§5.3）。
- MacoCaml 语言中三种动态检查的实现（§6）。带有 C4C 检查的实现可作为制品获取，见 [Lee 等人 2025]。

最后，§7 介绍相关工作，并由§8作结。

2 概述

虽然有许多元编程语言，但我们的讨论将以 Maco- Caml 为基础 [Chiang 等人 2024；Xie 等人 2023]。MacoCaml 项目扩展了 OCaml 编程语言，加入了用于编译时程序生成的元编程功能：用于表示类型为α `expr` 的代码的类型构造器 α，以及用于构造 `<< >>` 和拼接 `$` 形式，以生成 `expr` 值。

从较高层面来看， α `expr` 的元素对应于类型为 α的抽象语法树。代码引述会将表达式转换为抽象语法树，而拼接会停止转换，从而允许在抽象语法树创建期间进行求值。例如，元程序 `<<$(print_int (1+2); <<1+2>>) + 0>>` 可以理解为`加法((print_int(1+2); 加法(Int(1), Int(2))), Int(0))`。我们将在§3中精确定义这一概念模型。

列表 3 展示了我们的贯穿示例，改编自Kiselyov [2014]，，它会生成矩阵乘法的代码。参数a、b和c 都是二维数组；`a.(0)` 访问表示矩阵第一行的数组。实际的矩阵乘法实现通常会采用各种复杂的优化，但这段简单代码足以展示代码生成与效应之间的交互，这正是本文关注的重点。

MacoCaml

```
宏迭代 a body = << 执行 i = 0 至 length $a - 1 执行 $(body <<i>>) 完成 >>
宏mmul a b c =
  迭代 a @@ 函数 i →
  迭代 <<$a.(0)>> @@ 函数 k →
  迭代 <<$b.(0)>> @@ 函数 j →
    << $c.($i).($j) ← $c.($i).($j)
                    + $a.($i).($k)
                    * $b.($k).($j) >>
```

列表 3. 生成常见矩阵乘法代码的分阶段代码

第 1 行定义了一个宏（即编译时函数），它根据数组类型项a 的代码以及以循环变量i调用 body 函数所得的结果，生成for 循环的代码。第 2–8 行定义了另一个宏mmul ，它使用iter 构造三重嵌套循环。@@操作符表示函数应用，用于避免括号过多。

2.1 分阶段处理中的效应处理器

效应处理器是一种可组合且可定制的模拟效应机制 [Pretnar2015]。与元编程一样，效应处理器有许多变体，我们的讨论以允许多次续延的深层无名效应处理器为基础（相关演算见§3.1）。在本节中，我们的示例使用 OCaml 的深层无名效应处理器，它们只允许单次调用续延 [Sivaramakrishnan 等人 2021]。

鉴于元编程和效应处理器的实用性，考虑同时提供二者的语言应如何协调它们之间的交互是明智的。完全分离可能并不可取，因为效应对于放宽栈纪律非常有用；否则，栈纪律会将生成代码与生成它的代码的结构紧密耦合起来。具体而言，效应允许程序员轻松执行 *let* 插入 [Kameyama 等，2011；Yallop 和 Kiselyov 2019]（列表 4）：

```
MacoCaml
1  类型_效应.t += Genlet : 整数表达式 * 整数表达式 →整数表达式 t
2  宏genlet x e = 执行 (Genlet (x, e))
3  宏处理_genlet body i = 尝试 body i
4                          带有效应 Genlet (x, e), k →
5                            如果x ==i那么 <<令y =$e于$(继续k <<y>>)>>
6                            否则继续k(genletxe)
```

列表 4. 效应处理器、引号和拼接结合起来实现 let 插入

第 1–2 行定义了一个新的效应构造器 Genlet 以及一个执行Genlet 效应的编译时函数。Genlet 效应有两个参数：第一个用于标识新绑定的插入点，第二个是要绑定的表达式。第 3–6 行定义了一个处理Genlet 的处理器，它要么将一个 let 绑定安装到栈上（第 5 行），要么将效应转发给外层处理器（第 6 行）。调用 处理_genlet body i 会用一个 let 绑定包装由 body i 生成的代码；在执行 body期间，每次调用 genlet i e 都会添加一个这样的绑定。

在mmul 示例（列表 3）中，表达式$a.($i).($k) 不依赖于j，因此可以将其提升到循环之外，这种优化称为循环不变代码外提。效应是在分阶段程序中执行此类优化的便捷方式，同时还能保持生成代码的结构。没有效应时，栈纪律会将生成代码与生成器代码的结构耦合起来，因此代码移动需要更新生成器，将表达式$a.($i).($k) 提升到iter <<$b.(0)>>表达式之上。

列表 5 展示了更新后使用 let 插入的示例。在第 1 行，处理_genlet 现在会在生成每个循环时安装一个 Genlet 处理器，而第 7 行和第 8 行现在会执行 genlet 效应。

```
MacoCaml
1  宏遍历 a 的主体 = << 对 i = 0 到 length $a - 1 执行 $(处理_genlet 主体 <<i>>) 完成 >>
2  宏mmul a b c =
3    遍历 a @@ 函数 i →
4    遍历 <<$a.(0)>> @@ 函数 k →
5    遍历 <<$b.(0)>> @@ 函数 j →
6      << $c.$i.$j ← $c.($i).($j)
7                  + genlet k <<$a.($i).($k)>>(*而不是$a.($i).($k)*)
8                  * genlet j <<$b.($k).($j)>>(*而不是$b.($k).($j)*) >>
```

列表 5。效应处理器、引号和拼接结合起来执行 let 插入

遗憾的是，进行此类优化时很容易出错，从而导致作用域外逸。假设根据数组在内存中的布局以及缓存预取器的具体设计，交换 j 循环和 k 循环的顺序会更高效：

```
MacoCaml
3  迭代 a @@函数 i  →迭代 <<$b.(0)
4  >> @@函数j  →迭代 <<$a.(0)>> @@函
5  数k  → ...
```

如果程序员意识到这一点，就可以在不更改 let 插入代码的情况下执行这种交换。但对 `genlet in` 列表 5 的使用假定 k 循环位于 j 循环之上，因此这一更改会导致作用域外逸。

或者，程序员可能会确定错误的 let 插入点：

```
MacoCaml
6  << $c.$i.$j ← $c.($i).($j)
7              + 生成器 i <<$a.($i).($k)>>(*应该是 k，而不是 i*)
8              * 生成器 k <<$b.($k).($j)>>(*应该是 j，而不是 k*) >>
```

这个错误也会导致作用域外逸。检测此类错误需要进行作用域外溢检查。

2.2 检查作用域外逸

理论上，可以采用**惰性检查**(§4.2)，这种方法会等到程序生成过程结束后，才检查生成的程序是否包含自由变量 [Kiselyov2014]。惰性检查相当于对生成代码进行类型检查，这种方法用于最初的 MetaOCaml 实现 [Taha 和 Nielsen 2003], LMS [Ofenbeck 等人 2016], 以及其他系统中。惰性方法有三个缺点：第一，效率低下，因为即使发生错误，它仍允许代码生成器继续运行直至完成。第二，它产生的错误消息信息不足，指向的是生成代码，而非代码生成器。第三，在某些系统中，它可能会以非预期的方式绑定变量。

因此，BER MetaOCaml 转而采用**急切检查** [Kiselyov2014，2024a], 我们将在§4.3中对其进行形式化描述。通过在代码生成过程中的多个阶段进行检查，急切检查能够及早发现错误，并给出信息丰富的错误消息 [Kiselyov 2014，§5。1]。

尽管急切检查比惰性检查能提供更好的错误报告，但它无法让效应处理器与元编程像人们所期望的那样自由地交互。例如，效应处理器的一种常见用途是参数化：为同一效应选择不同的处理器，可以让同一段代码在各种上下文中进行特化 [Wang 等人 2019]。参数化以非常简单的方式使用效应处理器；我们在示例中采用这种方式，以说明即便是对效应处理器的直接使用，也会与急切检查产生不良交互。

为了通过参数化扩展矩阵乘法生成器，假设我们希望默认生成`e1 + e2 * e3`，并在性能优先于精确保留预期浮点行为的上下文中，生成对融合乘加指令的调用`__fma(e1,e2, e3)`。列表 6 展示了一种在这些选项之间进行参数化的方法：首先将选择抽象为一种效应，**融合乘加**（第 1–2 行）；然后修改`mmul` 的主体以执行该效应（第 6–8 行）；接着定义可用于调整生成过程的处理器（第 10–13 行）；最后，在对`mmul` 的调用周围安装一个处理器（第 15 行）。第 15 行使用了顶层拼接，在 MacoCaml 中，它用于将宏生成的代码插入到更大的程序中。

遗憾的是，使用**融合乘加** 在列表 6 中并不允许通过急切检查，因为执行第 11 行或第 13 行时会抛出作用域外泄错误。例如，在hdl_fma_def

处理器中，代码模板 <<$x + $y * $z>> 会在一个i、j和k 均为自由变量的作用域中求值。然而，由于x 被绑定到 <<c.($i).($j)>>，急切检查会报告作用域外逸。

MacoCaml

```
类型_ Effect.t += 融合乘加 : 整数表达式 * 整数表达式 * 整数表达式 → 整数表达式 t
宏fmax y z = 执行 (融合乘加 (x, y, z))

宏mmul a b c =
  ...
    <<$c.($i).($j) ← $(融合乘加 <<$c.($i).($j)>>
                          (genlet k <<$a.($i).($k)>>)
                          (genlet j <<$b.($k).($j)>>))>>

宏hdl_fma_定义主体 = 尝试 主体 ()
                      带效应融合乘加(x,y,z),    k → 继续 k << $x + $y * $z >>
宏hdl_fma_优化主体 = 尝试 主体 ()
                      带有效应融合乘加(x,y,z),  k → 继续 k << __fma($x,    $y, $z) >>

令 代码 = $(hdl_融合乘加_定义 @@ 函数 () → mmul <<a>> <<b>> <<c>>)
```

列表 6. 用于选择乘加指令的处理器

为了支持效应处理器与元编程之间更灵活的交互，我们引入了一种新颖的延续感知 **C4C** 检查，详见 §4.4，它允许像列表 6这样的代码运行至完成，而不会报告作用域外逸。

3 演算

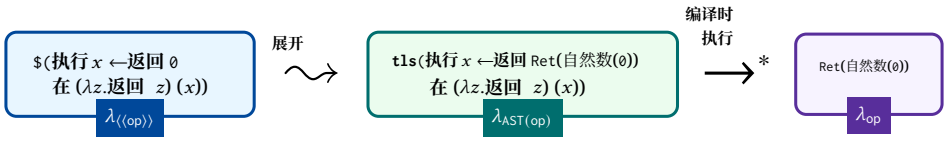


图 1. $\lambda_{\langle\langle op\rangle\rangle}$ 首先被展开为 $\lambda_{AST(op)}$，随后在 **编译时** 执行，以获得运行时 λ_{op} 程序的抽象语法树。 **tls** 是一个标记，用于追踪 **顶层拼接** （拼接（§3.2）

为了探讨动态作用域外逸检查，我们引入两种新演算，用于研究类型化多阶段编程与效应和处理器之间的交互： $\lambda_{\langle\langle op\rangle\rangle}$和 $\lambda_{AST(op)}$。 $\lambda_{\langle\langle op\rangle\rangle}$ (§3.1) 提供了以引号和拼接形式实现的元编程，以及效应处理器。 $\lambda_{AST(op)}$ 提供了以 AST 构造器实现的元编程，以及效应处理器。遵循 Calcagno 等人 [2003]， $\lambda_{\langle\langle op\rangle\rangle}$ 没有操作语义； $\lambda_{\langle\langle op\rangle\rangle}$ 中的程序会被展开为 $\lambda_{AST(op)}$ (§3.3)，随后可在其中执行，从而获得不含引号和拼接的运行时程序的抽象语法树。此过程总结于 图 1。展开简化了操作语义，也是一种插入动态检查的便捷机制（§4）。 $\lambda_{\langle\langle op\rangle\rangle}$ 和 $\lambda_{AST(op)}$ 都是类型安全的（§3.4）。

3.1 源语言： $\lambda_{\langle\langle \mathbf{op}\rangle\rangle}$

$\lambda_{\langle\langle op\rangle\rangle}$ (图 2) 是一种同时提供元编程（以引号 $\langle\langle e\rangle\rangle$和拼接 $\$e$的形式）以及效应处理器的语言 [Pretnar 2015]。语法 $\lambda_{\langle\langle op\rangle\rangle}$ 项分为值、表达式和处理器，类似于细粒度按值调用方法 [Levy 等人2003]。忽略引号和拼接，并添加一个续延项构造子 $\kappa x.e$（它无法显式书写，但可在归约过程中生成），即可得到标准的效应与处理器基础演算的语法 [Biernacki 等人 2017；Isoda 等人 2024；Pretnar 2015]，，本文将其称为

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

语法

值	v	::=	$x \mid m \in \mathbb{N} \mid \lambda x.e$
表达式	e	::=	$v_1\ v_2 \mid$ **返回** $v \mid$ **执行** $x \leftarrow e_1$ **在** $e_2 \mid$ **操作**$(v) \mid$ **处理** e **用** $\{h\} \mid$ **继续** $v_1\ v_2$ $\mid e \mid \$e$
处理器	h	::=	**返回**$(x) \mapsto e \mid h;$ **操作**$(x, k) \mapsto e$

效应集

运行时 $\xi ::= \emptyset \mid \xi \cup \{\text{操作}_i^0\}$

编译时 $\Delta ::= \emptyset \mid \Delta \cup \{\text{操作}_i^{-1}\}$

类型上下文

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

类型

0 级

值	$S^0,\ T^0 ::=$	$\mathbb{N}^0 \mid (S \xrightarrow{\xi} T)^0$ $\mid (S \xrightarrow{\xi} T)^0$
计算		$T^0 ! \ \xi \mid T^0 ! \ \Delta \mid T^0 ! \ \Delta;\ \xi$ $\mid (S ! \xi_1 \Longrightarrow T ! \xi_2)^0 ! \ \Delta$
处理器		$(S ! \xi_1 \Longrightarrow T ! \xi_2)^0$

级别 −1

值	$S^{-1},\ T^{-1} ::=$	$\mathbb{N}^{-1} \mid (S \xrightarrow{\Delta} T)^{-1}$ $\mid (S \xrightarrow{\Delta} T)^{-1} \mid$ 代码$(T^0 ! \ \xi)^{-1}$
计算		$T^{-1} ! \Delta$
处理器		$(S ! \Delta_1 \Longrightarrow T ! \Delta_2)^{-1}$

图 2。 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ 语法和类型。为清晰起见，省略了可推断的层级：例如，$(S \xrightarrow{\xi} T)^0$ 表示$(S^0 \xrightarrow{\xi} T^0)^0$。

（并且在 λ_{op} 中有所描述；参见附录 B）。简而言之，**返回** v 将一个值提升为表达式，而 **执行** $x \leftarrow e_1$ **于** e_2 则用于对表达式进行序列化。 **操作**(v) 执行一个效应，使当前计算暂停，并抛出一个值 v，由某个处理器捕获 h；该处理器通过**处理** e **配合**h来安装。在处理器主体中，可以使用**继续** $k\ v$ 恢复暂停的程序（k），并将该值 v 插入到执行效应的位置。

元编程系统在几个关键维度上有所不同：它们可以是同质的（生成语言与生成出的语言相同）或异质的，可以是两阶段或多阶段的，也可以是编译时或运行时的 [Lilis 和 Savidis 2019]。$\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ 提供 同质、两阶段、编译时的 元编程。许多实用系统，如 MacoCaml 和 MetaOCaml，都是同质的。许多多阶段编程的实际用例只涉及两个阶段 [Inoue 和 Taha 2012]，，而作用域外逸也常在两阶段系统中研究 [Isoda 等人，2024；Kiselyov 等人 2016]。同样，$\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ 提供 深层无名处理器，它们允许多次续延，可用于建模 OCaml 效应处理器，不过将其推广到了多次续延。尽管 OCaml 不支持多次续延，但它们对于 if/case 插入很有用 [Yallop 2017]。其他效应系统还允许浅层处理器或 sheep 处理器、允许具名处理器，或仅允许单次续延 [Yallop 和社区贡献者 2025]：我们不研究这些系统。

按照 Calcagno 等人 [2003]， $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ 没有操作语义，而是被 elaboration 为 $\lambda_{\mathsf{AST(op)}}$。$\lambda_{\mathsf{AST(op)}}$ 程序随后可以执行，以获得运行时 λ_{op} 程序的抽象语法树，其中不含引号和拼接。此过程总结于 图 1。elaboration 简化了操作语义，并且是插入动态检查（§4）的一种便捷机制。

只有表达式可以被引用（值和处理器不可以）：因此，引号必须生成运行时计算。例如，1 不是有效语法，必须写作 **返回** 1。类似地，**返回** 1 是一个表达式，而不是一个值，因此必须写作 **执行** $a \leftarrow$ **返回** 1 **在 op 中**(a)，而不能写作 **op(** **返回** 1 **)**。不过，我们会滥用记号，用 **op(1)** 代替**执行** $a \leftarrow$ **返回** 1 **在 op 中**(a)。

3.1.1 类型系统。 图 2 总结了 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ 这些类型。为说明类型系统的作用，请考虑以下运行示例 e 其中 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ 扩展了算术运算：

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

表 1. 0级类型的分层

		运行时		
		值	计算	处理器
编译时	值	T^0	$T^0\,!\,\xi$	$(S^0\,!\,\xi_1 \Longrightarrow T^0\,!\,\xi_2)^0$
	计算	$T^0\,!\,\Delta$	$T^0\,!\,\Delta;\ \xi$	$(S^0\,!\,\xi_1 \Longrightarrow T^0\,!\,\xi_2)^0\,!\,\Delta$

无法直接创建

语法的类型 层级 0值 v

句法 级别 0 表达式 e

句法 级别 0 处理器 h

这里 e 是一个编译时函数，它 (1) 接受一个类型为$\mathbb{N}!\,\{\mathsf{print}\}$的运行时计算的抽象语法树，(2) 执行一个编译时效应（**获取**），并且 (3) 返回一个类型不同的运行时计算的抽象语法树。该程序具有以下类型：

$$(\text{代码}(\mathbb{N}^0!\{\mathsf{print}\})^{-1} \xrightarrow{\{\mathsf{get}\}} \text{代码}(\mathbb{N}^0\,!\,\{\mathsf{print},\ \text{读取整数}\})^{-1})^{-1}$$

3 编译时效应

1 编译时函数

2 输入：运行时计算的抽象语法树，类型为 $\mathbb{N}!\,\{$打印$\}$的计算

4 输出：运行时计算的抽象语法树，类型为 $\mathbb{N}!\,\{$打印, 读取整数$\}$的计算

类型系统将类型分为编译时（T^{-1}）和运行时（T^0）级别。函数 e具有编译时类型（$S \to T$）$^{-1}$，不能作为类型为（$S \to T$）^{0}的运行时函数使用。

为了支持对运行时程序进行编译时操作，代码$(T^0\,!\,\xi)^{-1}$类型使 0 级计算的抽象语法树在 -1级可用。只有计算，而不是值或处理器，才能转换为抽象语法树。

效应集分为 Δ （编译时）和 ξ （运行时）。在我们的示例中，假设e 在编译时应用于某个具有正确类型的项 e'。该应用具有编译时效应 get，并返回一个包含两个运行时效应 print 和 readInt 的抽象语法树。

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

拼接应用的结果会将编译时抽象语法树提升为运行时类型，该类型在**编译时和运行时**都具有未处理效应。

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

我们分别用两个集合跟踪编译时和运行时效应。编译时效应记录在 $\Delta(=\{\mathsf{get}\})$ 中，运行时效应记录在 $\xi(=\{\mathsf{print},\ \mathsf{readInt}\})$中。区分编译时和运行时效应使类型分层（表 1）：运行时的计算可能在编译时是一个值，反之亦然。

在 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$中，使用一个在层级0 类型化的项，总是会产生编译时计算（表 1的第二行）。例如，层级0 值会被 elaboration 为编译时计算（$T^0\,!\ \Delta$），并求值为运行时值的抽象语法树（§3.3）：

$T^0\,!\,\Delta$编译时计算，运行时值成员：0 级值 v，例如 $\lambda x.$**返回** $xT^0\,!\,\Delta;\ \xi$编译时计算，运行时计算成员：0 级表达式 e，例如 **返回1** $(S^0\,!\,\xi_1 \Longrightarrow T^0\,!\,\xi_2)^0\,!\,\Delta$编译时计算，运行时处理器成员：0 级处理器 h，例如 $\{$**返回**$(x) \mapsto$ **返回** $x\}$

表 2。九种 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ 类型判断

	值 (v)	**表达式** (e)	**处理器** (h)
编译 (c)	$\Gamma \vdash^{0}_{\mathbf{c}} v : T^0 \,!\, \Delta$	$\Gamma \vdash^{0}_{\mathbf{c}} e : T^0 \,!\, \Delta;\ \xi$	$\Gamma \vdash^{0}_{\mathbf{c}} h : (S^0 \,!\, \xi_1 \Longrightarrow T^0 \,!\, \xi_2)^{\,0} \,!\, \Delta$
引用 (q)	$\Gamma \vdash^{0}_{\mathbf{q}} v : T^0 \,!\, \Delta$	$\Gamma \vdash^{0}_{\mathbf{q}} e : T^0 \,!\, \Delta;\ \xi$	$\Gamma \vdash^{0}_{\mathbf{q}} h : (S^0 \,!\, \xi_1 \Longrightarrow T^0 \,!\, \xi_2)^{\,0} \,!\, \Delta$
拼接 (s)	$\Gamma \vdash^{-1}_{\mathbf{s}} v : T^{-1}$	$\Gamma \vdash^{-1}_{\mathbf{s}} e : T^{-1} \,!\, \Delta$	$\Gamma \vdash^{-1}_{\mathbf{s}} h : (S^{-1} \,!\, \Delta_1 \Longrightarrow T^{-1} \,!\, \Delta_2)^{-1}$

$$\underset{\mathbf{c}}{\lambda x.}\ \underset{\mathbf{s}}{\$(\textbf{执行}\ f \leftarrow (\underset{\mathbf{s}}{\lambda y.\$(y)} + 2)\ \textbf{在执行}\ a \leftarrow 1\ \textbf{在}\ fa)} + \underset{\mathbf{c}}{3}$$

图 3. 一个标注了编译器模式的元程序

因此，语法上的值（v）在0 级没有值类型（T^0）。语法与类型之间的关系比 λ_{op}中的情况更复杂。相比之下，0 级编译时值类型（表 1中的第一行）在 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ 中没有成员（但 T^0 用于为*0*级函数的形式参数标注类型，例如 x 在 $\lambda x.$**返回**0中）。

由于分层较为微妙，最好在介绍完类型规则（§3.1.1）、核心语言（§3.2）和 elaboration（§3.3）之后再回顾。

精选 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ 类型规则汇总于 图 4。与 Xie 等人 [2023], 类似，类型判断按三个编译器模式之一进行索引： **编译**(c)、**引用** (q)，或 **拼接** (s)。然而，与 Xie 等人 [2023], 不同，类型判断无需按层级索引：由于 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ 是一个双层系统，每个编译器模式都唯一确定一个层级（$\mathbf{c}|\mathbf{q} \mapsto 0$、$\mathbf{s} \mapsto -1$）。每种模式有三种类型判断，分别对应一种语法类别（表 2）。

模式对 elaboration 很有用。**c** 表示代码是**环境式**且**惰性的**(没有外围的引号或拼接)。**s** 表示代码**会在编译时操作抽象语法树**(最后一个外围注释是拼接)。**q** 表示代码的**抽象语法树会在编译时被操作** (最后一个外围注释是引号)。因此，顶层拼接会从 **c** 转换为**s**。引用会从 **s** 转换为 **q**。拼接 ($e) 会从 **q** 转换为 **s**。图 3 用模式标注了一个元程序（其求值结果是 $\lambda x.1 + 2 + 3$的抽象语法树）。

对于 **c** 和**q** ，几乎所有情况下的类型判断都相同。为避免重复，我们引入记号 $\Gamma \vdash_{\mathbf{c}|\mathbf{q}} e : T$，用来表示两个判断 $\Gamma \vdash_{\mathbf{c}} e : T$和 $\Gamma \vdash_{\mathbf{q}} e : T$。结论的模式将与假设的模式一致，除非另有说明。

这些 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ 类型规则大多是带效应处理器的演算中的标准规则（参见附录 C）。在 **c** 和 **q** 模式下，编译时效应 Δ 会贯穿类型判断，而且只能访问上下文中的 0 级0变量。在 **s** 模式下，只能访问 -1 级变量。由于大多数情况下可以推断类型层级，为了便于阅读，通常也会省略它们。三个关键规则是 **s**-引用、**q**-拼接和 **c**-拼接，它们用于在模式和层级之间切换。

闭合 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ 表达式在 **c** 模式下，若能以空的编译时和运行时效应集进行类型判定，则为良类型：所有效应均可证明已在编译时和运行时得到处理。

定义 3.1（良类型闭合表达式）。闭合表达式 e 是良类型的，当且仅当 $\cdot \vdash_{\mathbf{c}} e : T^0 \,!\, \emptyset;\ \emptyset$

3.2 核心语言： $\lambda_{\mathsf{AST(op)}}$

$\lambda_{\mathsf{AST(op)}}$ (图 5) 是一种提供 AST 构造器和效应处理器的语言。语法分为范式、项和处理器。$\lambda_{\mathsf{AST(op)}}$ 将标准的效应处理器演算（λ_{op}）与 AST 构造机制以及用于作用域外逸检查的原语相结合。

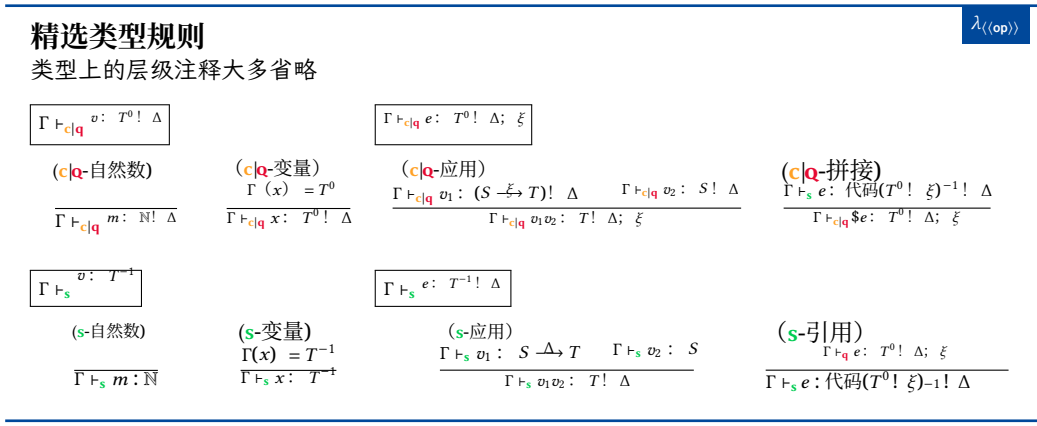


图 4. $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$的精选类型规则。

$\lambda_{\mathsf{AST(op)}}$ 的 AST 构造机制包括一个 AST 节点，对应用户可以编写的每种 λ_{op} 项构造形式（例如，表示变量的 Var、表示 _ 的 Hop；op(_,_) ↦ _），以及带类型注释的形式参数（α_R，其中 R 是某种运行时值预类型（图 5），以下简称“类型”）。形式参数表示绑定位置，例如 x 在 $\lambda x.$ return 0 中。将抽象语法树与形式参数分开，体现了 Calcagno 等人采用的方法 [2003]，，不过他们使用的是无类型形式参数。此外，$\lambda_{\mathsf{AST(op)}}$ 还添加了 mkvar R，这是一种用于生成类型为 R, α_R 的新鲜形式参数的原语；对 mkvar 的不同调用会返回不同的形式参数 [Taha 1999]。

$\lambda_{\mathsf{AST(op)}}$的作用域外逸检查机制包括：

- **err**，一种用于指示存在作用域外逸的错误状态，
- **检查** 以及 **检查**M，受保护的 **返回**s，它们要么报告作用域外逸，要么正常**返回**，
- **dlet**，一种用于跟踪哪些变量处于有效作用域内、哪些变量的作用域已外逸的原语，以及
- **tls**，一个标记，表示源程序中出现了顶层拼接：此时，剩余的栈帧要么引入新的顶层拼接，要么以完全直接明了的方式构造抽象语法树，并采用标准（因而安全）的控制流。

请注意，尽管该演算提供了用于作用域外逸检查的机制，但它并不要求使用该机制，也不要求正确使用它。作用域外逸检查并非语言特性，而是在该演算之上构建的一种算法。

3.2.1 操作语义。$\lambda_{\mathsf{AST(op)}}$ 的操作语义定义在配置 $t; E; U; M; I$ 上。从高层来看，t是项，而 E是求值上下文，定义为由求值框架组成的栈，类似于 Felleisen 等人 [1988] 的做法。U 充当新名称源。M 是静默变量的集合，即使这些变量的作用域已外逸，它们也不会触发作用域外泄错误。I 表示应解除 M 中变量静默状态的时机，方法是将 M 设为 $\emptyset$。总体而言，M和 I决定是立即执行检查（$M = \emptyset$），还是将其推迟到稍后的时刻（由 I标记）。当存在之后可能用于从作用域外逸中恢复的延续时，推迟检查的做法被 C4C 检查采用，因此它是“延续感知”的。惰性检查和急切检查的语义可以更简单地用三元组转换系统给出；这些系统是用于比较三种检查的五元组系统的直接投影。

带效应处理器演算的操作语义大体上符合预期。有趣的规则汇总于图 5，完整规则见附录 D。

语法 形式参数 α_R**范式** $n ::= x \mid m \in \mathbb{N} \mid \lambda x.t \mid \kappa x.t \mid$ **自然数**$(m) \mid \alpha_R \mid$ **变量**$(\alpha_R) \mid$ **Lam**$(n_1, n_2) \mid$ **应用**$(n_1, n_2) \mid$ **继续**$(n_1, n_2) \mid$ **Ret**$(n) \mid$ **执行**$(n_1, n_2, n_3) \mid$ **操作**$(n) \mid$ **Hwith**$(n_1, n_2) \mid$ **Hret**$(n_1, n_2) \; (n_1, n_2) \mid$ **Hop**(n_1, n_2, n_3, n_4) 项 $t ::= n_1 \; n_2 \mid$ **返回** $n \mid$ **执行** $x \leftarrow t_1$ **于** $t_2 \mid$ **操作**$(t) \mid$ **处理** n **与** $\{h\} \mid$ **继续** $n_1 \; n_2 \mid$ **check** $n \mid$ **Check**$_M \; n \mid$ **mkvar** $R \mid$ **dlet**$(n, t) \mid$ **tls**$(t) \mid$ **err** **处理器** $h ::=$ **返回**$(x) \mapsto t \mid$ **操作**$(x, k) \mapsto t$

类型上下文

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

类型

运行时预类型

效应集 $\xi ::=$	$\emptyset \mid \xi \cup \{\mathrm{op}_i\}$	效应集	$\Delta ::= \emptyset \mid \Delta \cup \{$操作$_i\}$
值类型 $Q, R ::=$	$\mathbb{N} \mid Q \xrightarrow{\xi} R$	值类型 $S, \; T ::=$	$\mathbb{N} \mid S \xrightarrow{\Delta} T \mid S \xrightarrow{\Delta} T$
	$\mid Q \xrightarrow{\xi} R$		|FParam(R) |抽象语法树(R)
计算类型	$R \,!\, \xi$		|抽象语法树$(R \,!\, \xi)$ |抽象语法树$(Q \,!\, \xi_1 \Longrightarrow R \,!\, \xi_2)$
处理器类型	$Q \,!\, \xi_1 \Longrightarrow R \,!\, \xi_2$	计算类型	$T \,!\, \Delta$
		处理器类型	$S \,!\, \Delta \Longrightarrow T \,!\, \Delta_2$

操作语义

选定规则

辅助定义

求值框架	F	::=	**执行** $x \leftarrow [-]$ **于** $t_2 \mid$ **处理** $[-]$**，使用** $\{h\} \mid$ **DLet**$(\alpha_R, \; [-]) \mid$ **tls**$([-])$
求值上下文	E	::=	$[-] \mid E[F]$
处理器的定义域	定义域(h)	$\triangleq$	定义域(**返回**$(x) \mapsto t) = \emptyset$, 定义域$(h;$ **操作**$(x, \; k) \mapsto t) =$ 定义域$(h) \cup \{$**操作**$\}$
已处理效应	已处理(E)	$\triangleq$	已处理$([-]) = \emptyset$, 已处理$(E[$**执行** $x \leftarrow [-]$ **于** $t_2]) =$已处理(E), 已处理$(E[$**处理** $[-]$ **并使用** $\{h\}])$ = 已处理$(E) \setminus$dom(h), 已处理$(E[$**dlet**$(\alpha_R, \; [-])])$ = 已处理(E), 已处理$(E[$**tls**$([-])])$ = 已处理(E)

归约规则

与静音和取消静音相关的机制已突出显示

(Ast-Gen)
mkvar $R; \; E; U; \; M; \; I \;\; \rightarrow \;\; \langle$**返回** $\alpha_R; \; E; U \cup \{\alpha\}; \; M; \; I\rangle$
其中 $\alpha =$ 下一个（U），下一个（U） $\notin U$，下一个 确定性

(Sec-Tls)
$\langle$**tls**(**返回** n)$; \; E; U; \; M; I\rangle \;\; \rightarrow \;\; \langle$**返回** $n; E; U; \; \emptyset; \; \top\rangle$

(Sec-Chs)
检查 $n; \; E; U; \; M; I \;\; \rightarrow \;\; \langle$**返回** $n; \; E; \; U; \; M; \; I\rangle$
如果 $FV^0(n) \subseteq \pi_{\mathsf{Var}}(E)$

(Sec-Cms)
check$_M \; n; \; E; U; \; M; I \;\; \rightarrow \;\; \langle$**返回** $n; \; E; U; \; M; I\rangle$
如果 $FV^0(n) \setminus M \subseteq \pi_{\mathsf{Var}}(E)$

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

(Sec-Cmf)
check$_M \; n; \; E; U; \; M; I \;\; \rightarrow \;\;$ **err**$; \; E; U; M; I$
如果 $FV^0(n) \setminus M \not\subseteq \pi_{\mathsf{Var}}(E)$

(Sec-Dlt)
$\langle$**DLet**$(\alpha_R,$ **返回** $n); \; E; U; \; M; I\rangle \;\; \rightarrow \;\; \langle$**返回** $n; \; E; U; \; M'; I'\rangle$
如果 $len(E) > I$ 则 $M' = M, I' = I,$ 否则 $M' = \emptyset, I' = \top$

(Eff-Op)
$\langle$**操作**$(v); \;\; E_1[$**处理** E_2 **及** $\{h\}]; \;\; U; \;\; M; \;\; I\rangle \;\; \rightarrow \;\; \langle c[v/x, \; \mathrm{cont}/k]; \;\; E_1; \;\; U; \;\; M \cup \pi_{\mathsf{Var}}(E_2); \;\; I'\rangle$
其中 $cont = \kappa x.$ **处理** $E_2[$**返回** $x]$ **并** $\{h\}$ 以及 **操作**$(x, k) \mapsto c \in h$ 以及 操作 $\notin$ 已处理(E_2) 以及 $I' =$ **最小值**(**长度**$(E_1) \; I)$

图 5. $\lambda_{\mathsf{AST(op)}}$：语法、类型和操作语义。

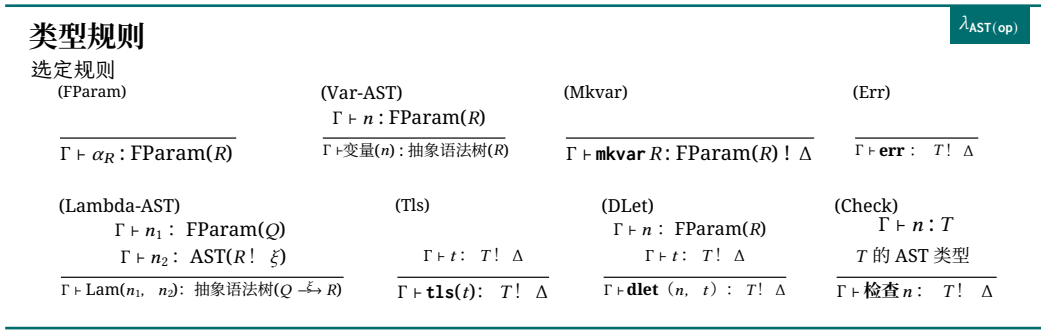


图 6. 部分 $\lambda_{\mathsf{AST}(\mathsf{op})}$ 类型规则

在 Ast-Gen 规则中，U 通过记录先前生成的名称来确保新鲜性。为了确保语义的确定性，新鲜名称由某种（未指定的）确定性过程选取。

check 原语的作用类似于受保护的返回。对于某个任意的 AST 类型范式 n，如果 n 的所有自由变量都处于正确的作用域内，那么 check n 会化简为 return n（Sec-Chs）；否则，如果 n 的某些自由变量不处于正确的作用域内，那么 check n 会化简为 err（Sec-Chf）。按照 Kiselyov [2024的做法，], dlet 声明变量处于正确的作用域内，方法是在求值上下文 E中放置一个形式为 $\mathsf{dlet}(\alpha_R, [-])$ 的框架。记号 $\pi_{\mathsf{Var}}(E)$会从 E中过滤掉以这种方式声明的变量。例如，$\pi_{\mathsf{Var}}(\mathsf{dlet}(\alpha_R, \mathsf{do}\ x \leftarrow [-]\ \mathsf{in}\ t)) = \{\mathsf{Var}(\alpha_R)\}$。给定项 $E[t]$，如果 t 中的 $\mathsf{Var}(\alpha_R)$ 满足 $\mathsf{Var}(\alpha_R) \in \pi_{\mathsf{Var}}(E)$，那么 $\mathsf{Var}(\alpha_R)$ 在 E 中就是“声明安全”的（定义 3.2）。

定义 3.2（声明安全）。给定项 $E[t]$，$\mathsf{Var}(\alpha_R)$ 在 t 中**声明安全**在 E 当且仅当$\mathsf{Var}(\alpha_R) \in \pi_{\mathsf{Var}}(E)$

给定求值上下文 E 中的范式 n，其中 n 是一棵抽象语法树，当且仅当 n 的自由变量（记作 $\mathsf{FV}^0(n)$）在 E 中都已声明为 Safe，即 $\mathsf{FV}^0(n) \subseteq \pi_{\mathsf{Var}}(E)$ 时，n 在 E 中作用域正确（也就是说，**Check** n 成功）。由于 $\lambda_{\mathsf{AST}(\mathsf{op})}$ 是 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ 的精化目标，因此由 elaboration 负责恰当地使用 **DLet** 和 **Check**。

该$\mathbf{Check}_{\mathsf{M}}$ 结构是 **Check** 的一种变体。如§4.4所述，$\mathbf{Check}_{\mathsf{M}}$ 还会忽略一些静音的变量，并将它们视为作用域正确（$\mathbf{Check}_{\mathsf{M}}\ n$ 当 $\mathsf{FV}^0(n) \setminus M \subseteq \pi_{\mathsf{Var}}(E)$) 时成功。

Sec-Tls、Sec-Dlt和Eff-Op 会对变量进行静音或取消静音。§4.4 介绍了静音和取消静音。忽略静音和取消静音，Sec-Tls 和Sec-Dlt 会分别静默移除一个$\mathbf{tls}([-])$ 和$\mathbf{dlet}(\alpha_R, [-])$框架，而Eff-Op 则赋予处理器预期的标准行为。

3.2.2 类型系统。 $\lambda_{\mathsf{AST}(\mathsf{op})}$ 类型大多是标准的。关键新增内容是一个 FParam 类型，用于形式参数；以及一个 抽象语法树类型，用于抽象语法树（图 5）。

这些 $\lambda_{\mathsf{AST}(\mathsf{op})}$ 类型规则（图 6）极其直观。根据这些类型规则，良类型的抽象语法树也可能是作用域不正确；例如，$\cdot \vdash$ 变量$(\alpha_R) : $抽象语法树$(R)$ 是一个有效的类型判断。作用域外溢检查实际上对类型系统不可见。唯一复杂的情况是**err**，它可以在任意上下文中被赋予任意类型，这与**abort** [Scherer 2017]类似。

闭合 $\lambda_{\mathsf{AST}(\mathsf{op})}$ 项若可在空效应集下进行类型判断，则称为良类型。

D定义 3.3（良类型闭项）。闭合项 t 若满足 $\cdot \vdash t :\ T ! \emptyset$

3.3 从 $\lambda_{\langle\langle \mathbf{op} \rangle\rangle}$ 到 $\lambda_{\mathsf{AST}(\mathbf{op})}$ 的 elaboration

本节介绍从 $[\![-]\!]$到 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ 的 $\lambda_{\mathsf{AST}(\mathsf{op})}$细化。这一细化很简单：它不会插入任何动态作用域外逸检查。§4中的其他细化会插入检查，并扩展这一细化。

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

类型细化

选定规则

$$
\begin{array}{lcl}
[\![T^0]\!] & = & \text{抽象语法树(擦除}(T^0)\text{)} \\
[\![T^0 \,!\, \xi]\!] & = & \text{抽象语法树(擦除}(T^0 \,!\, \xi)\text{)} \\
[\![T^0 \,!\, \Delta]\!] & = & \text{抽象语法树(擦除}(T^0)\text{)} \,!\, [\![\Delta]\!] \\
[\![T^0 \,!\, \Delta;\ \xi]\!] & = & \text{抽象语法树(擦除}(T^0 \,!\, \xi)\text{)} \,!\, [\![\Delta]\!]
\end{array}
\qquad
\begin{array}{lcl}
[\![\mathbb{N}^{-1}]\!] & = & \mathbb{N} \\
[\![(S^{-1} \xrightarrow{\Delta} T^{-1})^{-1}]\!] & = & [\![S^{-1}]\!] \overset{[\![\Delta]\!]}{\Rightarrow} [\![T^{-1}]\!] \\
[\![(S^{-1} \xrightarrow{\Delta} T^{-1})^{-1}]\!] & = & [\![S^{-1}]\!] \overset{[\![\Delta]\!]}{\Rightarrow} [\![T^{-1}]\!] \\
[\![\text{代码}(T^0 \,!\, \xi)^{-1}]\!] & = & \text{抽象语法树(擦除}(T^0 \,!\, \xi)\text{)}
\end{array}
$$

上下文条目展开

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

项 elaboration

选定规则（抽象语法树）

$$
\begin{array}{lcl}
[\![x]\!]_{\mathbf{c}|\mathbf{q}} & = & \textbf{返回}\ \text{变量}\ (x) \\
[\![\lambda x\colon\ T^0.\, e]\!]_{\mathbf{c}|\mathbf{q}} & = & \textbf{执行}\ x \leftarrow \textbf{mkvar}\ \text{擦除}\ (T^0)\ \text{于} \\
 & & \textbf{执行}\ \mathsf{body} \leftarrow [\![e]\!]_{\mathbf{c}|\mathbf{q}}\ \textbf{中返回}\ \mathsf{Lam}(x, \mathsf{body})
\end{array}
\qquad
\begin{array}{lcl}
[\![x]\!]_{\mathbf{s}} & = & x \\
[\![\lambda x\colon\ T^0.\, e]\!]_{\mathbf{s}} & = & \lambda x.[\![e]\!]_{\mathbf{s}}
\end{array}
$$

选定规则（引用/拼接）

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

图 7. 从 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ 到 $\lambda_{\mathsf{AST(op)}}$ 的选定推导规则。

该细化定义在类型判断上：$\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ 类型判断细化为 $\lambda_{\mathsf{AST(op)}}$ 类型判断。这可以分解为四种细化：对效应集、类型、上下文和项的细化。

3.3.1 推导效应集合和类型。 效应集的推导是恒等变换。为了定义类型的推导（图 7），可以方便地引用一个辅助函数 erase（附录 G）。给定一个层级为 0 的类型，erase 会擦除所有层级注释（并推导效应集），例如 $\mathsf{erase}((S^0 \xrightarrow{\xi} T^0)^0) = S \overset{[\![\xi]\!]}{\Rightarrow} T$。简而言之，层级为 0 的类型会推导为 抽象语法树 类型，而层级为–1 的类型会推导为其自身（去除层级注释），但代码 类型除外，它们会推导为抽象语法树 类型。

3.3.2 推导上下文。 上下文的推导较为微妙（图 7）。上下文中的层级为 0 的类型会推导为 FParam，而不是抽象语法树 类型。因此，上下文的推导需要针对上下文条目进行单独的推导，不能简单地依赖类型的推导。要理解层级为 0 的类型为何会推导为 FParam 类型，请注意，只有在上下文 Γ 扩展一个层级为 0 的变量时，才会出现在 **c** 或 **q** 中。这些模式会构建抽象语法树，因此 x 必须是一个 FParam。

3.3.3 推导项。 项的 elaboration（图 7）假设所有形式参数都已标注其类型，例如 $\lambda x : \mathbb{N}^0.\, e$。项的 elaboration 受以下**模式**：**c**、**q**或 **s**的调节。在较高层次上，在**c**和 **q**-模式下，需要构造抽象语法树。为确保形式参数得到适当重命名，elaboration 必须使用 **`mkvar`**。

elaboration 在**c**和 **q**-模式之间没有显著差异，但拼接规则例外：在 **c** -模式下会插入 **tls**，而在 **q**-模式下不会插入。构建作用域外逸检查时，**c**和**q**-模式变得十分重要。在 **s**-模式下，elaboration 实际上就是恒等操作。

3.3.4 推导类型判定。 现在可以按组合方式定义类型判断的 elaboration。例如，**c**-模式下 Lambda 表达式的类型判断通过逐分量应用 elaboration 组件来推导：

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

令 $Q = \text{擦除}(S)$、$R = \text{擦除}(T)$ 和 $[\![e]\!]_c = t$，并应用上文定义的 elaboration 函数，我们得到 类型推导 1；假设前提是有效的类型推导，它对应于一个有效的 $\lambda_{\mathsf{AST}(\mathsf{op})}$ 类型推导。

$$\frac{[\![\Gamma]\!], x : \mathsf{FParam}(Q) \vdash t : \mathsf{AST}(R \,!\, \xi) \,!\, \Delta}{[\![\Gamma]\!] \vdash \textbf{执行}\ x \leftarrow \textbf{mkvar}\ \text{擦除}(T^0)\ \textbf{in}\ \textbf{执行}\ \text{主体}\ \leftarrow t\ \textbf{in}\ \textbf{返回}\ \mathsf{Lam}(x, \text{主体}) : \text{抽象语法树}(Q \xrightarrow{\xi} R) \,!\, \Delta}$$

类型推导 1。 $\Gamma \vdash_c \lambda x$ 的展开推导 $.e : S \xrightarrow{\xi} T$

3.4 元理论

类型正确的 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ 程序会细化为类型正确的 $\lambda_{\mathsf{AST}(\mathsf{op})}$ 程序：

定理 3.1（细化保持性）。若 $\Gamma \vdash_\star e : \tau$，则 $[\![\Gamma]\!] \vdash [\![e]\!]_\star : [\![\tau]\!]$，其中$\star = c \mid q \mid s$ 且 τ 是一个层级为0 或层级为 -1 的值、计算或处理器类型。

证明通过对类型规则进行归纳得出，例如§3.3.4中的类型推导 1。此外，核心语言 $\lambda_{\mathsf{AST}(\mathsf{op})}$ 具有进展性和保持性。

定理 3.2（进展）。若$\cdot \vdash E[t] : T \,!\, \Delta$，则对所有 U，M，I，以下情况之一成立：

(1) t 的形式为 return n，且 $E = [-]$；(2) t 的形式为 $\mathsf{op}(v)$，其中 $\mathsf{op} \in \Delta$，且操作 op 已处理E；(3) t 的形式为 err；(4) $\exists t'$、E'、U' M'、I' 满足 $t; E; U; M; I \longrightarrow t'; E'; U'; M'; I'$

请注意第三个子句，演算可用它来报告作用域外逸。

进展性证明通过对类型推导进行归纳得出。大多数情况都是标准的，Bauer 和 Pretnar [2014]已经证明过这一点。证明只需考虑抽象语法树构造和作用域外逸检查的类型规则，这些规则都很直接。

定理 3.3（归约保持性）。若 $\cdot \vdash E[t] : T \,!\, \Delta$且 $t; E; U; M; I \longrightarrow t'; E'; U'; M'; I'$，则 $\cdot \vdash E'[t'] : T \,!\, \Delta$

证明通过对操作语义进行归纳得出。同样，证明只需考虑抽象语法树构造和作用域外逸检查的规则，这些规则都很简单。

由此可推得类型安全这一概念。

推论 3.4（类型安全）。若 $\cdot \vdash_c e : T^0 \,!\, \emptyset; \emptyset$ 则以下情况之一成立

(1) $\langle [\![e]\!]_c; [-]; \emptyset; \emptyset; \top \rangle \longrightarrow^\omega$, *(2)* $\langle [\![e]\!]_c; [-]; \emptyset; \emptyset; \top \rangle \longrightarrow^* \langle err; E; U; M; I \rangle$，对于某些 E, U, M, I，或者 *(3)* $\langle [\![e]\!]_c; [-]; \emptyset; \emptyset; \top \rangle \longrightarrow^* \langle return\ n; [-]; U; M; I \rangle$，对于某些 U, M, I

其中，初始配置由一个精化项、空的求值上下文、一个表示此前未生成任何变量的空集合、另一个表示没有变量被静音的空集合，以及 $\top$ 组成，表示（目前）没有解除变量静音的计划。

重要的是，这种类型安全概念是弱的。根据此定义，始终报告作用域外逸错误（**err**）的语义是类型安全的，而从不报告作用域外逸的语义也是如此。由于可能出现作用域外逸，推论 3.4的第三种情况不能进一步断言范式 n 代表一个类型正确的 λ_{op} 程序。

最后，在顶层拼接之下，代码引述和拼接互为对偶。

定理 3.4（引文-拼接对偶性）。在顶层拼接之下，代码引述和拼接互为对偶：

$\$\langle\langle e \rangle\rangle =_q e$ $\langle\langle \$e \rangle\rangle =_s e$

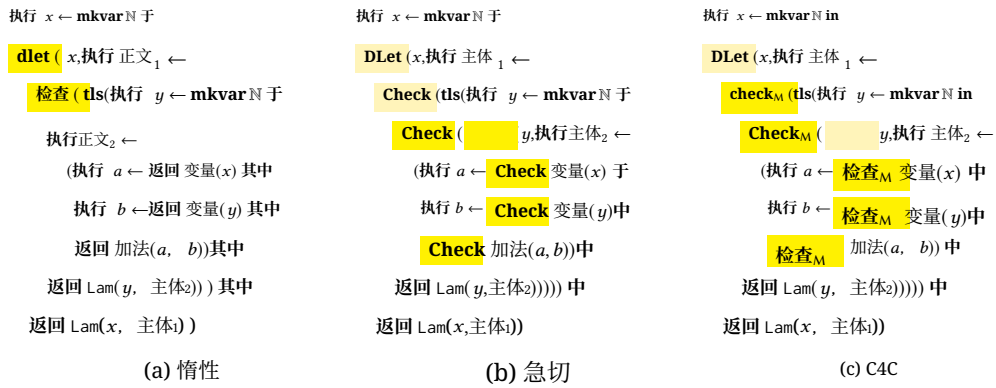


图 8. 在不同检查下 $\lambda x:\ \mathbb{N}.\ \$\lambda y:\ \mathbb{N}.\ x + y$ 的 elaboration

其中 $=_\star$ 表示“将 $\lambda_{\mathsf{AST}(\mathsf{op})}$ 中的程序 elaboration 为上下文等价的 ★模式程序”。必须按模式进行参数化，因为模式会影响 elaboration 的结果。还可以证明一个更强的结论：它们 elaboration 为相同的语法 $\lambda_{\mathsf{AST}(\mathsf{op})}$ 程序（上下文等价可由自反性推出）。定理 3.4 的证明通过检查 elaboration 的定义即可得出，其中：

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

4 动态作用域外逸检查

本节采用 $\lambda_{\langle\langle \mathsf{op}\rangle\rangle}$ 来给出作用域外逸（包括现有方法 [Isoda 等人 2024；Kiselyov 2014]）的精确定义，以及作用域外逸检查的性质。

4.1 动态作用域外逸检查的性质

由于检查被定义为项 elaboration，我们用 $[\![-]\!]^{\mathbf{Check}}$ 表示任意检查。我们将§3.3中的项 elaboration 称为朴素 elaboration。

给定一个将作用域外逸定义为关于配置的谓词 Φ，如果对于任何良类型 $\lambda_{\langle\langle \mathsf{op}\rangle\rangle}$ 表达式 e，其朴素 elaboration 化简为一个呈现作用域外逸的配置（$\Phi(t;\ E; U;\ M;\ I)$），则带有该 Check 的 elaboration 会化简为**正确**，那么这个 Check 就是正确的。作用域外溢检查的**宽容性**指的是这样一组良类型 $\lambda_{\langle\langle \mathsf{op}\rangle\rangle}$ 表达式：即使它们呈现作用域外逸，其 elaboration 也不会化简为**err**。

定义 *4.1*（动态作用域外逸检查的正确性）。给定关于配置的谓词 Φ，如果对于所有封闭的、良类型的 $\lambda_{\langle\langle \mathsf{op}\rangle\rangle}$ 表达式 e，$\langle [\![e]\!];\ [-];\ \emptyset;\ \emptyset;\ \top\rangle \to^* \langle t;\ E; U;\ M;\ I\rangle \wedge \Phi(\langle t;\ E;\ U;\ M;\ I\rangle) \implies \langle [\![e]\!]^{\mathit{Check}};\ [-];\ \emptyset;\ \emptyset;\ \top\rangle \to^* \langle \mathit{err};\ E'; U';\ M';\ I'\rangle$（对于某些 $E', U' M', I'$），那么动态作用域外逸检查 $[\![-]\!]^{\mathit{Check}}$ 相对于 Φ 是正确的。

定义 4.2（动态作用域外逸检查的宽容性）。令 良类型为封闭的良类型 $\lambda_{\langle\langle \mathsf{op}\rangle\rangle}$ 表达式集合。动态作用域外逸检查的**宽容性**定义为$\{e \in \text{良类型} \mid \langle [\![e]\!]^{\mathit{Check}};\ [-];\ \emptyset;\ \emptyset;\ \top\rangle \not\to^* \langle \mathbf{err};\ E;\ U;\ M;\ I\ \rangle\}$

4.2 惰性检查

A $\lambda_{\mathsf{AST}(\mathsf{op})}$ 配置呈现**惰性作用域外逸**，当它是编译时执行的结果且作用域不正确时。本文将 Kiselyov 提出的定义形式化。Kiselyov [2014]。

定义 4.3（惰性作用域外逸）。$A\ \lambda_{\mathsf{AST(op)}}$形式为 $\langle t;\ E;\ U;\ M;\ I\rangle$的配置，在满足以下条件时呈现**惰性作用域外逸**：$t =$ **返回** n 某个 *AST* 类型的 n，$E = E'[\mathtt{tls}([-])]$ 某个E'，且 $FV^0(n) \not\subseteq \pi_{\mathsf{Var}}(E)$。

惰性检查，$[\![-]\!]^{\mathbf{Lazy}}$，通过两种方式扩展朴素 elaboration（图 8a）。首先，在顶层拼接之后执行**检查**：$([\![\$e]\!]^{\mathbf{Lazy}}_{\mathbf{c}} \triangleq \textbf{检查}(\mathtt{tls}([\![e]\!]^{\mathbf{Lazy}}_{\mathbf{s}}))$。其次，插入**dlet**，以确保在顶层拼接之外绑定的变量（处于**c**模式下）在顶层拼接周围的上下文中声明为安全（定义 3.2）。在**c**模式（但非**q**模式）下对形式参数进行 elaboration 时，应插入**dlet**：

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

由于该算法十分简单，验证惰性检查的正确性（就惰性作用域外逸而言）和宽容性是轻而易举的：当且仅当朴素 elaboration 在归约后会出现惰性作用域外逸时，惰性检查才会检测到作用域外逸。

定理 *4.1*（惰性检查的正确性与许可性）。对于所有封闭且类型正确的$\lambda_{\langle\langle \mathsf{op}\rangle\rangle}$程序 e，$\langle [\![e]\!]^{\mathit{Lazy}};\ [-];\ \emptyset;\ \emptyset;\ \top\rangle \rightarrow^* \langle \mathit{err};\ E; U;\ M;\ I\rangle \Longleftrightarrow$ 对于某些 $E', U'\ M', I'$，$\langle [\![e]\!];\ [-];\ \emptyset;\ \emptyset;\ \top\rangle \rightarrow^* \langle$返回 $n;\ E'; U';\ M';\ I'\rangle$，并且 $\langle$返回 $n;\ E'; U';\ M';\ I'\rangle$ 会出现惰性作用域外逸

因此，惰性检查刻画了允许执行的 Safe $\lambda_{\langle\langle \mathsf{op}\rangle\rangle}$ 程序集合。该集合用于定义检查的表达能力，而惰性检查是最具表达能力的：

定义 *4.4*（动态作用域外逸检查的表达能力）。定义集合 $\mathit{Safe} \triangleq \{e \in$ 良类型 $\mid \langle [\![e]\!]^{\mathit{Lazy}};\ [-];\ \emptyset;\ \emptyset;\ \top\rangle \not\rightarrow^* \langle \mathit{err};\ E; U;\ M;\ I\rangle\}$。则动态作用域外逸检查的表达能力定义为 $\{e \in \mathit{Safe} \mid \langle [\![e]\!]^{\mathit{Check}};\ [-];\ \emptyset;\ \emptyset;\ \top\rangle \not\rightarrow^* \langle \mathit{err};\ E; U;\ M;\ I\rangle\}$

给定一个作用域外溢检查，所有被拒绝但惰性检查会允许的程序都被视为假阳性：

定义 4.5（动态作用域外逸检查的假阳性）。动态作用域外逸检查的**假阳性**定义如下：$\{e \in \mathit{Safe} \mid [\![e]\!]^{\mathit{Check}};\ [-];\ \emptyset;\ \emptyset; \top \rightarrow^* \mathbf{err};\ E; U;\ M;\ I\}$

然而，同样由于其简单性，惰性检查被认为不适合实际使用。Ofenbeck 等人 [2016], 使用惰性检查的他们报告如下：

> 我们实现中的错误……可能会表现为以下错误：
>
> 前向引用超出了值 x1620[error]valx1343 = x1232(x1123, x1124, x1180, x1181, x1223, x1224, x1223, x1229, x1216, x1120, x1122, x1121) 的定义范围
>
> ... *[在遇到这个错误之前，已经处理了]*一大段代码……此类错误的根本原因往往非常简单，却在其显现的代码中被严重掩盖。

惰性检查无法提供有用信息：由于它要等到求值结束才执行，因此错误指向的是生成代码，而不是生成程序 [Kiselyov 2014]。这种混淆使所有程序的调试都变得困难。此外，惰性检查必须等到求值结束后才能报告错误。这使得调试大型分阶段程序时效率低下，例如 Ofenbeck 等人生成的程序。此外，Kameyama 等人 [2015, §4。1]指出，在某些系统中，惰性检查可能导致变量被意外绑定。

4.3 急切检查

如果某个配置在执行过程的任意时刻呈现**急切作用域外逸**，即它**返回**一棵作用域不当的抽象语法树。定义 4.6 因此推广了定义 4.3。

定义 4.6（急切作用域外逸）。$\lambda_{\mathsf{AST(op)}}$形式为 $\langle t;\ E;\ U;\ M;\ I\rangle$的配置若满足：对于某种 *AST* 类型，$t = \textbf{返回}\ n\ n$，且 $FV^0(n) \not\subseteq \pi_{\mathsf{Var}}(E)$

可以通过扩展惰性检查来定义急切检查（图 8b）。除了顶层拼接**检查**和 c-模式的 **dlet**之外，急切检查还会为以 **q**-模式构造的抽象语法树添加**检查**，例如：

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

不同于朴素 elaboration 和惰性检查，急切检查会生成 **检查** 应用$(f,\ a)$，而不是 **返回** 应用$(f,\ a)$。因此，为了防止假阳性，在 **q**-模式下绑定的变量必须生成 **dlet**：

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

直观地说，急切检查会在每次构造抽象语法树时执行检查。因此，假设求值规约到一个表现出急切作用域外逸的配置。设违规的抽象语法树为 n。当在某个求值上下文 E中使用 n 来构造更大的抽象语法树 n'，且 n' 中的并非所有自由变量都在 E(清单 7) 中被声明安全时，错误就会被检测并报告。Kiselyov [2014] 指出，在抽象语法树构造时执行检查所产生的开销可以忽略不计。

```
$(执行 z ← (处理 λx.$( 急切检查(x) ) with {返回(u) ↦
0;急切检查(y, k) ↦ 返回 y})in$z + 1)
```

清单 7。 当在未声明 Var($x_{\mathbb{N}}$) 的上下文中使用 z $z + 1时，会报告作用域外逸 fe

急切检查模拟了 Kiselyov [2024b] 所描述的 BER MetaOCaml Check。可以通过执行配套制品中的 BER MetaOCaml N153 翻译来验证该模型： 代码清单 7 至 10 [Lee 等人 2025]。

4.3.1 急切检查的正确性。急切检查对于急切作用域外逸而言并不正确。求值可能导致从未被检测到的急切作用域外逸。例如，违规的抽象语法树可能会被丢弃（清单 8）。

```
$(处理 λx.$( (x) )
  与{返回(u) ↦ 0;操作(y, k) ↦ 执行 w ← 返回 y 于0})
```

清单 8。 当违规的抽象语法树被丢弃时，急切检查不会报告急切作用域外逸 .

急切检查的一个显著特性是，它允许程序通过恢复一个延续来从作用域外逸中恢复。在清单 9 中，程序恢复捕获的求值上下文，该上下文声明 Var($x_{\mathbb{N}}$) safe。只有这样，才会使用 Var($x_{\mathbb{N}}$) 来构建抽象语法树，因此检查得以通过。

```
$(处理 λx.返回 $( 操作(x) )
  与 {返回(u) ↦ 返回 u;操作(y, k) ↦ 执行 u ← 返回 y 在继续 k u})
```

清单 9。急切检查不会报告违规的抽象语法树仅以 Safe 方式使用的情况。

急切检查不正确（即它未能报告所有急切作用域外逸）的原因自然源于这些定义。Kiselyov [2014] 将急切作用域外逸定义为在求值过程中的任意时刻出现自由变量。急切 *Check*则不同，只有在使用自由变量时才会调用，例如执行自由变量或用它来构造更大的代码片段。然而，急切检查的不正确性有时反而是可取的。由于代码清单 8和9均属于 Safe，宽容性使急切检查更具表达力。

处理 do $x_{\mathbb{N}}$ **←mkvar** $\mathbb{N}$ **in check(dlet(**$x_{\mathbb{N}}$**, do body ←(do** a **←变量(**$x_{\mathbb{N}}$**) in 操作(**a**)) in 返回 Lam(**$x_{\mathbb{N}}$**, body)) with {返回(**u**)** $\mapsto$ **返回** u**; 操作(**y, k**)** $\mapsto$ **do** w **←check 加法(**y**, 自然数(0)) in 继续** k w**}**

图9. 对列表 10 使用急切检查进行展开的结果

4.3.2 急切检查的表达能力。 存在一等延续时，急切检查并非 具有最大表达能力。它会报告假阳性，例如 列表 10。

```
$(处理  λx.$( 操作(x) )
  与 {返回(u)  ↦ 返回  u;操作(y,  k)  ↦ 执行  u ← $y + 0 在继续 k u})
```

列表 10. 假阳性：一个安全的程序未通过急切检查。

在 列表 10中，有问题的抽象语法树 (变量($x_{\mathbb{N}}$)) 位于一个上下文中，而其中的 变量($x_{\mathbb{N}}$) 未声明安全，因此急切检查会报告错误。然而，如果允许求值继续进行，求值上下文中的绑定 变量($x_{\mathbb{N}}$) 及其安全声明就会恢复，所有变量也都会正确地限定在各自的作用域内。

比较 列表 9（通过急切检查）与 列表 10（未通过该检查）可见，这项检查的结果难以预测：若不参考操作语义，就难以描述其表达能力。遗憾的是，$\langle\langle \$e \rangle\rangle \neq_s \langle\langle \$e + 0 \rangle\rangle$。更一般地，对于程序片段 P和 P'， $P[e] =_s P'[e] \not\Longrightarrow \langle\langle P[\$\langle\langle e \rangle\rangle] \rangle\rangle =_s \langle\langle P'[\$\langle\langle e \rangle\rangle] \rangle\rangle$。

这种不可预测性源于急切检查的设计。 Kiselyov [2014, 脚注 10]指出，在存在一等延续的情况下，急切检查可能会报告假阳性，但他在实践中并未观察到此类情况。我们称急切检查不具备 延续感知能力。

4.4 值得关注（C4C）检查

如果惰性检查过于不切实际，而急切检查又过于不可预测，那么有没有可能找到一种“恰到好处”的方案？这样的检查应允许 列表 10中的程序，并以可预测的方式保持宽松。当某个配置 **不可避免地发生作用域外逸** 时，必然 会导致惰性作用域外逸。

定义 *4.7*（不可避免的作用域外逸）。形式为 $\lambda_{\mathsf{AST(op)}}$配置 $t; E; U; M; I$的配置，如果$\langle t; E; U; M; I \rangle \rightarrow^* \langle t'; E'; U'; M'; I' \rangle$ 和 $\langle t'; E'; U'; M'; I' \rangle$ 表现出惰性作用域外逸，则该配置表现出不可避免的作用域外逸。

本节介绍一种值得关注（C4C）检查，用于近似判断不可避免的作用域外逸，但会产生假阳性。C4C 检查的 elaboration 与急切检查的 elaboration 略有不同，其中 **检查**$_{\mathsf{M}}$ 取代**检查** (图 8c)。例如，

$[\![\lambda x\colon\ T^0.\, e]\!]^{\mathbf{BE}}_{\mathbf{q}} = \mathbf{执行}\ \ x \leftarrow \mathbf{mkvar}\ \text{擦除}(T^0)\ \mathbf{在检查}_{\mathsf{M}}(\mathbf{dlet}(x,\ \mathbf{执行}\ \text{主体}\ \leftarrow [\![e]\!]^{\mathbf{BE}}_{\mathbf{q}}\ \mathbf{在返回}\ \mathsf{Lam}(x,\ \text{主体})))$

要理解 C4C 检查，请看图 9，其中列表 10 使用急切检查进行 elaboration，得到 $\lambda_{\mathsf{AST(op)}}$，并为便于阅读而简化（例如，使用**检查** t而不是**执行** $x \leftarrow t$**在检查** x）。未通过的检查已加下划线。

检查失败，因为执行**操作** 时，变量 Var($x_{\mathbb{N}}$) 在新的求值上下文中已不再声明安全。由于 y绑定到 Var($x_{\mathbb{N}}$)，检查 Plus(y， Nat(0)) 时会报错。问题在于，延续 k 可用于绑定 Var($x_{\mathbb{N}}$)。当构造并检查

```
checkM(dlet(zℕ, 执行 b ←
    (执行 f ←正文
     在执行 a ← 返回 自然数(1)
     在 checkM 应用(f， a))
   在 返回 Lam(zℕ, b)))
```

(a) 初始项

```
检查M(dlet(zℕ, 执行 b ←
    (执行 f ← [返回 Lam(xℕ, 加法(变量(xℕ), 自然数(0)))
     在执行 a ← 返回 自然数(1)
     在 CheckM 应用(f， a))
   在返回 Lam(zℕ， b)))
```

（b）归约项

图 10。（a）生成 $\lambda z.(\lambda x.x+0)(1)$ 的抽象语法树的 $\lambda_{\mathsf{AST(op)}}$ 程序。（b）将（a）中的程序归约到可以取消变量静音这一程度后的结果。

Plus 抽象语法树时，并不清楚急切作用域外逸必须 导致惰性作用域外逸。因此，为了让检查更具表达力，暂时允许 $\mathrm{Var}(x_{\mathbb{N}})$ 外逸其作用域，并将错误检测延迟到必须进行惰性作用域外逸时，或许会更有用。

该 $\mathbf{Check}_{\mathsf{M}}$ 会检查作用域外逸，但允许一组静默变量 M 暂时外逸其作用域。在我们的示例中，我们可以将 变量($x_{\mathbb{N}}$)静音，方法是将其添加到 M中。$\lambda_{\mathsf{AST(op)}}$ 操作语义（图 5）会自动执行此过程，在关键点策略性地对变量进行静音和取消静音：

- 当执行效应时，在新的求值上下文中不再声明安全的变量（例如 $\mathrm{Var}(x_{\mathbb{N}})$）会被添加到静默变量集合中（Eff-Op）。
- 当不存在有界延续时，变量会解除静音，因此也就无法恢复可能绑定 $\mathrm{Var}(x_{\mathbb{N}})$ 的延续 k。通过追踪栈 E 中从未被效应处理捕获的最大长度，可以确定这一点 I。

直观来说， I 是一个栈标记（或延续标记），用于追踪效应与异常变得无法区分的位置。C4C 检查在此位置之前的行为类似于惰性检查，在此位置之后则类似于急切检查。急切检查的实现 [Kiselyov 2014，附录 B], 以及带有延续的语言的语义和实现 [Flatt 和 Dybvig 2020；Kiselyov 2012]中都使用了栈标记。不过，由于急切检查并非延续感知，栈标记的作用有限。

例如，图 10a中的程序构建了 $\lambda z.(\lambda x.x+0)(1)$的抽象语法树。令body 为图 9中的程序。围绕 body 的上下文由 I标识：它从未被任何效应的处理过程捕获，因此不得包含对捕获的延续的引用 k。

如果栈从未因处理效应而被捕获（例如，未执行任何操作），则 I会被设置为 $\top$，$\forall n \in \mathbb{N},\ \top \geq n$。执行效应可能会减少 I，但绝不会增加它。这是 Eff-Op 的旁条件。

在归约过程中，当栈的长度小于或等于 I时，不得再存在对任何延续的引用 k，因此 I 可以被重置为 $\top$，并且所有静默变量都可以取消静默。图 Figure10a 中的程序最终归约为 Figure10b中的项。 $[\ -\ ]$用于分隔求值上下文（外部）和项（内部）。此时，栈的长度小于或等于 I。取消所有静默变量的静默状态是安全的。当不存在静默变量时，$\mathbf{check}_{\mathsf{M}}$ 和 **check** 具有相同的行为。

然而，以这种方式改变语义意味着任何转换都可能取消变量的静音。为了保持语义的标准形式，并更贴切地建模 check 的实现，我们将取消静音这一操作与 **dlet** 和 **tls**关联起来。从 **dlet** 出发的转换会有条件地取消变量的静音（Sec-Dlt，图 5）。在 图 10b中，从 **dlet**($z_{\mathbb{N}}$, **返回** n)出发的转换会取消变量的静音。因此，在检查 变量($x_{\mathbb{N}}$) 时，它仍处于静音状态；而在检查外层 应用构造子时，它已取消静音；外层 Lam构造子则在检查时取消静音。

此外，从 **tls** 出发的转换会无条件地 取消变量的静音，因为 **tls** 之后的求值上下文必须是惰性的，因此永远不会被处理器捕获（Sec-Tls）。

由于**检查**$_{\mathsf{M}}$ 若失败，则**检查** 也绝不会成功，因此 C4C 检查至少与急切检查同样宽松。

4.4.1 C4C 检查的正确性。C4C 检查对于不可避免的作用域外逸是正确的。证明很简单：要么某个非顶层拼接**检查**$_{\mathsf{M}}$报告错误，要么都不报告错误。后一种情况退化为惰性检查，此时顶层拼接**检查**$_{\mathsf{M}}$ 必须报告错误。

定理 *4.2*（*C4C Check* 的正确性）。给定一个封闭且类型正确的 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ 表达式e，如果 $\langle [\![e]\!]; [-]; \emptyset; \emptyset; \top \rangle$ 表现出不可避免的作用域外逸，那么存在 E, U, M, I，使得 $\langle [\![e]\!]^{BE}; [-]; \emptyset; \emptyset; \top \rangle \rightarrow^* \langle err; E; U; M; I \rangle$

4.4.2 C4C 检查的表达能力。 C4C 检查的表达能力并非最大。具体而言，它不允许执行代码清单 11中的程序。

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

```
$(λx.$(处理 λy.$(操作(y);返回 y)
        with {返回(u) ↦ 返回 ⟨⟨0⟩⟩; 操作(z, k) ↦ 返回 z}));
 1)
```

代码清单 11。C4C 检查报告了假阳性。

代码清单 11 尝试构建抽象语法树 $\lambda x.$**返回** y，其中 y 已经外提了其作用域，但随后又将其丢弃，返回了 1的抽象语法树。关键在于，外层 Lambda 的构造函数 $\lambda x.[-]$从未被任何效应捕获。因此，代码清单 11 最终归约为如下配置：

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

其中 $\mathrm{len}(E[\mathbf{检查}_{\mathsf{M}}\,[-]]) \leq I$。后续转换会取消 $\mathsf{Var}(y_{\mathbb{N}})$ 的静音状态，而外围的**检查**$_{\mathsf{M}}$ 会失败，因为 $\mathsf{Var}(y_{\mathbb{N}})$ 是自由的、未静音的，并且在 E中未声明为安全。

关注原因性质刻画了 C4C 检查的表达能力[1]。该性质的非正式定义如下：假设检查报告错误，并令违规的抽象语法树为n。现在回溯到检查失败的位置，并考虑另一种执行方式：其中所有的 **检查**$_{\mathsf{M}}$都被擦除（转换为 **返回**）。在这种反事实执行中，所有由 n' 构造的抽象语法树 n 在其求值上下文中都至少包含一个未声明为安全的变量。因此，在 代码清单 11中，安全使用 $\lambda x.$**返回** y 的唯一办法就是将其丢弃。

定理 *4.3*（关注原因性质）。假设有一个封闭且类型良好的 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ 表达式 e，若 $\exists E, U, M, I$使得 $\langle [\![e]\!]^{BE}; [-]; \emptyset; \emptyset; \top \rangle \rightarrow^* \langle$ 检查 $_M\, n; E; U; M; I \rangle$，且 $\langle$ 检查 $_M\, n; E; U; M; I \rangle \rightarrow \langle err; E; U; M; I \rangle$，那么，假设 $\langle$ 返回 $n;$ 擦除-检查$(E); U; M; I \rangle \rightarrow^* \langle$返回 $n'; E'; U'; M'; I' \rangle$，且 n 是 n'的一个子树，则必有 $FV^0(n') \not\subseteq \pi_{\mathsf{Var}}(E')$。

定理 4.3 的证明采用反证法。直观地说，如果$\mathsf{FV}^0(n') \subseteq \pi_{\mathsf{Var}}(E')$，那么 n 中的所有变量都必须声明为安全。这意味着，当初始 **检查**$_{\mathsf{M}}$ n 失败时，栈上存在一个延续，可以将 n 声明为安全。但这样一来， M 就不可能为空，因此检查不会失败。

急切检查的表达能力无法用关注原因性质来刻画，代码清单 10 就是一个反例。因此，C4C 检查的表达能力**更强**，而且其表达能力**更易于预测**，胜过急切检查。

与急切检查一样，可以通过在配套制品中运行 代码清单 7至11的 MacoCaml 转换版本，实证验证 C4C 检查的表达能力 [Lee 等人 2025]。

[1]并给出 这是它的名称；与惰性检查和急切检查不同，该名称描述的是面向用户的行为，而非其操作。

表3. 正确性比较

	代码清单				
	8	9	10	11	12
惰性 (§4.2)	Y	Y	Y	Y	Y
急切 (§4.3)	N	N	Y	Y	Y
C4C (§4.4)	Y	Y	Y	Y	Y
参考环境分类器 (§5)	Y	Y	Y	Y	Y

表 4. 表达能力比较

	代码清单				
	8	9	10	11	12
惰性 (§4.2)	Y	Y	Y	Y	Y
急切 (§4.3)	Y	Y	N	N	Y
C4C (§4.4)	Y	Y	Y	N	Y
参考环境分类器 (§5)	N	N	N	N	Y

4.5 $\lambda_{\langle\langle \mathbf{op} \rangle\rangle}$的求值

我们已经证明 $\lambda_{\langle\langle \mathbf{op} \rangle\rangle}$ 适用于编码和求值作用域外逸检查。表 3 和 表 4 总结了这些检查的正确性和表达能力（包括精化环境分类器，以及下一节将讨论的代码清单12）。配套制品 [Lee 等人 2025] 提供了代码清单 7 至11 在 BER MetaOCaml N153 和 MacoCaml 中的译文。通过执行这些译文，可以对表 3 和表 4 的前三行进行经验验证。在 $\lambda_{\langle\langle \mathbf{op} \rangle\rangle}$ 框架下统一这些检查，便于针对同一组程序进行比较评估。此外，在 $\lambda_{\langle\langle \mathbf{op} \rangle\rangle}$ 中形式化作用域外逸，有助于开发新颖的 C4C 检查，它在急切检查和惰性检查之间取得了恰当的平衡。

值得再次强调的是，$\lambda_{\langle\langle \mathbf{op} \rangle\rangle}$ 关注的是作用域外逸。本文未探讨生成代码中与绑定相关的其他有趣问题。具体而言，$\lambda_{\langle\langle \mathbf{op} \rangle\rangle}$ 并不能阻止遮蔽：在 $\lambda_{\langle\langle \mathbf{op} \rangle\rangle}$ 中，可能生成包含多个绑定子且使用相同形式参数的程序。例如，使用多次续延时，可以在 $\lambda_{\langle\langle \mathbf{op} \rangle\rangle}$ 中生成如下代码：Lam(x, Lam(x, 主体))，因为绑定子 Lam(x, –) 可能被一个以嵌套方式重新实例化的延续捕获。三种动态检查都无法检测到每一种作用域遮蔽情况。将 $\lambda_{\langle\langle \mathbf{op} \rangle\rangle}$ 限制为只允许单次续延（如MetaOCaml 和 MacoCaml等系统），可以防止遮蔽；但在实践中，多次续延在多阶段编程中很有用，例如用于分支插入[Yallop, 2017年, §4。4], 而且不会损害类型安全。

5 扩展：精化环境分类器

本节介绍 $\lambda_{\langle\langle \mathbf{op} \rangle\rangle^{\gamma}}$，这是对 $\lambda_{\langle\langle \mathbf{op} \rangle\rangle}$ 的扩展，加入了 精化环境分类器 [Kiselyov 等人 2016]提出的静态预防作用域外逸的方法，遵循Isoda 等人 [2024]的工作。§4展示了如何使用 $\lambda_{\langle\langle \mathbf{op} \rangle\rangle}$ 来比较动态作用域外逸检查，并且 $\lambda_{\langle\langle \mathbf{op} \rangle\rangle^{\gamma}}$ 说明如何扩展该框架，以描述和评估静态预防技术。

5.1 演算

图 12 展示了 $\lambda_{\langle\langle \mathbf{op} \rangle\rangle^{\gamma}}$的类型和精选类型规则。该演算与 $\lambda_{\langle\langle \mathbf{op} \rangle\rangle}$共享语法，并扩展了它，加入[2] 了 Isoda 等人类型系统的简化版本。

直观地说，分类器 表示一个允许一组自由变量的作用域。如果抽象语法树在给定作用域内类型正确，且其所有自由变量都被该作用域允许，则称该抽象语法树在该作用域内作用域良好。

例如，请看图 11，其中有两个分类器：γ_α 是仅允许变量(α)的作用域，而 $\gamma_\perp$ 是不允许任何变量的作用域（即“顶层”）。为了表示作用域的嵌套关系，分类器之间存在偏序关系$\gamma \sqsubseteq \gamma'$，其中 γ 是外层作用域，而 γ' 是内层作用域；在此例中，我们有 $\gamma_\perp \sqsubseteq \gamma_\alpha$。

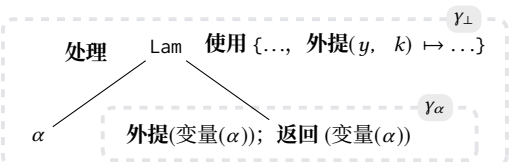


图 11. 精化后的环境分类器

[2]Isoda 等人的 处理器和延续的类型规则对分类器是多态的，以允许 let 插入 .

分类器通过检查所创建的抽象语法树是否作用域正确，防止作用域外逸；此外，它们还确保对抽象语法树的操作会保留作用域正确性。具体而言，它们会阻止变量被提升到不允许其出现的作用域中。在图 11中，**外逸**效应试图将变量(α)提升到 $\gamma_\perp$ 作用域中的处理器（该作用域不允许变量(α)）。无论处理器的主体是什么，图 11都无法通过类型检查。

语法。 我们使用分类器标注 -1 代码 类型 γ。对于 0 级类型，分类器会关联在类型上下文和类型判断中。我们定义 扩展 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle^\gamma}$ 类型，这一概念对于定义§5.2中的逻辑关系很有用。

定义 5.1（扩展 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle^\gamma}$ 类型）。一个扩展 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle^\gamma}$ 类型可以是以下任一种：
（*1*）一个级别为 -1 的类型，例如 (代码($\mathbb{N}^0$! $\emptyset$)$^\gamma$)$^{-1}$；（*2*）一个带分类器的 0 级类型，例如 $\mathbb{N}^0(\gamma)$；或者（*3*）一个 0 级形式参数类型，它是一个带分类器γ和下划线的 0 级值类型（例如 $\mathbb{N}^0$），用于表示它会被展开为 *FParam* 类型 $\mathbb{N}^0(\gamma)$。

类型上下文 Γ 将项映射到其类型，并跟踪 0 级类型的环境分类器。此外，它还跟踪分类器 γ及其偏序 $\gamma \sqsubseteq \gamma'$。如果 0 级 -1 类型T^{-1} 在上下文 Γ下良构，记作 $\Gamma \vdash T^{-1}$，则其所有分类器都属于 Γ。如果一个上下文包含最小 分类器 $\gamma_\perp$，且其中所有类型都良构，那么该上下文就是良构的。我们假设所有上下文都是良构的。

类型规则。 大多数类型规则都可以直接改编，关键规则列于图 12。 **c**|**q**-变量 规则指出，分类器为 γ的变量在分类器 γ下类型正确。特别值得关注的是 **c**|**q**-Lambda 规则。由于分类器将作用域的概念形式化，该规则会引入一个新作用域，并以一个新鲜分类器 $\gamma' \notin \Gamma$ 表示，同时将变量 x与 γ'关联起来。此外，由于 γ' 是在 γ的作用域内创建的，因此有 $\gamma \sqsubseteq \gamma'$。

该**c**|**q**-子表达式和**s**-子规则形式化了作用域的嵌套：若要证明某个项在嵌套作用域 γ中作用域良构，只需证明它在其任一父作用域中作用域良构 γ'，其中 $\gamma' \sqsubseteq \gamma$。

遵循 Isoda 等人， [2024], 操作（s-Op）、延续（s-Continue）和处理器（s-Handle）仅限用于 Code 类型，不过很容易推广到非 Code 类型。这些规则协同作用，以防止作用域外逸。具体而言，handle 构造作用于 Code 类型，并在某个作用域 γ中执行。因此，处理器不能改变计算结果所允许处于的作用域，例如，不能将类型从 Code($\mathbb{N}$)$^\gamma$ 改为具有不同分类器的 Code($\mathbb{N}$)$^{\gamma'}$。值得注意的是，每个处理器子句（因而也包括每个已处理效应）都会继承此作用域 γ。因此，传递给已处理效应的值若为代码类型，就必须带有一个可替换为 γ 的分类器。由于绑定子会引入新的分类器 γ'，而 γ' 不能替换为 γ（因为 $\gamma' \not\sqsubseteq \gamma$），导致作用域外逸的示例无法通过类型检查。

最后，我们指出，由于上下文必须包含最小分类器 $\gamma_\perp$，表达式 e是封闭的类型良好的表达式，当且仅当 $\gamma_\perp \vdash^{\gamma_\perp}_{\mathbf{c}} e : T^0 \,!\, \emptyset; \emptyset$。

Elaboration（细化）。与 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ 类似， $\lambda_{\langle\langle \mathsf{op} \rangle\rangle^\gamma}$ 没有操作语义，而是被 elaboration 为 $\lambda_{\mathsf{AST}(\mathsf{op})^\gamma}$ 项，其中形式参数带有分类器注解（例如 α^γ_R）。分类器只出现在形式参数中，对类型不可见。由于 elaboration 不需要任何动态作用域外逸检查机制，$\lambda_{\mathsf{AST}(\mathsf{op})^\gamma}$ 没有 **check**、**check**$_\mathsf{M}$、**dlet**、**tls** 和 **err**。因此， $\lambda_{\mathsf{AST}(\mathsf{op})^\gamma}$ 配置的形式为 $t; E; U$。

elaboration 与 §3.3类似，不同之处在于，类型的 elaboration 会擦除分类器，上下文条目的 elaboration 会擦除证明论项，而项的 elaboration 假定绑定子已标注扩展源类型，并且 不会 擦除分类器。最后，顶层拼接的 elaboration 不会插入 **tls**。

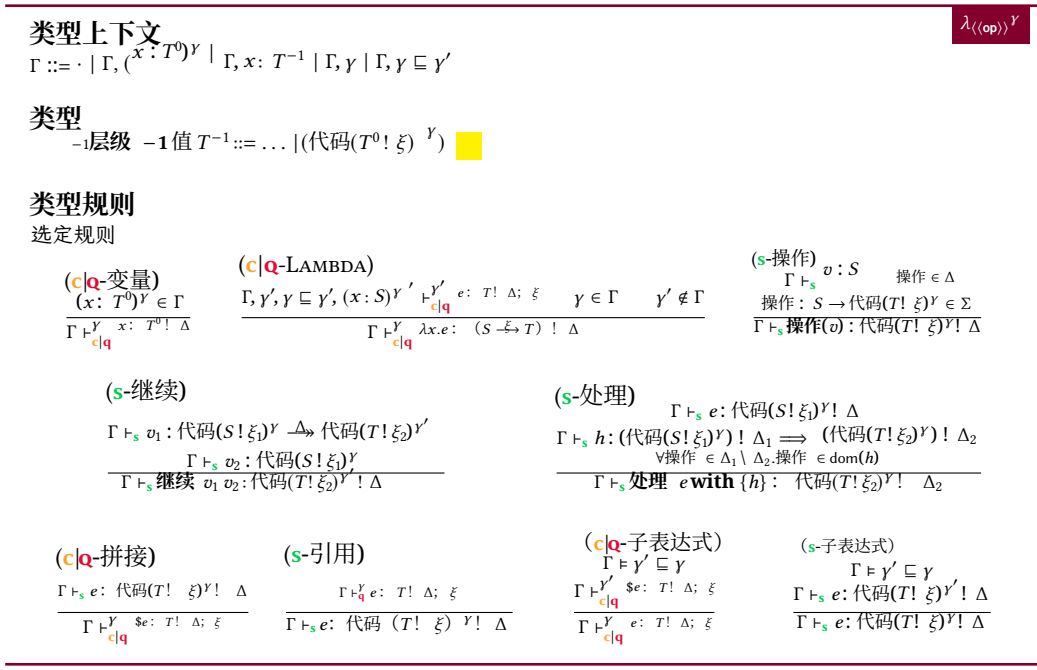


图 1 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle^\gamma}$：类型与精选类型规则。

弱化。 我们证明一个弱化引理，这对之后证明正确性很有用。由于类型分为两个层级，并且分为值类型、计算类型和处理器类型，因此有多个子引理。作为示例，我们给出 0 级计算的弱化：

引理 5.2（0 级计算的弱化）。 若 $\Gamma \vdash^\gamma_{c|q} e : T^0 ! \Delta$ 则

(1) $\Gamma, (x : S^0)^{\gamma'} \vdash^\gamma_{c|q} e : T^0 ! \Delta$，其中 $\gamma' \in \Gamma$、 $x \notin \Gamma$ 任意；*(2)* $\Gamma, (x : S^{-1}) \vdash^\gamma_{c|q} e : T^0 ! \Delta$，其中 $\Gamma \vdash S^{-1}$、 $x \notin \Gamma$；*(3)* $\Gamma, \gamma' \vdash^\gamma_{c|q} e : T^0 ! \Delta$，其中 $\gamma' \notin \Gamma$ 任意

(4) $\Gamma, \gamma' \sqsubseteq \gamma'' \vdash^\gamma_{c|q} e : T^0 ! \Delta$，其中 γ'、 $\gamma'' \in \Gamma$ 任意

5.2 精化环境分类器的正确性

在本节中，我们证明精化环境分类器的正确性：每个类型规则正确的 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle^\gamma}$项在终止时都会生成一个作用域正确的抽象语法树。然而，在基于 elaboration 的语义中，直接对 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle^\gamma}$ 进行推理颇具挑战性。因此，我们采用 Tait 式逻辑关系[Tait 1967] 来证明类型保证会通过 elaboration 得以保留 [Benton 和 Hur2009], 从而确立精化环境分类器的正确性。

图13给出了定义在核心语言（$\lambda_{\mathsf{AST}(\mathsf{op})^\gamma}$）项上的逻辑关系 Scoped。该关系由证明论项上下文 Θ 和扩展的 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle^\gamma}$ 类型（定义5.1）进行索引。给定一个上下文 Γ， π_γ（Γ）仅投影出证明论项。例如，给定 $\Gamma = \gamma_\perp,\ \gamma_1,\ \gamma_\perp \sqsubseteq \gamma_1,\ (x : \mathbb{N}^0)^{\gamma_1},\ \gamma_2,\ \gamma_1 \sqsubseteq \gamma_2,\ y : (\text{Code}(\mathbb{N}^0 ! \emptyset)^{\gamma_2})^{-1}$，该上下文的证明论部分为 $\pi_\gamma(\Gamma) = \gamma_\perp,\ \gamma_1,\ \gamma_\perp \sqsubseteq \gamma_1,\ \gamma_2,\ \gamma_1 \sqsubseteq \gamma_2$，它是 Θ 的一个实例。

两个关键定义分别是关于 $T^0(\gamma)$ 值类型的关系（$\text{Scoped}_{\Theta,\ T^0(\gamma)}$），以及关于项的关-系（$\text{Scoped}_{\Theta,\ \tau ! \Delta}$）。要使范式 n 属于 $\text{Scoped}_{\Theta,\ T^0(\gamma)}$， n 必须具有类型

Scoped$_{\Theta,\ T}$逻辑关系 $\lambda_{\langle\langle \mathbf{op} \rangle\rangle^{\gamma}}$

证明论术语的上下文

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

范式

下文中，令 τ 表示以下任一形式：$T^0\ (\gamma)$、$T^0\,!\ \xi\ (\gamma)$、$(S^0\,!\ \xi_1 \Longrightarrow T^0\,!\ \xi_2)^{\ 0}\ (\gamma)$，或 $(\text{代码}\ (T^0\,!\ \xi)^{\gamma})^{-1}$

$n \in \quad _{\Theta\,\mathbb{N}^{-1}} \triangleq n \in \mathbb{N}_{\text{Scoped},\text{Scoped}\Theta,\tau} \triangleq .$

$n \in$

$\vdash n \in [\![\tau]\!]$ 和 $\Theta \vdash \text{FV}^0(n) \subseteq$ 允许$(\gamma) n \in \text{Scoped}\Theta, T^0(\gamma) \triangleq$ 变量$(n) \in \text{Scoped}\Theta, T^0(\gamma) n \in \text{Scoped}\Theta,$

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

$_{-1} \quad \triangleq \forall n' \in \text{Scoped}\Theta, S^{-1},$ 继续 $n\ n' \in \text{Scoped}\Theta, T^{-1}\,!\ \Delta$

处理器

$h \in \text{Scoped}_{\Theta,(S^{-1}\,!\,\Delta_1 \Longrightarrow T^{-1}\,!\,\Delta_2)^{-1}} \triangleq$ if $h = \mathbf{return}(x) \mapsto t_{\text{ret}}$

$\forall n' \in \text{Scoped}_{\Theta, S^{-1}}, t_{\text{ret}}[n'/x] \in \text{Scoped}_{\Theta, T^{-1}\,!\,\Delta_2}$

else $h = h'; \mathbf{op}(x, k) \mapsto t_{\text{op}}, \text{op} : A^{-1} \to B^{-1}$

$h' \in \text{Scoped}_{\Theta,(S^{-1}\,!\,\Delta_1 \Longrightarrow T^{-1}\,!\,\Delta_2)^{-1}}$ and

$\forall n \in \text{Scoped}_{\Theta, A^{-1}}, n' \in \text{Scoped}_{\Theta, B^{-1} \to T^{-1}}, t_{\text{op}}[n/x, n'/k] \in \text{Scoped}_{\Theta, T^{-1}\,!\,\Delta_2}$

项

In the following, let $\tau\,!\,\Delta\ b$ *e shorthand f*或任何 $o^{\ f\ T^0\,!\ \Delta\ (\ \ \ \gamma), T^0\,!\ \Delta; \xi(\gamma), (S^0\,!\,\xi_1 =\!\Rightarrow T^0\,!\ \ \xi)_2^0\,!\ \ \Delta(\ \ \gamma), \text{or}\ T^{-1}\,!\,\Delta}$

给定编译时计算类型 $\tau!\,\Delta$，令 τ 表示相应的值类型。例如，如果 $\tau!\,\Delta = T^0!\,\Delta; \xi(\gamma)$，那么 $\tau = T^0!\,\xi(\gamma)$

$\text{Scoped}_{\Theta,\tau\,!\,\Delta} \triangleq$ 项上的最小属性 t，满足以下任一条件：

(1) 对于任意与 t 一致的 U，存在 $U\prime$，使得 $t;\ [-];U \to^* \mathbf{返回}\ n;\ [-];U\prime$，满足 U' 与 n一致，且 $n \in \text{Scoped}\Theta_{,\tau}$

(2) 对于任意与 t一致的U，存在$U\prime$，使得 $t;\ [-];U \to^* \mathbf{操作}(n);\ E;U\prime$，其中 $\mathbf{操作} \notin$ 已处理(E)，$U\prime$与 $E[\mathbf{操作}(n)]$，一致，且

(a) 操作 $: A^{-1} \to B^{-1}$，

(b) $n \in \text{Scoped}\Theta_{,A^{-1}}$，以及

(c) 对于所有 $n' \in \text{Scoped}\Theta_{,B^{-1}}$，$E[n'] \in \text{Scoped}\Theta_{,\tau\,!\,\Delta}$

其中，在此上下文中，与 t一致意味着，对于所有 $\text{Var}(\alpha_R^{\gamma})$ 或 $\alpha_R^{\gamma} \in t$，$\alpha \in U$。此旁条件确保我们正确使用**mkvar**。

图 13. 作用域逻辑关系的定义

抽象语法树(擦除(T^0))，且 n 的自由变量需要能在 γ 所表示的作用域内获准使用。许可性假定分类器上存在某种已知的偏序，例如 $\gamma' \sqsubseteq \gamma$，该偏序由索引 Θ 携带。$\text{Scoped}_{\Theta,\ \tau\,!\,\Delta}$ 定义为最小不动点，遵循 Plotkin 和 Xie [2025]以及Kuchta [2023]，给出的类似定义，由此导出 Scoped-归纳原理：

归纳 原理 5.3 (Scoped-归纳)。对于类型为 $[\![\tau\ !\ \Delta]\!]$ 的闭项上的性质 Φ，*(1)* 若 $\langle t;\ [-];U\rangle \to^* \langle$返回 $n;\ [-];\ U'\rangle$ 蕴含 $\Phi(t)$，且 *(2)* 若 $\langle t;\ [-];U\rangle \to^* \langle$操作$(n);\ E;\ U'\rangle$，其中操作 $\notin$ 已处理(E)，操作 $: A^{-1} \to B^{-1}, n \in \text{Scoped}_{\Theta,A^{-1}}$，并且对于任意 $n' \in \text{Scoped}_{\Theta,B^{-1}}, \Phi(E[n'])$ 蕴含 $\Phi(t)$，

那么对于所有 $t \in \text{Scoped}_{\Theta,\ \tau\,!\,\Delta}$，$\Phi(t)$

该证明还依赖于闭包引理 [Kuchta 2023] 以及闭替换的概念$\rho \vDash \Gamma$。替换层级为 0 的变量时必须格外小心，因为这些变量应属于 FParam的逻辑关系，而非 抽象语法树（条款 2，见 定义 5.5）。

引理 5.4（反向归约下的封闭性）。假设$\langle t;\ E;\ U\rangle \to^* \langle t';\ E';\ U'\rangle$。则 $E'[t'] \in \mathit{Scoped}_{\Theta,\ \tau\ !\ \Delta} \implies E[t] \in \mathit{Scoped}_{\Theta,\ \tau\ !\ \Delta}$

定义 5.5（闭替换）。给定一个上下文 Γ，并假设 $\Theta = \pi_\gamma(\Gamma)$，闭替换集合 $\rho \vDash \Gamma$ 归纳定义如下：

(1) $() \vDash \gamma_\perp$ *(2)* 若 $\rho \vDash \Gamma$，则对任意 $\gamma \in \Gamma, n \in \mathit{Scoped}_{\Theta,T^0(\gamma)}$，$(\rho, n/x) \vDash \Gamma, (x: T^0)^\gamma$ *(3)* 若 $\rho \vDash \Gamma, \Gamma \vdash T^{-1}$ 且 $n \in \mathit{Scoped}_{\Theta,T^{-1}}$，则 $(\rho, n/x) \vDash \Gamma, (x: T^{-1})$ *(4)* 若 $\rho \vDash \Gamma$，则对任意 $\gamma \notin \Gamma$，$\rho \vDash \Gamma, \gamma$ *(5)* 若 $\rho \vDash \Gamma$，则对任意 γ，$\gamma' \in \Gamma$，$\rho \vDash \Gamma, \gamma \sqsubseteq \gamma'$

最后，我们引入一个 Θ-截断引理，使我们能够在证明不需要证明论术语（γ、$\gamma \sqsubseteq \gamma'$）时将其省略。

引理 5.6（Θ-截断）。假设有一个抽象语法树 n。若 $FVs^0(n)$中不出现变量 $Var(\alpha_S^{\gamma'})$，且 $Var(\alpha_S^{\gamma'})$是唯一一个带有分类器 γ' 标记的变量，那么 $n \in \mathrm{Scoped}_{(\Theta,\gamma',\gamma\sqsubseteq\gamma')\ \tau}$蕴含 $n \in \mathrm{Scoped}_{\Theta,\ \tau}$

类型分层和模态索引将基本引理分解为许多子引理；这里我们给出其中一个子引理：

引理 5.7（基本引理 $[\mathrm{c},\ T^0\ !\ \Delta;\ \xi]$ 关于作用域逻辑关系）。若 $\Gamma \vdash^{\gamma}_{c} e : T^0\ !\ \Delta;\ \xi$，则对 $\Theta = \pi_\gamma(\Gamma)$，以及所有满足 $\rho \vDash \Gamma$ 的 ρ，$[\![e]\!]_c(\rho) \in \mathrm{Scoped}_{\Theta,T^0\ !\ \Delta;\xi(\gamma)}$

引理 5.7 的证明通过对 $\lambda_{\langle\langle \mathrm{op} \rangle\rangle^\gamma}$ 类型规则进行归纳。在 c-Lambda 情形中，只需证明对于 $\rho \vDash \Gamma$，do $x \leftarrow \mathrm{mkvar\ erase}(S^0(\gamma'))$ in do body $\leftarrow [\![e]\!]_c(\rho)$ in return $\mathrm{Lam}(x, \mathrm{body})$ 属于 $\mathrm{Scoped}_{\Theta,(S^0 \xrightarrow{\xi} T^0)^0\ !\ \Delta(\gamma)}$。这可化简为 do body $\leftarrow [\![e]\!]_c(\rho, \alpha_S^{\gamma'}/x)$ in return $\mathrm{Lam}(\alpha_S^{\gamma'}, \mathrm{body})$。根据反向归约（引理 5.4），只需证明该项属于逻辑关系。根据弱化（引理 5.2）和归纳假设（归纳假设），$[\![e]\!]_c(\rho, \alpha_S^{\gamma'}/x) \in \mathrm{Scoped}_{\Theta',T^0\ !\ \Delta;\xi(\gamma')}$，其中 $\Theta' = \Theta, \gamma', \gamma \sqsubseteq \gamma'$。只需证明：

$\forall t \in \mathrm{Scoped}_{\Theta',\ T^0\ !\ \Delta;\ \xi(\gamma')}$，**执行**主体 $\leftarrow t$**在返回**$\mathrm{Lam}(\alpha_S^{\gamma'}$，主体$)$中$\mathrm{Scoped}_{\Theta,\ (S^0 \xrightarrow{\xi} T^0)^0\ !\ \Delta(\gamma)}$

应用作用域归纳，

(1) $t \in \mathrm{Scoped}_{\Theta',\ T^0\ !\ \Delta;\ \xi(\gamma')}$归约为某个**返回** n **执行**主体 $\leftarrow$ **返回** n **在返回**$\mathrm{Lam}(\alpha_S^{\gamma'}$，主体$)$归约为**返回**$\mathrm{Lam}(\alpha_S^{\gamma'}, n)$，其中 $\mathrm{Var}(\alpha_S^{\gamma'})$ 已绑定。根据归纳假设，$n \in \mathrm{Scoped}_{\Theta',T^0\ !\ \xi(\gamma')}$。因此，$n$ 中的所有自由变量都被 γ'许可。根据类型规则，只有 α带有分类器 γ'注释。因此，根据引理 5.6，在 Θ下，$\mathrm{Lam}(\alpha_S^{\gamma'}, n)$的自由变量都被 γ许可。结论由反向归约得出。

(2) $t \in \mathrm{Scoped}_{\Theta',\ T^0\ !\ \Delta;\ \xi\ (\gamma')}$ 归约为 $E[$**操作**$(n)], $**操作**$(n)$ 未处理。由于**执行**主体 $\leftarrow [-]$ **中返回**$\mathrm{Lam}(\alpha_S^{\gamma'}$，body$)$ 不会引入任何处理器，因此结论可直接由 Scoped-Induction 归纳假设和反向归约得出。

利用逻辑关系，以及与 Bauer 和 Pretnar的 [2008] 推论 4.2 完全相同的类型安全性结果，我们证明精化环境分类器的正确性：

定理 5.1（精化环境分类器的正确性）。如果$\gamma_\perp \vdash^{\gamma_\perp}_{c} e : T^0\ !\ \emptyset;\ \emptyset$，且$[\![e]\!]_c = t$，那么对于某个 U，$t;\ [-];\ \emptyset \to^*$ **返回** $n;\ [-]; U$，且$FV^0(n) = \emptyset$

5.3 精化环境分类器的表达能力

$\lambda_{\langle\langle \mathsf{op} \rangle\rangle^Y}$ 通过仅检查效应的参数而非处理器，来防止作用域外逸。在良类型的 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle^Y}$ 程序中，可能传递给效应**操作**的唯一变量，是定义**操作**的处理器时处于作用域内的变量，例如 z 中的代码清单12：

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

```
λz.$(处理  λx.返回 $( 操作(z) )
      其中 {返回(u)  ↦ 返回  u; 操作(y, k)  ↦ 继续 k() })
```

代码清单12. 精化环境分类器允许将变量传递给效应，只要该变量永远不会导致作用域外泄错误（例如，z 可以被传递，因为它在处理器定义之外绑定）。

表 4 总结了精化环境分类器在我们这组程序中的表达能力；如图所示，精化环境分类器的表达能力不及所有动态检查。

6 实现

我们已在 MacoCaml 编译器中实现了各种动态检查，并将带有 C4C 检查的实现作为制品提供 [Lee 等人 2025]。MacoCaml 通过 elaboration 实现代码引述，其方式与§3.3中的细化类似，但目标是较低层级的中间语言 Lambda，而不是抽象语法树。我们扩展了 elaboration，加入外逸检查，其方式类似于第 4.2 节至4.4节中的扩展细化。

MacoCaml 的实现§4。该实现将**检查**、**dlet**和**err** 实现为效应与处理器的一种使用模式：

(1) **check** n 通过执行`FreeVar` 效应实现，并将 n的自由变量传递给它。**check**$_{\mathsf{M}}$ n 与此类似，但还依赖于一个`Mute` 效应，该效应在除`FreeVar` 之外的效应处理器中执行，以实现 Eff-Op 规则（图 5）。(2) **dlet**(α_R, t) 实现为`FreeVar` 效应的处理器：它从自由变量集合中减去变量(α_R)，然后执行以下操作之一：(a) 如果自由变量集合此时为空（即所有自由变量都已声明安全），则恢复延续；或者 (b) 执行另一个`FreeVar` 效应，以检查剩余的自由变量是否已声明安全。如果检查成功返回，则恢复延续。

(3) **err** 被实现为一个未处理的 FreeVar 效应。

7 相关工作

使用变异和控制效应生成代码，尤其是进行 let 插入，有着悠久的历史 [Lawall 和 Danvy 1994; Sumii 和 Kobayashi 2001]; Kameyama 等人 [2011, §8] 以及 Kameyama 等人 [2015, §5.3] 对此进行了全面综述。生成包含未绑定变量的代码所带来的危险也逐渐显现。针对这一问题，已有两类研究：预防和检测。预防方面的大多数研究都致力于设计合适的类型系统，例如闭合类型 [Calcagno 等人 2000] 或环境分类器 [Taha 和 Nielsen2003]（不过，后者预防的是由 eval 而非效应引起的作用域外逸）。旨在预防作用域外逸的大多数类型系统都复杂得多[Isoda 等人 2024; Kameyama 等人 2015; Kiselyov 等人 2016; Parreaux 2020], ，本质上都是 Nanevski 等人 [2008] 上下文模态类型理论的变体。

除了类型之外，还可以通过限制变异的作用域（即所谓的弱可分离性 [Westbrook 等人 2010]），或通过在每个未来阶段绑定符下放置一个效应处理器来限制控制效应的作用域 [Kameyama 等人 2011]。延续传递或单子变换 [Swadi 等人 2006] 也能达到相同效果。

大多数预防技术都会限制语言的表达能力，而且往往限制得很严重。Kameyama 等人 [2015] 提出了一组基准测试，用于评估程序生成系统的表达能力；当时只有Kameyama 等人 [2015]通过了所有基准测试。

而 预防 技术会在编译代码生成器时，静态拒绝可能不安全的代码生成程序；而 检测 技术则在执行代码生成器时发挥作用：当生成了包含作用域外逸变量的代码片段时，它们会提醒元编程人员。由于生成的代码最终必须经过编译，最简单的检测技术便是在代码生成过程中什么也不做，而是依靠生成代码的编译器报告外逸问题。第2节 将这种方法称为惰性检查；正如我们强调过的，它在实践中存在严重的易用性问题。 Kiselyov [2014] 努力实现了急切检查，在作用域外逸发生时立即检测，在代码完全生成之前就给出信息丰富的错误消息。然而，这种设计和实现并未经过形式化。因此，很难评估这项检查（甚至难以判断它是否能在尽可能早的时机检测到错误）。

很少有预防方法被应用于实际使用的系统中（或者至少是用于真实、大规模示例的系统）：例如 Mint [Westbrook 等人 2010], Contextual Squid [Parreaux 2020] 和 StagedHaskell [Kameyama 等人 2015]。Mint 的限制非常严格，完全禁止在绑定子之后进行 let 插入和断言插入。其他基于类型的方法允许 let 插入和循环交换等优化，但复杂得多。Contextual Squid 基于 Scala-2 宏实现，需要访问 Scala 编译器内部机制，而这些机制在 Scala-3 中已不再可用。StagedHaskell 依赖复杂的 Haskell 类型类编程，通常需要类型注解，而且类型可能变得相当复杂，错误消息也令人难以理解。遗憾的是，类型的复杂性是必要的：Kameyama 等人的工作第 4.1 节展示了他们系统的一个类型更简单的版本中，意外出现的绑定变量可能引发微妙而严重的问题。Parreaux [2020] 指出，一些用户觉得处理上下文类型负担过重。 Kiselyov [2014] 进一步讨论了作用域外溢检查的实际应用方面。

许多实用的元编程系统，例如 TemplateHaskell [Sheard 和 Jones 2002]，依赖生成时检测作用域外逸，尤其是惰性检查，也就是将所有检测工作都交给编译生成代码的编译器。一个显著的例外是 (BER) MetaOCaml [Kiselyov 2014、2024b],，其中 急切的作用域外逸检测是其核心特性。

8 结论

我们提出了首个用于比较作用域外溢检查的形式化框架，该框架基于演算 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ 及其向 $\lambda_{\mathsf{AST}(\mathsf{op})}$ 核心语言的 elaboration。借助这一框架，我们对作用域外逸检查的两种主要方法——惰性检查和急切检查——进行了建模，并提出了一种结合二者优点的新检查方式；这种方式与效应和处理器能够良好交互。我们已将这一新检查方式集成到 MacoCaml 的实现中。我们的框架还可扩展用于建模经过改进的环境分类器系统，以防止作用域外逸；我们预计，该框架也可以类似地扩展，用于建模其他静态系统。

致谢

我们感谢匿名的 POPL 审稿人以及 Neel Krishnaswami，他们的意见帮助改进了这项工作；也感谢 Alistair O’Brien 和 YulongHuang，他们的意见帮助改进了制品。本研究由 Jane Street Capital、Ahrefs 以及加拿大自然科学与工程研究委员会资助。

参考资料

Andrej Bauer 和 Matija Pretnar。2014。代数效应与处理器的效应系统。《计算机科学中的逻辑方法》第 10 卷，第 4 期（2014 年 12 月）。doi:10.2168/lmcs-10(4:9)2014 Nick Benton 和 Chung-Kil Hur。2009。双正交性、步索引与编译器正确性。载于第 14 届 ACM SIGPLAN 国际函数式编程会议论文集（苏格兰爱丁堡）（ICFP '09）。计算机协会，美国纽约州纽约市，97–108。doi:10.1145/1596550.1596567 Dariusz Biernacki、Maciej Piróg、Piotr Polesiuk 和 Filip Sieczkowski。2017。谨慎处理：代数效应与处理器的关系解释。ACM 程序设计语言会议论文集 2，POPL，文章 8 （ 2017 年 12 月），30 页。doi:10.1145/3158096 Cristiano Calcagno 、 Eugenio Moggi 和 Walid Taha。2000。闭合类型：安全命令式多阶段编程的一种简单方法。载于《自动机、语言与编程》，Ugo Montanari、José D. P. Rolim 和 Emo Welzl（编）。Springer Berlin Heidelberg，柏林、海德堡，25–36。Cristiano Calcagno、Walid Taha、Liwen Huang 和 Xavier Leroy。2003。使用抽象语法树、Gensym 和反射实现多阶段语言。载于《生成式编程与组件工程》，Frank Pfenning 和 Yannis Smaragdakis（编）。Springer Berlin Heidelberg，柏林、海德堡，57–76。Jacques Carette、Mustafa Elsheikh 和 W. Spencer Smith。2011。生成式几何内核。载于 2011 年 ACM SIGPLAN 部分求值与程序操纵研讨会论文集，PEPM 2011，2011 年 25，24 53 62 10 1145 1929501 1929510，美国得克萨斯州奥斯汀，1 月 - Siau-Cheng Khoo 和 Jeremy G. Siek（编）。ACM，– 。doi: . / . Tsung-Ju Chiang、Jeremy Yallop、Leo White 和 Ningning Xie。2024。使用模块函子进行分阶段编译。ACM 程序设计语言会议论文集 8，ICFP，文章 260（2024 年 8 月），35 页。doi:10.1145/3674649 Matthias Felleisen 和 Daniel P. Friedman。1987。控制运算符、SECD机与 λ 演算。载于《编程概念的形式化描述 III：IFIP TC 2/WG 2.2 编程概念形式化描述工作会议论文集 III》，丹麦埃贝鲁普，1986 年 8 月 25–28 日，Martin Wirsing（编）。North-Holland，193–222。Matthias Felleisen、Mitchell Wand、Daniel P. Friedman 和 Bruce F. Duba。1988。抽象延续：完全跳转处理的数学语义。载于 1988 年 ACM LISP 与函数式编程会议论文集，LFP 1988，美国犹他州斯诺伯德，1988 年 7 月 25–27 日，Jérôme Chailloux（编）。ACM，52–62。doi:10.1145/62678.62684 Matthew Flatt 和 R. Kent Dybvig。2020。延续标记的编译器与运行时支持。载于第 41 届 ACM SIGPLAN 编程语言设计与实现国际会议论文集，PLDI 2020，英国伦敦，2020 年 6 月 15–20 日，Alastair F. Donaldson 和 Emina Torlak（编）。ACM，45–58。doi:10.1145/3385412.3385981 Jun Inoue 和 Walid Taha。2012。多阶段程序推理。载于《编程语言与系统》，Helmut Seidl（编）。Springer Berlin Heidelberg，柏林、海德堡，357–376。Kanaru Isoda、Ayato Yokoyama 和 Yukiyoshi Kameyama。2024。使用代数效应与处理器实现类型安全代码生成。载于第 23 届 ACM SIGPLAN 生成式编程：概念与实践国际会议论文集（美国加利福尼亚州帕萨迪纳）（GPCE '24）。计算机协会，美国纽约州纽约市，53–65。doi:10.1145/3689484.3690731 Yukiyoshi Kameyama、Oleg Kiselyov 和 Chung chieh Shan。2015。用于非纯但卫生的代码生成的组合子。《计算机编程科学》112（2015），120–144。doi:10.1016/j.scico.2015.08.007 Yukiyoshi Kameyama、Oleg Kiselyov 和 Chung-chieh Shan。2011。转换阶段：使用分隔控制进行分阶段处理。《函数式编程杂志》21，6（2011），617–662。doi:10.1017/S0956796811000256 Oleg Kiselyov。2012。OCaml 中的分隔控制：抽象与具体。《理论计算机科学》435（2012），56–76。doi:10.1016/j.tcs.2012.02.025 函数式与逻辑编程。Oleg Kiselyov。2014。BER MetaOCaml 的设计与实现。载于《函数式与逻辑编程》，Michael Codish 和 Eijiro Sumii（编）。Springer International Publishing，Cham，86–102。Oleg Kiselyov。2024a。生成 C：异质元编程系统描述。《计算机编程科学》231（2024），103015。doi:10.1016/j.scico.2023.103015 Oleg Kiselyov。2024b。MetaOCaml：十年之后：系统描述。载于《函数式与逻辑编程：第 17 届国际研讨会 FLOPS 2024，日本熊本，2024 年 5 月 15–17 日，论文集》（日本熊本）。Springer-Verlag，柏林、海德堡，219–236。doi:10.1007/978-981-97-2300-3_12 Oleg Kiselyov、Aggelos Biboudis、Nick Palladinos 和 Yannis Smaragdakis。2017。流融合，直至完备。载于第 44 届 ACM SIGPLAN 编程语言原理研讨会论文集，POPL 2017，法国巴黎，2017 年 1 月 18–20 日，Giuseppe Castagna 和 Andrew D. Gordon（编）。ACM，285–299。doi:10.1145/3009837.3009880 Oleg Kiselyov、Yukiyoshi Kameyama 和 Yuto Sudo。2016。精化环境分类器。载于《编程语言与系统》，Atsushi Igarashi（编）。Springer International Publishing，Cham，271–291。Wiktor Kuchta。2023。效应处理器正规化的一个证明。https://icfp23.sigplan.org/details/hope-2023/4/A-proof-of-normalization-for-effect-handlers 美国华盛顿州西雅图。Julia L. Lawall 和 Olivier Danvy。1994。基于延续的部分求值。SIGPLAN Lisp Pointers VII，3（1994 年 7 月），227–238。doi:10.1145/182590.182483

Michael Lee、Ningning Xie、Oleg Kiselyov 和 Jeremy Yallop。2025年。处理作用域检查：动态作用域外逸检查的比较框架（制品）。Zenodo（2025年11月）。doi:10.5281/zenodo.17273738 Paul Blain Levy、John Power 和 Hayo Thielecke。2003年。值调用编程语言中的环境建模。Information and Computation 185, 2（2003），182–210。doi:10.1016/S0890-5401(03)00088-9 Yannis Lilis 和 Anthony Savidis。2019。元编程语言综述。ACM Comput. Surv. 52, 6，文章113（2019年10月），39页。doi:10.1145/3354584 Aleksandar Nanevski、Frank Pfenning 和 Brigitte Pientka。2008年。上下文模态类型理论。Transactions on Computational Logic 9, 3（2008年6月），23:1–49。Georg Ofenbeck、Tiark Rompf 和 Markus Püschel。2016年。RandIR：嵌入式编译器的差分测试。载于第7届 ACM SIGPLAN Scala 研讨会论文集，SCALA@SPLASH 2016（荷兰阿姆斯特丹），Aggelos Biboudis、Manohar Jonnalagedda、Sandro Stucki 和 Vlad Ureche（编）。ACM，21–30。doi:10.1145/2998392 Lionel Emile Vincent Parreaux。2020年。用于设计高级语言高效系统的类型安全元编程与编译技术。博士学位论文。洛桑联邦理工学院。Gordon Plotkin 和 Ningning Xie。2025年。处理选择单子。ACM 程序设计语言会议论文集 9，PLDI（2025）。doi:10.1145/3729321 Matija Pretnar。2015年。代数效应与处理器导论。特邀教程论文。Electronic Notes in Theoretical Computer Science 319（2015），19–35。doi:10.1016/j.entcs.2015.12.003 第31届程序语义数学基础会议（MFPS XXXI）。Tiark Rompf 和 Nada Amin。2015年。函数式珍珠：用500行代码实现 SQL 到 C 的编译器。载于第20届 ACM SIGPLAN 国际函数式编程会议论文集，ICFP 2015，3，2015，1 2 9 10 1145 2784731 2784760，加拿大不列颠哥伦比亚省温哥华，9月——Kathleen Fisher 和 John H. Reppy（编）。ACM，–。doi：。/。Gabriel Scherer。2017年。利用和类型与空类型判定等价性。载于第44届 ACM SIGPLAN 编程语言原理研讨会论文集（法国巴黎）（POPL '17）。计算机协会，美国纽约州纽约市，374–386。doi:10.1145/3009837.3009901 Tim Sheard 和 Simon Peyton Jones。2002年。用于 Haskell 的模板元编程。载于2002年 ACM SIGPLAN Haskell 研讨会论文集（宾夕法尼亚州匹兹堡）（Haskell '02）。计算机协会，美国纽约州纽约市，1–16。doi:10.1145/581690.581691 KC Sivaramakrishnan、Stephen Dolan、Leo White、Tom Kelly、Sadiq Jaffer 和 Anil Madhavapeddy。2021年。为 OCaml 改造效应处理器。载于第42届 ACM SIGPLAN 编程语言设计与实现国际会议论文集（线上，加拿大）（PLDI 2021）。计算机协会，美国纽约州纽约市，206–221。doi:10.1145/3453483.3454039 Nicolas Stucki、Aggelos Biboudis 和 Martin Odersky。2018年。多阶段编程与宏的实用统一。载于第17届 ACM SIGPLAN 生成式编程：概念与实践国际会议论文集，GPCE 2018，美国马萨诸塞州波士顿，2018年11月5日至6日，Eric Van Wyk 和 Tiark Rompf（编）。ACM，14–27。doi:10.1145/3278122.3278139 Eijiro Sumii 和 Naoki Kobayashi。2001年。在线与离线部分求值的混合方法。Higher-Order and Symbolic Computation 14, 2–3（2001年9月），101–142。Kedar Swadi、Walid Taha、Oleg Kiselyov 和 Emir Pašalić。2006年。通过单子方法避免分阶段处理记忆化函数时的代码重复。载于 PEPM（南卡罗来纳州查尔斯顿）。160–169。Walid Taha。1999年。多阶段编程：理论与应用。博士学位论文。瑞典哈尔姆斯塔德大学。https://urn.kb.se/resolve?urn=urn:nbn:se:hh:diva-15052 Walid Taha 和 Michael Florentin Nielsen。2003年。环境分类器。载于 POPL 2003 会议记录：第30届 SIGPLAN-SIGACT 编程语言原理研讨会，美国路易斯安那州新奥尔良，2003年1月15日至17日，26 37 10 1145 604131 604134——Alex Aiken 和 Greg Morrisett（编）。ACM，–。doi：。/。W. W. Tait。1967年。有限类型泛函的内涵解释 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 和 Tiark Rompf。2019年。揭示可微编程的奥秘：shift/reset 终极反向传播器。ACM 程序设计语言会议论文集 3，ICFP，文章96（2019年7月），31页。doi:10.1145/3341700 Edwin Westbrook、Mathias Ricken、Jun Inoue、Yilong Yao、Tamer Abdelatif 和 Walid Taha。2010年。Mint：使用弱可分离性的 Java 多阶段编程。载于第31届 ACM SIGPLAN 编程语言设计与实现会议论文集（加拿大安大略省多伦多）（PLDI '10）。计算机协会，美国纽约州纽约市，400–411。doi:10.1145/1806596.1806642 Ningning Xie、Matthew Pickering、Andres Löh、Nicolas Wu、Jeremy Yallop 和 Meng Wang。2022年。带类的分阶段编程：类型化模板 Haskell 的规范。ACM 程序设计语言会议论文集 6，POPL（2022），1–30。doi:10.1145/3498723 Ningning Xie、Leo White、Olivier Nicole 和 Jeremy Yallop。2023年。MacoCaml：可组合且可编译宏的分阶段处理。ACM 程序设计语言会议论文集 7，ICFP，文章209（2023年8月），45页。doi:10.1145/3607851

Jeremy Yallop。2017年。分阶段泛型编程。 *ACM 程序设计语言会议论文集*。 1，ICFP，文章29（2017年8月），29页。 doi:10.1145/3110273Jeremy Yallop 和社区贡献者。2025年。effects-bibliography：与计算效应理论和实践相关工作的协作式书目。GitHub 仓库。 https://github.com/yallop/effects-bibliography Jeremy Yallop 和 Oleg Kiselyov。2019年。生成相互递归定义。载于 *2019年 ACM SIGPLAN 部分求值与程序操纵研讨会论文集 （葡萄牙卡斯凯什） （PEPM 2019）*。计算机协会，美国纽约州纽约市，75–81。 doi:10.1145/3294032.3294078 Jeremy Yallop、Ningning Xie 和 Neel Krishnaswami。2023年。flap：带融合词法分析的确定性解析器。 *ACM 程序设计语言会议论文集*。 7，PLDI，文章155（2023年6月），24页。 doi:10.1145/3591269

A 矩阵乘法示例的扩展版本

我们通过以下 MacoCaml 代码来演示第2节中示例生成的代码。这里以这段代码作为贯穿示例，它定义了一个函数`f`，其函数体由调用`mmul`宏生成：

```MacoCaml
令 fa bc = $(mmul <<a>> <<b>> <<c>>)
```

对于列表 3，`mmul`会生成以下代码：

```MacoCaml
令 fa bc = 执行 i = 0 至 length a - 1 执行
             执行 k = 0 至 lengtha.(0)-1执行
               执行 j = 0 至 lengthb.(0)-1执行
                 c.i.j ← c.i.j + a.i.k * b.k.j
               done
             done
           done
```

对于列表 3 中插入 let 的变体，mmul 会生成以下代码：

```MacoCaml
let fa bc = 执行 i = 0 到 length a - 1 执行
              执行 k = 0 到 lengtha.(0)-1执行
                let e1 = a.i.k在
                  执行 j = 0 到lengthb.(0)-1执行
                    设e2 = b.k.j于
                      c.i.j ← c.i.j + a.i.k * b.k.j
                  done
                done
              done
```

注意，`Genlet`效应已被用于执行循环不变代码外提。然而，如果`mmul`有错误，因为程序员交换了循环，却没有修改 let 插入代码，那么生成的程序如下：

```MacoCaml
设 fa bc = 对于 i = 0 从 length a - 1 执行
             对于 j = 0 到 lengthb.(0)-1执行
               令 e2 = b.k.j在(*k 不在作用域内*)
                 对于 k = 0到lengtha.(0)-1执行
                   令e1 = a.i.k在
                     c.i.j ← c.i.j + a.i.k * b.k.j
                 done
               done
             done
```

第 3 行发生了作用域外逸。

此外，如果`mmul`存在错误，是因为程序员在执行 let 插入时识别错了循环，那么生成的程序将如下所示：

```
MacoCaml
let fa bc  = 执行 i  = 0 至 length a - 1 执行
                 lete1 = a.i.k in(*k 不在作用域内*)
                   执行 k  = 0 至 lengtha.(0)-1执行
                     令 e2 =b.k.j其中(*j 不在作用域内*)
                       对于j  = 0 到lengthb.(0)-1执行
                         c.i.j ← c.i.j  + a.i.k * b.k.j
                       done
                     done
                   done
```

其中第 2 行和第 4 行发生了作用域外逸。

B λ_{op}：效应与处理器的基础演算

λ_{op} 是一个带有深层效应处理器和多次续延的基础演算，大体上与以下演算相似：Pretnar [2015], 但 λ_{op} 将处理器视为独立的语法类别，使用泛型效应，在类型和语法层面区分函数与延续（如Isoda 等人 [2024]），并且为处理器提供多条类型规则（如 Biernacki 等人 [2017]）。列表 13展示了一个示例 λ_{op} 程序，其求值结果为22 （即 $1 +$ （$1 +$ （（$20 * 2$） $-$ （$10 * 2$））））。

```
λop
使用处理器执行 x ← accum(20) 其中执行 y ←accum(10) 其中返回 (
x − y)并处理 {accum(v, k) ↦ 执行 z ← 继续 k (v ∗ 2) 其中返回 1 + z}
```

列表 13. 一个返回 22 的程序。

图 14 展示了 λ_{op} 项和类型的语法，以及针对以下形式的配置给出的操作语义：$\langle c; E\rangle$ 其中项为 c，求值上下文为 E，采用Felleisen 和 Friedman [1987]的风格。求值上下文表示为由求值框架组成的栈 F，依照Kiselyov [2012]。

为清晰起见，我们用嵌套形式压缩多个栈帧。例如，我们将**执行** $x \longleftarrow$ **于** c_2 :: **执行** $y \leftarrow -$ **于** c_1写作**执行** $x \leftarrow$ (**执行** $y \leftarrow -$ **于** c_1) **于** c_2。

类型分为值类型（例如， $\mathbb{N}$）、计算类型（$\mathbb{N}$! **{打印}**）和处理器类型（$\mathbb{N}$! **{打印}** $\Longrightarrow \mathbb{N}$! $\emptyset$）。

效应签　名可以是递归的，因此 λ_{op} 支持编写不终止程序，例如：

```
λop
处理 (λ_.递归()) () 并使用 {递归(_,  k)  ↦ 继续 k (λ_.递归()) }
```

项的类型规则是标准的（图 15）。由于类型是分层的，类型判断也具有层次性：$\Gamma \vdash v :\ T$（值），$\Gamma \vdash c :\ T\,!\ \Delta$（计算），以及 $\Gamma \vdash h :\ S\,!\ \Delta_1 \Longrightarrow T\,!\ \Delta_2$（处理器）。一个封闭的计 p算若可以用一个效应集进行类型规 yp则判断，则称其类型规yp则判断为具有一个效 pty 应集。

定义 B.1（良类型闭计算）。 c 是一个闭的良类型计算，当且仅当 $\cdot \vdash c :\ T\,!\,\emptyset$

B.0.1 元理论。关于作用域外提的讨论建立在λ_{op}的一些元理论性质之上，这些性质由Bauer 和 Pretnar [2014]证明。

定理 B.1（进展）。若$\cdot \vdash E[c] : T\,!\ \Delta$，则以下情况之一成立

(1) c 的形式为**返回** v 且 $E = [-]$, *(2)* c 的形式为**操作**(v)，其中 $op \in \Delta$，且操作 $\notin$ 已处理 (E)*(3)* $\exists c', E'$，使得 $\langle c; E\rangle \rightarrow \langle c'; E'\rangle$

Theo引理 B.2（保持性）。若$\cdot \vdash E[c] : T\,!\ \Delta$且$c; E \rightarrow c'; E'$，则$\cdot \vdash E'[c'] : T\,!\ \Delta$

λ_{op}

语法

值	v	:=	$x \mid m \in \mathbb{N} \mid \lambda x.c \mid \kappa x.c$
计算	c	:=	$v_1\ v_2 \mid \textbf{返回}\ v$
			$\mid \textbf{执行}\ x \leftarrow c_1\ \textbf{中}\ c_2$
			$\mid \textbf{操作}(v)$
			$\mid \textbf{处理}\ c\ ,\ \textbf{使用}\ \{h\}$
			$\textbf{继续}\ v_1\ v_2$
处理器	h	:=	$\textbf{返回}(x)\ \mapsto c$
			$\mid h; \textbf{操作}(x,k)\ \mapsto c$

类型

效应集	$\Delta ::= \emptyset \mid \Delta \cup \{\text{操作}_i\}$
值类型	$S,T ::= \mathbb{N} \mid S \xrightarrow{\Delta} T \mid S \xrightarrow{\Delta\!\!\Rightarrow} T$
计算类型	$T\,!\,\Delta$
处理器类型	$S\,!\,\Delta_1 \Longrightarrow T\,!\,\Delta_2$

操作语义

辅助定义

求值框架 $F ::=$ 执行 $x \leftarrow [-]$ 于 $c_2 \mid$ 处理 $[-]$ 并使用 $\{h\}$ 求值上下文 $E ::= [-] \mid E[F]$处理器的定义域 $\mathrm{dom}(h) \triangleq \mathrm{dom}(\mathrm{return}(x) \mapsto c) = \emptyset$, $\mathrm{dom}(h; \text{操作}(x,k) \mapsto c) = \mathrm{dom}(h) \cup \{\mathrm{op}\}$ 已处理效应 $\mathrm{handled}(E) \triangleq \mathrm{handled}([-]) = \emptyset$, $\mathrm{handled}(E[\text{执行}\ x \leftarrow [-]\ \text{于}\ c_2]) = \mathrm{handled}(E)$, $\mathrm{handled}(E[\text{处理}\ [-]\ \text{并使用}\ \{h\}]) = \mathrm{handled}(E) \cup \mathrm{dom}(h)$,

归约规则

$$
\begin{array}{lrcll}
\text{(Red-App)} & (\lambda x.c)v;\ E & \longrightarrow & c[v/x];\ E & \\
\text{(Red-Seq)} & \langle \textbf{执行}\ x \leftarrow \textbf{返回}\ v\ \textbf{中}\ c; E\rangle & \longrightarrow & c[v/x];\ E & \\
\text{(Red-Hdl)} & \langle \textbf{处理器返回}\ v\ \textbf{与}\ \{h\}; E\rangle & \longrightarrow & c[v/x];\ E & (\text{其中}\ \textbf{返回}(x) \mapsto c \in h) \\
\text{(Cng-Psh)} & F[c];\ E & \longrightarrow & c;\ E[F] & \\
\text{(Cng-Pop)} & \langle \textbf{返回}\ v; E[F]\rangle & \longrightarrow & \langle F[\textbf{返回}\ v]; E\rangle & \\
\text{(Eff-Op)} & \langle \textbf{操作}(v);\ E_1[\textbf{处理}\ E_2\ \textbf{with}\ \{h\}]\rangle & \longrightarrow & \langle c[v/x,\ \kappa x.\textbf{处理}\ E_2[\textbf{返回}\ x]\ \textbf{with}\ \{h\}/k];\ E_1\rangle & (\text{其中}\ \textbf{操作}\in \mathrm{dom}(h)\ \text{且}\ \textbf{操作}\notin \text{已处理}(E_2)) \\
\text{(效应续延)} & \langle \textbf{继续}\ (\kappa x.E_2[\textbf{返回}\ x])\ v;\ E_1\rangle & \longrightarrow & \langle \textbf{返回}\ v;\ E_1[E_2]\rangle &
\end{array}
$$

图 14. λ_{op}：语法、类型、操作语义。

推论 B.2（类型安全）。若 $\cdot \vdash c : T\,!\,\emptyset$ 则以下情况至少有一种成立：

(1) $\langle c; [-]\rangle \rightarrow^{\omega}$（不终止）*(2)* $\langle c; [-]\rangle \rightarrow^{*} \langle \textbf{返回}\ v; [-]\rangle$

C 完整 $\lambda_{\langle\langle \mathsf{op}\rangle\rangle}$ 规则

为方便起见，图 16 重复列出了 $\lambda_{\langle\langle \mathsf{op}\rangle\rangle}$ 的语法和类型，这些内容已在图 2中给出。图 17给出了 $\lambda_{\langle\langle \mathsf{op}\rangle\rangle}$的完整类型规则。选定规则已在图 4 (§3.1.1)中给出。

D 完整 $\lambda_{\mathsf{AST(op)}}$ 规则

为方便起见，图 18 重复列出了图 5中的语法和类型。图 19 和20 给出了 $\lambda_{\mathsf{AST(op)}}$的完整操作语义和完整类型规则，从而补全了图 5(§3.2)中介绍的子集。

E 完整 elaboration 规则

图 21 展示了从 $\lambda_{\langle\langle \mathsf{op}\rangle\rangle}$ 到 $\lambda_{\mathsf{AST(op)}}$的完整项展开。选定规则见 图 7 (§3.3)。

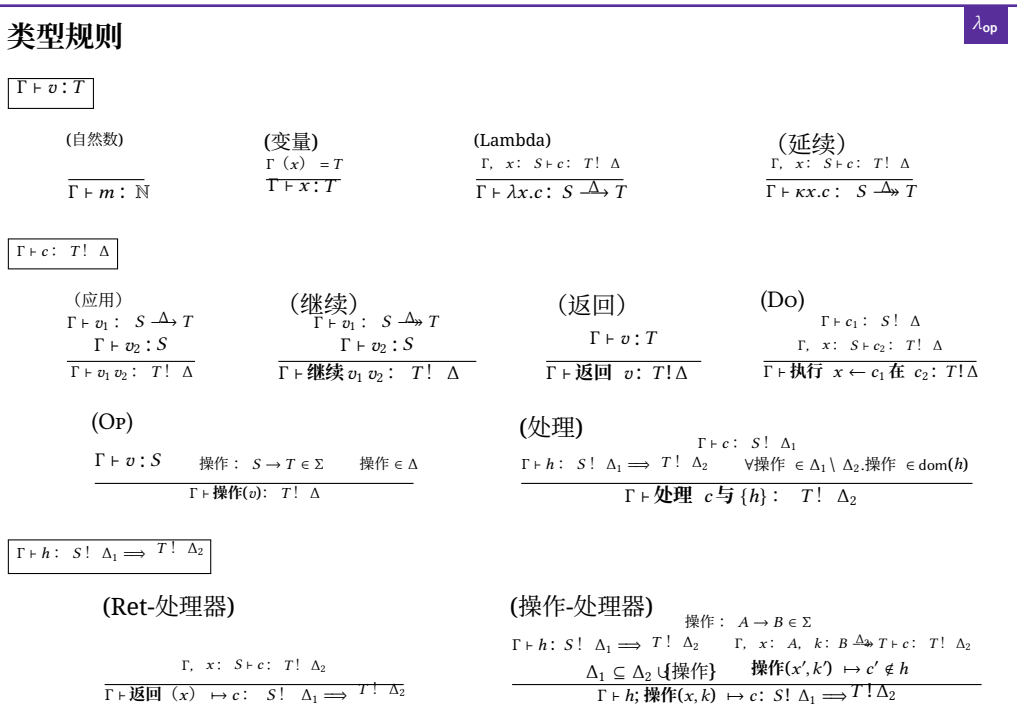


图 15. λ_{op} 类型规则

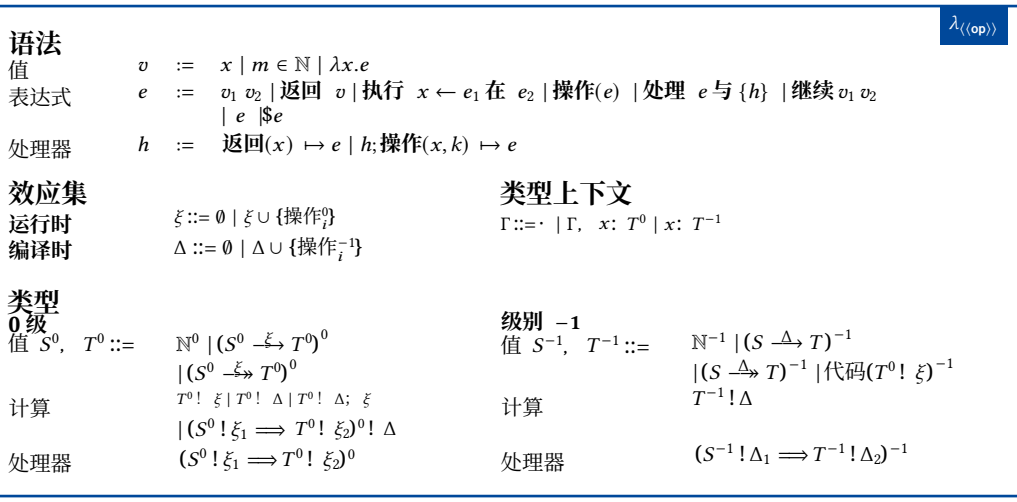


图 16. $\lambda_{\langle\langle \text{op} \rangle\rangle}$ 语法和类型（重复）

F 精化环境分类器的完整规则

图 22 和 23 展示了 c|q-类型规则和 s-类型规则，适用于 $\lambda_{\langle\langle \text{op} \rangle\rangle^\gamma}$。选定规则见 图 12(§5.1)。

类型规则

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

类型上的层级注释大多省略

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

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

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

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

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

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

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

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

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

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

(处理)
$$\frac{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} e : S \,!\, \Delta;\ \xi_1 \qquad \Gamma \vdash_{\mathsf{c}|\mathsf{q}} h : S \,!\, \xi_1 \Longrightarrow T \,!\, \xi_2 \,!\, \Delta \qquad \forall \text{操作} \in \xi_1 \setminus \xi_2 . \text{操作} \in \mathrm{dom}(h)}{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} \textbf{处理}\ e, \textbf{使用}\ \{h\} : T \,!\, \Delta;\ \xi_2}$$

(拼接)
$$\frac{\Gamma \vdash_{\mathsf{s}} e : \text{代码}\ (T^0 \,!\, \xi)^{-1} \,!\, \Delta}{\Gamma \vdash_{\mathsf{c}|\mathsf{q}} \$e : T^0 \,!\, \Delta;\ \xi}$$

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

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

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

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

(S-自然数)
$$\frac{}{\Gamma \vdash_{\mathsf{s}} m : \mathbb{N}}$$

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

(S-Lambda)
$$\frac{\Gamma,\ x : S \vdash_{\mathsf{s}} e : T \,!\, \Delta}{\Gamma \vdash_{\mathsf{s}} \lambda x.e : (S \xrightarrow{\Delta} T)}$$

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

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

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

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

(S-返回)
$$\frac{\Gamma \vdash_{\mathsf{s}} v : T}{\Gamma \vdash_{\mathsf{s}} \textbf{返回}\ v : T \,!\, \Delta}$$

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

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

(S-处理)
$$\frac{\Gamma \vdash_{\mathsf{s}} e : S \,!\, \Delta_1 \qquad \Gamma \vdash_{\mathsf{s}} h : (S \,!\, \Delta_1 \Longrightarrow T \,!\, \Delta_2) \qquad \forall \text{操作} \in \Delta_1 \setminus \Delta_2 . \text{操作} \in \mathrm{dom}(h)}{\Gamma \vdash_{\mathsf{s}} \textbf{处理}\ e\ \textbf{with}\ \{h\} : T \,!\, \Delta_2}$$

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

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

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

(S-操作处理器)
$$\frac{\Gamma \vdash_{\mathsf{s}} h : S \,!\, \Delta_1 \Longrightarrow T \,!\, \Delta_2 \qquad \Gamma, x : A, k : (B \xrightarrow{\Delta\!\!\!\!\rightarrow} T) \vdash_{\mathsf{s}} e : T \,!\, \Delta_2 \qquad \text{操作} : A \to B \in \Sigma \qquad \Delta_1 \subseteq \Delta_2 \cup \{\text{操作}\} \qquad \textbf{操作}(x', k') \mapsto e' \notin h}{\Gamma \vdash_{\mathsf{s}} h; \textbf{操作}(x, k) \mapsto e : S \,!\, \Delta_1 \Longrightarrow T \,!\, \Delta_2}$$

图 17. $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$的类型规则

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

语法

形式参数 α_R

范式

$$n ::= x \mid m \in \mathbb{N} \mid \lambda x.t \mid \kappa x.t \mid \text{自然数}(m) \mid \alpha_R \mid \text{变量}(\alpha_R) \mid \text{Lam}(n_1,\ n_2) \mid \text{应用}(n_1,\ n_2) \mid \text{继续}(n_1, n_2) \mid \text{Ret}(n) \mid \text{执行}(n_1, n_2, n_3) \mid \text{操作}(n) \mid \text{Hwith}(n_1, n_2) \mid \text{Hret}(n_1,\ n_2)\ (n_1,\ n_2) \mid \text{Hop}(n_1,\ n_2,\ n_3,\ n_4)$$

项

$$t := n_1\ n_2 \mid \textbf{返回}\ n \mid \textbf{执行}\ x \leftarrow t_1\ \textbf{in}\ t_2 \mid \textbf{操作}(t) \mid \textbf{处理}\ n\ \textbf{with}\ \{h\} \mid \textbf{继续}\ n_1\ n_2 \mid \textbf{check}\ n \mid \textbf{Check}_{\mathsf{M}}\ n \mid \textbf{mkvar}\ R \mid \textbf{dlet}(n, t) \mid \textbf{tls}(t) \mid \textbf{err}$$

处理器

$$h := \textbf{返回}\ (x)\ \mapsto t \mid \textbf{操作}\ (x,\ k)\ \mapsto t$$

类型上下文

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

类型

运行时预类型

效应集 $\xi ::= \emptyset \mid \xi \cup \{\mathsf{op}_i\}$

值类型 $Q,\ R ::= \mathbb{N} \mid Q \xrightarrow{\xi} R \mid Q \xrightarrow{\xi}\!\!\!\!\rightarrow R$

计算类型 $R\,!\,\xi$

处理器类型 $Q\,!\,\xi_1 \Longrightarrow R\,!\,\xi_2$

类型

效应集 $\Delta ::= \emptyset \mid \Delta \cup \{\text{操作}_i\}$

值类型 $S,\ T ::= \ldots \mid \text{FParam}(R) \mid \text{抽象语法树}(R) \mid \text{抽象语法树}(R\,!\,\xi) \mid \text{抽象语法树}(Q\,!\,\xi_1 \Longrightarrow R\,!\,\xi_2)$

计算类型 $T\,!\,\Delta$

处理器类型 $S\,!\,\Delta \Longrightarrow T\,!\,\Delta_2$

图 18. $\lambda_{\mathsf{AST(op)}}$：语法和类型（重复）

G 擦除函数

擦除函数接收一个 0 级类型，擦除所有层级注释，并展开效应行。它通过对 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ 类型进行直接归纳来定义。

$$\begin{aligned}
\text{erase}(\mathbb{N}^0) &= \mathbb{N} \\
\text{erase}((S^0 \xrightarrow{\xi} T^0)^0) &= \text{erase}(S^0) \xrightarrow{[\![\xi]\!]} \text{erase}(T^0) \\
\text{erase}((S^0 \xrightarrow{\xi}\!\!\!\!\rightarrow T^0)^0) &= \text{erase}(S^0) \xrightarrow{[\![\xi]\!]}\!\!\!\!\rightarrow \text{erase}(T^0)
\end{aligned}$$

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

$$\text{erase}((S^0\,!\,\xi_1 \Longrightarrow T^0\,!\,\xi_2)^0) = \text{erase}(S^0)\,!\,[\![\xi_1]\!] \Longrightarrow \text{erase}(T^0)\,!\,[\![\xi_2]\!]$$

请注意，对编译时为值（无编译时效应集 Δ）的 0 级类型擦除层级注释后，会生成一个运行时前类型。

操作语义

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

辅助定义

求值框架 $F ::= \mathbf{执行}\ x \leftarrow [-]\ \mathbf{在}\ t_2 \mid \mathbf{处理}\ [-]\ \mathbf{使用}\ \{h\} \mid \mathbf{DLet}(\alpha_R,\ [-]) \mid \mathtt{tls}([-])$

求值上下文 $E ::= [-] \mid E[F]$

处理器的定义域 $\text{定义域}(h) \triangleq$
$\text{定义域}(\mathbf{返回}(x) \mapsto t) = \emptyset,$
$\text{定义域}(h; \mathbf{操作}(x,\ k) \mapsto t) = \text{定义域}(h) \cup \{\mathbf{操作}\}$

已处理效应 $\text{已处理}(E) \triangleq$
$\text{已处理}([-]) = \emptyset,$
$\text{已处理}(E[\mathbf{执行}\ x \leftarrow [-]\ \mathbf{于}\ t_2]) = \text{已处理}(E),$
$\text{已处理}(E[\mathbf{处理}\ [-]\ \mathbf{配合}\ \{h\}]) = \text{已处理}(E) \cup \text{dom}(h),$
$\text{已处理}(E[\mathbf{DLet}(\alpha_R,\ [-])]) = \text{已处理}(E),$
$\text{已处理}(E[\mathtt{tls}([-])]) = \text{已处理}(E)$

归约规则

与静音和取消静音相关的机制已突出显示

(Red-应用)
$(\lambda x.t)n;\ E; U;\ M;\ I \longrightarrow t[n/x];\ E; U;\ M;\ I$
(红-序列)
$\langle \mathbf{执行}\ x \leftarrow \mathbf{返回}\ n \mathbf{在}\ t;\ E;\ U;\ M;\ I\rangle \longrightarrow \langle t[n/x];\ E;\ U;\ M;\ I\rangle$
(Red-Hdl)
$\langle \mathbf{处理器返回}\ n \mathbf{with}\ \{h\}; E; U; M; I\rangle \longrightarrow \langle t[n/x]; E; U; M; I\rangle$
其中 $\mathbf{返回}(x) \mapsto t \in h$
(Cng-Psh)
$F[t];\ E; U;\ M;\ I \longrightarrow t;\ E[F]; U;\ M; I$
(Cng-Pop)
$\langle \mathbf{返回}\ n; E[F]; U; M; I\rangle \longrightarrow \langle F[\mathbf{返回}\ n]; E; U; M; I\rangle$
(Ast-Gen)
$\mathbf{mkvar}\ R;\ E; U;\ M;\ I \longrightarrow \mathbf{返回}\ \alpha_R;\ E; U \cup \{\alpha\};\ M;\ I$
其中 $\alpha = next(U),\ next(U) \notin U$ next 确定性
(Sec-Chs)
$\langle \mathbf{检查}\ n; E; U; M; I\rangle \longrightarrow \langle \mathbf{返回}\ n; E; U; M; I\rangle$
如果 $FV^0(n) \not\subseteq \pi_{\mathsf{Var}}(E)$
(Sec-Chf)
$\mathbf{check}\ n;\ E; U;\ M;\ I \longrightarrow \mathtt{err};\ E; U;\ M; I$
如果 $FV^0(n) \not\subseteq \pi_{\mathsf{Var}}(E)$
(Sec-Dlt)
$\langle \mathtt{dlet}(\alpha_R,\ \mathbf{返回}\ n);\ E;\ U;\ M;\ I\rangle \longrightarrow \langle \mathbf{返回}\ n;\ E;\ U;\ M';\ I'\rangle$
若 $len(E) > I$ 则 $M' = M,\ I' = I,$ 否则 $M' = \emptyset,\ I' = \top$
(Eff-Op)
$\mathbf{操作}(v);\ E_1[\mathbf{处理}\ E_2, \mathbf{使用}\ \{h\}];\ U;\ M;\ I \longrightarrow c[v/x, \mathbf{cont}/k];\ E_1;\ U;\ M \cup \pi_{\mathsf{Var}}(E_2);\ I'$
其中 $cont = \kappa x.\ \mathbf{处理}\ E_2[\mathbf{返回}\ x]\ \mathbf{使用}\ \{h\}$ 以及 $\mathbf{操作}(x,\ k) \mapsto c \in h$ 以及 操作 $\notin$ 已处理(E_2) 以及 $I' = \text{最小值}(\text{长度}(E_1)\ I)$
(效应续延)
$\langle \mathbf{继续}\ (\kappa x.E_2[\mathbf{返回}\ x])\ n; E_1; U; M; I\rangle \longrightarrow \langle \mathbf{返回}\ n; E_1[E_2]; U; M; I\rangle$
(Sec-Tls)
$\langle \mathtt{tls}(\mathbf{返回}\ n); E; U; M; I\rangle \longrightarrow \langle \mathbf{返回}\ n; E; U; \emptyset; \top\rangle$
(Sec-Cms)
$\mathbf{Check}_M\ n;\ E;\ U;\ M;\ I \longrightarrow \mathbf{返回}\ n;\ E;\ U;\ M;\ I$
如果 $FV^0(n) \setminus M \subseteq \pi_{\mathsf{Var}}(E)$
(Sec-Cmf)
$\mathbf{Check}_M\ n;\ E;\ U;\ M;\ I \longrightarrow \mathtt{err};\ E;\ U;\ M;\ I$
如果 $FV^0(n) \setminus M \not\subseteq \pi_{\mathsf{Var}}(E)$

图 19。 $\lambda_{\mathsf{AST(op)}}$的操作语义

类型规则

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

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

(自然数)
$$\frac{}{\Gamma \vdash m : \mathbb{N}}$$

(变量)
$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}$$

(Lambda)
$$\frac{\Gamma,\ x :\ S \vdash t :\ T\ !\ \Delta}{\Gamma \vdash \lambda x.t :\ S \xrightarrow{\Delta} T}$$

(延续)
$$\frac{\Gamma,\ x :\ S \vdash t :\ T\ !\ \Delta}{\Gamma \vdash \kappa x.t :\ S \xrightarrow{\Delta\!\!\!\!\rightarrow} T}$$

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

(Nat-AST)
$$\frac{}{\Gamma \vdash \text{自然数}(m) : \text{抽象语法树}(\mathbb{N})}$$

(Var-AST)
$$\frac{\Gamma \vdash n : \mathsf{FParam}(R)}{\Gamma \vdash \text{变量}(n) : \text{抽象语法树}(R)}$$

(Lambda-AST)
$$\frac{\Gamma \vdash n_1 : \mathsf{FParam}(Q) \quad \Gamma \vdash n_2 : \text{抽象语法树}(R\ !\ \xi)}{\Gamma \vdash \mathsf{Lam}(n_1, n_2) : \text{抽象语法树}(Q \xrightarrow{\xi} R)}$$

(应用-抽象语法树)
$$\frac{\Gamma \vdash n_1 : \text{抽象语法树}(Q \xrightarrow{\xi} R) \quad \Gamma \vdash n_2 : \text{抽象语法树}(Q)}{\Gamma \vdash \text{应用}(n_1, n_2) : \text{抽象语法树}(R\ !\ \xi)}$$

(继续-抽象语法树)
$$\frac{\Gamma \vdash n_1 : \text{抽象语法树}(Q \xrightarrow{\xi\!\!\!\!\rightarrow} R) \quad \Gamma \vdash n_2 : \text{抽象语法树}(Q)}{\Gamma \vdash \text{继续}(n_1,\ n_2) : \text{抽象语法树}(R\ !\ \xi)}$$

(返回-抽象语法树)
$$\frac{\Gamma \vdash n : \text{抽象语法树}(R)}{\Gamma \vdash \mathsf{Ret}(n) : \text{抽象语法树}(R\ !\ \xi)}$$

(执行-抽象语法树)
$$\frac{\Gamma \vdash n_1 : \text{抽象语法树}(Q\ !\ \xi) \quad \Gamma \vdash n_2 : \mathsf{FParam}(Q) \quad \Gamma \vdash n_3 : \text{抽象语法树}(R\ !\ \xi)}{\Gamma \vdash \text{执行}(n_1, n_2, n_3) : \text{抽象语法树}(R\ !\ \xi)}$$

(操作-抽象语法树)
$$\frac{\text{操作} : Q \to R \in \Sigma \quad \text{操作} \in \xi \quad \Gamma \vdash n : \text{抽象语法树}(Q)}{\Gamma \vdash \text{操作}(n_1) : \text{抽象语法树}(R\ !\ \xi)}$$

(处理-抽象语法树)
$$\frac{\Gamma \vdash n_1 : \text{抽象语法树}(Q\ !\ \xi_1) \quad \Gamma \vdash h : \text{抽象语法树}(Q\ !\ \xi_1 \Longrightarrow\ R\ !\ \xi_2) \quad \Gamma \vdash n : \text{抽象语法树}(Q) \quad \forall \text{操作} \in \xi_1 \setminus \xi_2.\text{操作} \in \mathsf{dom}(n_2)}{\Gamma \vdash \mathsf{Hwith}(n_1, n_2) : \text{抽象语法树}(R\ !\ \xi_2)}$$

(Ret-Handler-AST)
$$\frac{\Gamma \vdash n_1 : \mathsf{FParam}(Q) \quad \Gamma \vdash n_2 : \text{抽象语法树}(R\ !\ \xi_2)}{\Gamma \vdash \mathsf{Hret}(n_1, n_2) : \text{抽象语法树}(Q\ !\ \xi_1 \Longrightarrow\ R\ !\ \xi_2)}$$

(Op-Handler-AST)
$$\frac{\text{操作} :\ A \to B \in \Sigma \quad \Gamma \vdash n_1 :\ \text{抽象语法树}(Q\ !\ \xi_1 \Longrightarrow\ R\ !\ \xi_2) \quad \Gamma \vdash n_2 :\ \mathsf{FParam}(A) \quad \Gamma \vdash n_3 :\ \mathsf{FParam}(B \xrightarrow{\xi_2\!\!\!\!\rightarrow} R) \quad \Gamma \vdash n_4 :\ \text{抽象语法树}(R\ !\ \xi_2) \quad \xi_1 \subseteq \xi_2 \cup \{\text{操作}\}}{\Gamma \vdash \mathsf{Hop}(n_1,\ n_2,\ n_3,\ n_4) :\ \text{抽象语法树}(Q\ !\ \xi_1 \Longrightarrow\ R\ !\ \xi_2)}$$

$\boxed{\Gamma \vdash t :\ T\ !\ \Delta}$

(应用)
$$\frac{\Gamma \vdash v_1 :\ S \xrightarrow{\Delta} T \quad \Gamma \vdash v_2 :\ S}{\Gamma \vdash v_1\ v_2 :\ T\ !\ \Delta}$$

(继续)
$$\frac{\Gamma \vdash v_1 :\ S \xrightarrow{\Delta\!\!\!\!\rightarrow} T \quad \Gamma \vdash v_2 :\ S}{\Gamma \vdash \textbf{继续}\ v_1\ v_2 :\ T\ !\ \Delta}$$

(返回)
$$\frac{\Gamma \vdash v : T}{\Gamma \vdash \textbf{返回}\ \ v :\ T\ !\ \Delta}$$

(Do)
$$\frac{\Gamma \vdash c_1 :\ S\ !\ \Delta \quad \Gamma,\ x :\ S \vdash c_2 :\ T\ !\ \Delta}{\Gamma \vdash \textbf{执行}\ \ x \leftarrow c_1\ \textbf{在}\ \ c_2 :\ T\ !\ \Delta}$$

(Op)
$$\frac{\Gamma \vdash v : S \quad \text{操作} : S \to T \in \Sigma \quad \text{操作} \in \Delta}{\Gamma \vdash \textbf{操作}(v) :\ T\ !\ \Delta}$$

(处理)
$$\frac{\Gamma \vdash c :\ S\ !\ \Delta_1 \quad \Gamma \vdash h :\ S\ !\ \Delta_1 \Longrightarrow T\ !\ \Delta_2 \quad \forall \text{操作} \in \Delta_1 \setminus \Delta_2.\text{操作} \in \mathsf{dom}(h)}{\Gamma \vdash \textbf{处理}\ \ c\ \textbf{with}\ \{h\} :\ \ T\ !\ \Delta_2}$$

(tls)
$$\frac{\Gamma \vdash t :\ T\ !\ \Delta}{\Gamma \vdash \textbf{tls}(t) :\ T\ !\ \Delta}$$

(DLet)
$$\frac{\Gamma \vdash n :\ \mathsf{FParam}(R) \quad \Gamma \vdash t :\ T\ !\ \Delta}{\Gamma \vdash \textbf{dlet}\ (n,\ t) :\ T\ !\ \Delta}$$

(Err)
$$\frac{}{\Gamma \vdash \textbf{err} :\ T\ !\ \Delta}$$

(mkvar)
$$\frac{}{\Gamma \vdash \textbf{mkvar}\ R :\ \mathsf{FParam}(R)\ !\ \Delta}$$

(Check)
$$\frac{\Gamma \vdash n : T \quad T \text{ 的 AST 类型}}{\Gamma \vdash \textbf{check}\ n :\ T\ !\ \Delta}$$

(Check-M)
$$\frac{\Gamma \vdash n : T \quad T \text{ 属于 AST 类型}}{\Gamma \vdash \textbf{check}_{\mathsf{M}}\ n :\ T\ !\ \Delta}$$

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

(Ret-处理器)
$$\frac{\Gamma,\ x :\ S \vdash c :\ T\ !\ \Delta_2}{\Gamma \vdash \textbf{返回}(x)\ \mapsto c :\ \ S\ !\ \ \Delta_1 \Longrightarrow T\ !\ \Delta_2}$$

(操作-处理器)
$$\frac{\text{操作} :\ A \to B \in \Sigma \quad \Gamma \vdash h :\ S\ !\ \Delta_1 \Longrightarrow\ T\ !\ \Delta_2 \quad \Gamma,\ x :\ A,\ k :\ B \xrightarrow{\Delta_2\!\!\!\!\rightarrow} T \vdash c :\ T\ !\ \Delta_2 \quad \Delta_1 \subseteq \Delta_2 \cup \{\text{操作}\} \quad \textbf{操作}(x', k') \mapsto c' \notin h}{\Gamma \vdash h;\ \textbf{操作}(x,\ k)\ \mapsto c :\ S\ !\ \Delta_1 \Longrightarrow\ T\ !\ \Delta_2}$$

图 20. $\lambda_{\mathsf{AST(op)}}$ 类型规则

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

$[\![m]\!]_{\mathbf{c}\|\mathbf{q}}$	$=$ **返回** 自然数(m)
$[\![x]\!]_{\mathbf{c}\|\mathbf{q}}$	$=$ **返回** 变量(x)
$[\![\lambda x\colon T^0.\, e]\!]_{\mathbf{c}\|\mathbf{q}}$	$=$ **执行** $x \leftarrow$ **mkvar** 擦除(T^0) **于执行** 主体 $\leftarrow [\![e]\!]_{\mathbf{c}\|\mathbf{q}}$ **于返回** $\mathtt{Lam}(x,$ 主体$)$
$[\![v_1\, v_2]\!]_{\mathbf{c}\|\mathbf{q}}$	$=$ **执行** $f \leftarrow [\![v_1]\!]_{\mathbf{c}\|\mathbf{q}}$ **于执行** $\mathtt{a} \leftarrow [\![v_2]\!]_{\mathbf{c}\|\mathbf{q}}$ **于返回** 应用$(f,\ a)$
$[\![$**返回** $v]\!]_{\mathbf{c}\|\mathbf{q}}$	$=$ **执行** $a \leftarrow [\![v]\!]_{\mathbf{c}\|\mathbf{q}}$ **于返回** $\mathtt{Ret}(a)$
$[\![$**执行** $x\colon T^0 \leftarrow e_1$ **于** $e_2]\!]_{\mathbf{c}\|\mathbf{q}}$	$=$ **执行** $a \leftarrow [\![e_1]\!]_{\mathbf{c}\|\mathbf{q}}$ **于执行** $x \leftarrow$ **mkvar** 擦除(T^0) **于执行** $b \leftarrow [\![e_2]\!]$ **于返回** $\mathtt{Do}(x,\ a,\ b)$
$[\![$**操作**$(v)]\!]_{\mathbf{c}\|\mathbf{q}}$	$=$ **执行** $a \leftarrow [\![v]\!]_{\mathbf{c}\|\mathbf{q}}$ **在返回中** 操作(a)
$[\![$**处理** e **使用** $\{h\}]\!]_{\mathbf{c}\|\mathbf{q}}$	$=$ **执行** $a \leftarrow [\![e]\!]_{\mathbf{c}\|\mathbf{q}}$ **在执行** $b \leftarrow [\![h]\!]_{\mathbf{c}\|\mathbf{q}}$ **在返回** 使用处理器$(a,\ b)$
$[\![$**返回**$(x\colon T^0) \mapsto e]\!]_{\mathbf{c}\|\mathbf{q}}$	$=$ **执行** $x \leftarrow$ **mkvar** 擦除(T^0) **在执行** $a \leftarrow [\![e]\!]_{\mathbf{c}\|\mathbf{q}}$ **在返回** 处理返回值$(x,\ a)$
$[\![h;$ **操作**$(x\colon T^0,\ k\colon A^0 \xrightarrow{\xi} B^0) \mapsto e]\!]_{\mathbf{c}\|\mathbf{q}}$	$=$ **执行** $a \leftarrow [\![h]\!]_{\mathbf{c}\|\mathbf{q}}$ **在 执行** $x \leftarrow$ **mkvar** 擦除(T^0) **在 执行** $k \leftarrow$ **mkvar** 擦除$(A^0 \xrightarrow{\xi} B^0)$ **在 执行** $b \leftarrow [\![e]\!]_{\mathbf{c}\|\mathbf{q}}$ **在返回** $\mathtt{Hop}(a,\ x,\ k,\ b)$
$[\![\$e]\!]_{\mathbf{c}}$	$=$ $\mathtt{tls}([\![e]\!]_{\mathbf{c}})$
$[\![\$e]\!]_{\mathbf{q}}$	$=$ $[\![e]\!]_{\mathbf{q}}$
$[\![x]\!]_{\mathbf{s}}$	$= x$
$[\![\lambda x\colon T^0.\, e]\!]_{\mathbf{s}}$	$= \lambda x.\, [\![e]\!]_{\mathbf{s}}$
$[\![v_1\, v_2]\!]_{\mathbf{s}}$	$= [\![v_1]\!]_{\mathbf{s}}\, [\![v_2]\!]_{\mathbf{s}}$
$[\![$**返回** $v]\!]_{\mathbf{s}}$	$=$ **返回** $[\![v]\!]_{\mathbf{s}}$
$[\![$**执行** $x \leftarrow e_1$ **在** $e_2]\!]_{\mathbf{s}}$	$=$ **执行** $x \leftarrow [\![e_1]\!]_{\mathbf{s}}$ **在** $[\![e_2]\!]_{\mathbf{s}}$
$[\![$**操作**$(v)]\!]_{\mathbf{s}}$	$=$ **操作**$([\![v]\!]_{\mathbf{s}})$
$[\![$**处理** e **与** $\{h\}]\!]_{\mathbf{s}}$	$=$ **处理** $[\![e]\!]_{\mathbf{s}}$ **使用** $\{[\![h]\!]_{\mathbf{s}}\}$
$[\![$**返回**$(x) \mapsto e]\!]_{\mathbf{s}}$	$=$ **返回**$(x) \mapsto [\![e]\!]_{\mathbf{s}}$
$[\![h;$**操作**$(x, k) \mapsto e]\!]_{\mathbf{s}}$	$= [\![h]\!]_{\mathbf{s}};$**操作**$(x, k) \mapsto [\![e]\!]_{\mathbf{s}}$
$[\![\quad e \quad]\!]_{\mathbf{s}}$	$= [\![e]\!]_{\mathbf{q}}$

图 21. 从 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle}$ 到 $\lambda_{\mathsf{AST(op)}}$的完整项展开

c| q-类型规则类型上的层级注释

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

大多省略

$$\boxed{\Gamma \vdash^{\gamma}_{c|q} v : T^0 ! \Delta}$$

(自然数)
$$\frac{}{\Gamma \vdash^{\gamma}_{c|q} m : \mathbb{N} ! \Delta}$$

(变量)
$$\frac{(x : T^0)^{\gamma} \in \Gamma}{\Gamma \vdash^{\gamma}_{c|q} x : T^0 ! \Delta}$$

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

(子值)
$$\frac{\Gamma \vdash^{\gamma'}_{c|q} v : T ! \Delta \qquad \Gamma \vDash \gamma' \sqsubseteq \gamma}{\Gamma \vdash^{\gamma}_{c|q} v : T ! \Delta}$$

$$\boxed{\Gamma \vdash^{\gamma}_{c|q} e : T^0 ! \Delta; \xi}$$

(应用)
$$\frac{\Gamma \vdash^{\gamma}_{c|q} v_1 : (S \xrightarrow{\xi} T) ! \Delta \qquad \Gamma \vdash^{\gamma}_{c|q} v_2 : S ! \Delta}{\Gamma \vdash^{\gamma}_{c|q} v_1 v_2 : T ! \Delta; \xi}$$

(继续)
$$\frac{\Gamma \vdash^{\gamma}_{c|q} v_1 : (S \xrightarrow{\xi} T) ! \Delta \qquad \Gamma \vdash^{\gamma}_{c|q} v_2 : S ! \Delta}{\Gamma \vdash^{\gamma}_{c|q} \textbf{继续}\ v_1\ v_2 : T ! \Delta; \xi}$$

(返回)
$$\frac{\Gamma \vdash^{\gamma}_{c|q} v : T ! \Delta}{\Gamma \vdash^{\gamma}_{c|q} \textbf{返回}\ v : T ! \Delta; \xi}$$

(Do)
$$\frac{\Gamma \vdash^{\gamma}_{c|q} e_1 : S ! \Delta; \xi \qquad \Gamma, \gamma', \gamma \sqsubseteq \gamma', (x : S)^{\gamma'} \vdash^{\gamma'}_{c|q} e_2 : T ! \Delta; \xi}{\Gamma \vdash^{\gamma}_{c|q} \textbf{执行}\ x \leftarrow e_1\ \textbf{在}\ e_2 : T ! \Delta; \xi}$$

(Op)
$$\frac{\Gamma \vdash^{\gamma}_{c|q} v : S ! \Delta \qquad \text{操作} : S \to T \in \Sigma \qquad \text{操作} \in \xi}{\Gamma \vdash^{\gamma}_{c|q} \textbf{操作}(v) : T ! \Delta; \xi}$$

(处理)
$$\frac{\Gamma \vdash^{\gamma}_{c|q} e : S ! \Delta; \xi_1 \qquad \Gamma \vdash^{\gamma}_{c|q} h : S ! \xi_1 \Longrightarrow T ! \xi_2 ! \Delta \qquad \forall \text{操作} \in \xi_1 \setminus \xi_2 . \text{操作} \in \text{域}(h)}{\Gamma \vdash^{\gamma}_{c|q} \textbf{处理}\ e\ \textbf{with}\ \{h\} : T ! \Delta; \xi_2}$$

(拼接)
$$\frac{\Gamma \vdash_{s} e : (\text{代码}\ (T^0 ! \xi)^{\gamma})^{-1} ! \Delta}{\Gamma \vdash^{\gamma}_{c|q} \$e : T^0 ! \Delta; \xi}$$

(子表达式)
$$\frac{\Gamma \vdash^{\gamma'}_{c|q} e : T ! \Delta; \xi \qquad \Gamma \vDash \gamma' \sqsubseteq \gamma}{\Gamma \vdash^{\gamma}_{c|q} e : T ! \Delta; \xi}$$

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

(Ret-处理器)
$$\frac{\Gamma, \gamma', \gamma \sqsubseteq \gamma', (x : S)^{\gamma'} \vdash^{\gamma'}_{c|q} e : T ! \Delta; \xi_2}{\Gamma \vdash^{\gamma}_{c|q} \textbf{返回}(x) \mapsto e : (S ! \xi_1 \Longrightarrow T ! \xi_2) ! \Delta}$$

(操作处理器)
$$\frac{\text{操作} : A \to B \in \Sigma \qquad \Gamma \vdash^{\gamma}_{c|q} h : \Gamma \Longrightarrow S ! \xi T ! \xi_2 ! \Delta \qquad \Gamma, \gamma', \gamma \sqsubseteq \gamma', (x : A)^{\gamma'}, (k : B \xrightarrow{\xi_2} T)^{\gamma'} \vdash^{\gamma'}_{c|q} e : T ! \Delta; \xi_2}{\xi_1 \subseteq \xi_2 \cup \{\text{操作}\}\ \textbf{操作}(x', k') \mapsto e' \notin h \quad \Gamma \vdash^{\gamma}_{c|q} h; \textbf{操作}(x, k) \mapsto e : (S ! \xi_1 \Longrightarrow T ! \xi_2) ! \Delta}$$

(子处理程序)
$$\frac{\Gamma \vdash^{\gamma'}_{c|q} h : S ! \xi_1 \Longrightarrow T ! \xi_2 \qquad \Gamma \vDash \gamma' \sqsubseteq \gamma}{\Gamma \vdash^{\gamma}_{c|q} h : S ! \xi_1 \Longrightarrow T ! \xi_2}$$

图 22. c|q-的类型规则： $\lambda_{\langle\langle \mathsf{op} \rangle\rangle^{\gamma}}$

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

S-类型规则

类型上的层级注释大多省略

$\boxed{\Gamma \vdash_s v : T^{-1}}$

(S-自然数)
$$\frac{}{\Gamma \vdash_s m : \mathbb{N}}$$

(S-变量)
$$\frac{\Gamma(x) = T^{-1}}{\Gamma \vdash_s x : T^{-1}}$$

(S-Lambda)
$$\frac{\Gamma,\ x : S \vdash_s e : T\,!\,\Delta}{\Gamma \vdash_s \lambda x.e : (S \xrightarrow{\Delta} T)}$$

(S-延续)
$$\frac{\Gamma,\ x : S \vdash_s e : T\,!\,\Delta}{\Gamma \vdash_s \kappa x.e : (S \xrightarrow{\Delta\!\!\!\!\rightarrow} T)}$$

$\boxed{\Gamma \vdash_s e : T^{-1}\,!\,\Delta}$

(小程序-应用)
$$\frac{\Gamma \vdash_s v_1 : (S \xrightarrow{\Delta} T) \qquad \Gamma \vdash_s v_2 : S}{\Gamma \vdash_s v_1\ v_2 : T\,!\,\Delta}$$

(S-继续)
$$\frac{\Gamma \vdash_s v_1 : \text{代码}(S\,!\,\xi_1)^{\gamma} \xrightarrow{\Delta\!\!\!\!\rightarrow} \text{代码}(T\,!\,\xi_2)^{\gamma'} \qquad \Gamma \vdash_s v_2 : \text{代码}(S\,!\,\xi_1)^{\gamma}}{\Gamma \vdash_s \textbf{继续}\ v_1\ v_2 : \text{代码}(T\,!\,\xi_2)^{\gamma'}\,!\,\Delta}$$

(S-返回)
$$\frac{\Gamma \vdash_s v : T}{\Gamma \vdash_s \textbf{返回}\ v : T\,!\,\Delta}$$

(S-执行)
$$\frac{\Gamma \vdash_s e_1 : S\,!\,\Delta \qquad \Gamma,\ x : S \vdash_s e_2 : T\,!\,\Delta}{\Gamma \vdash_s \textbf{执行}\ x \leftarrow e_1\ \textbf{在}\ e_2 : T\,!\,\Delta}$$

(S-操作)
$$\frac{\Gamma \vdash_s v : S \qquad \text{操作} \in \Delta \qquad \text{操作} : S \to \text{代码}(T\,!\,\xi)^{\gamma} \in \Sigma}{\Gamma \vdash_s \textbf{操作}(v) : \text{代码}(T\,!\,\xi)^{\gamma}\,!\,\Delta}$$

(S-处理)
$$\frac{\Gamma \vdash_s e : \text{代码}(S\,!\,\xi_1)^{\gamma}\,!\,\Delta \qquad \Gamma \vdash_s h : (\text{代码}(S\,!\,\xi_1)^{\gamma})\,!\,\Delta_1 \Longrightarrow (\text{代码}(T\,!\,\xi_2)^{\gamma})\,!\,\Delta_2 \qquad \forall \text{操作} \in \Delta_1 \setminus \Delta_2.\text{操作} \in \mathrm{dom}(h)}{\Gamma \vdash_s \textbf{处理}\ e\ \textbf{与}\ \{h\} : \text{代码}(T\,!\,\xi_2)^{\gamma}\,!\,\Delta_2}$$

(S-引用)
$$\frac{\Gamma \vdash_q e : T\,!\,\Delta;\ \xi}{\Gamma \vdash_s e : \text{代码}(T\,!\,\xi)^{\gamma}\,!\,\Delta}$$

(S-子级)
$$\frac{\Gamma \vDash \gamma' \sqsubseteq \gamma \qquad \Gamma \vdash_s e : \text{代码}(T\,!\,\xi)^{\gamma'}\,!\,\Delta}{\Gamma \vdash_s e : \text{代码}(T\,!\,\xi)^{\gamma}\,!\,\Delta}$$

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

(S-Ret-处理)
$$\frac{\Gamma,\ x : \text{代码}(S\,!\,\xi_1)^{\gamma} \vdash_s e : \text{代码}(T\,!\,\xi_2)^{\gamma}\,!\,\Delta_2}{\Gamma \vdash_s \textbf{返回}(x) \mapsto e : (\text{代码}(S\,!\,\xi_1)^{\gamma})\,!\,\Delta_1 \Longrightarrow (\text{代码}(T\,!\,\xi_2)^{\gamma})\,!\,\Delta_2}$$

(S-操作处理器)
$$\frac{\text{操作} : A \to \text{代码}(B\,!\,\xi)^{\gamma} \in \Sigma \qquad \Gamma \vdash_s h : (\text{代码}(S\,!\,\xi_1)^{\gamma})\,!\,\Delta_1 \Longrightarrow (\text{代码}(T\,!\,\xi_2)^{\gamma})\,!\,\Delta_2 \qquad \Gamma,\ x : A,\ k : (\text{代码}(B\,!\,\xi)^{\gamma} \xrightarrow{\Delta_2\!\!\!\!\rightarrow} \text{代码}(T\,!\,\xi_2)^{\gamma}) \vdash_s e : (\text{代码}(T\,!\,\xi_2)^{\gamma})\,!\,\Delta_2 \qquad \Delta_1 \subseteq \Delta_2 \uplus \{\text{操作}\} \qquad \textbf{操作}(x', k') \mapsto e' \notin h}{\Gamma \vdash_s h;\ \textbf{操作}(x,\ k) \mapsto e : (\text{代码}(S\,!\,\xi_1)^{\gamma})\,!\,\Delta_1 \Longrightarrow (\text{代码}(T\,!\,\xi_2)^{\gamma})\,!\,\Delta_2}$$

图 23。 S-类型规则 $\lambda_{\langle\langle \mathsf{op} \rangle\rangle^{\gamma}}$

H MacoCaml 中 C4C 检查的实现

列表 14 列出了 MacoCaml 编译器中 C4C 检查的实现。

(1) **Check** 原语对应第 6 行的Check 函数。**Check**$_{\mathsf{M}}$ 原语依赖于第 15–17 行处理的 `Mute` 效应，并在第 18–21 行的分支中执行，在该分支中处理了除 `Mute`和`FreeVar`之外的效应。

(2) **`dlet`** 原语对应第 11–14 行。具体来说，它会计算自由变量（第 19 行）。由于自由变量集合为空时，Check 不执行任何操作，因此第 13–14 行会执行以下操作之一：(a) 如果自由变量集合为空，则恢复延续；(b) 执行另一个`FreeVar` 效应，以检查剩余的自由变量是否安全。如果此检查成功返回，则恢复延续。

(3) err 被实现为一个未处理的 `FreeVar` 效应（没有对应的行号）。

```
OCaml 1 type_Effect.t += FreeVar: IdentSet.t →unit Effect.t 2 type_Effect.t += Mute: IdentSet.t →unit
Effect.t 3 4 (* 部分代码已省略 *) 5 6 let checkvars = if not (IdentSet.is_empty vars) then perform (FreeVar vars)
7 8 let exec_in_当前_作用域 scope f =9 letmuted_vars = ref IdentSet.empty in 10 match f() with 11 effectFreeVar
fvs, k→12 let free = IdentSet.diff fvs (IdentSet.union (!muted_vars) scope) in 13 check free; 14 继续 k ()15 |
effect Mute (fvs), k →16 muted_vars := IdentSet.union (!muted_vars) (fvs); 17 继续 k ()18 | effect op, k →
19 perform (Mute (IdentSet.union scope (!muted_vars))); 20 muted_vars := IdentSet.empty ; 21 (match perform
op with v →继续 k v)22 | c, fvs →23 let free = IdentSet.diff fvs scope in 24 check free; 25 (c, free) 26 27
let new_scope alphas f =28 let scope = IdentSet.of_list alphas in 29 let c, fvs = exec_in_当前_作用域 scope f in
30 (c, fvs) 31 32 let empty = IdentSet.empty 33 let free_变量 var = IdentSet.singleton var 34 let merge_free_变量
s fvs1 fvs2 =IdentSet.union fvs1 fvs2
```

列表 14. MacoCaml 中 C4C 检查的实现

收稿日期：2025-07-10；录用日期：2025-11-06