UNIVERSITÉ DE LORRAINE

´洛林 IAEM 博士研究生院

纯函数式编程环境下安全目的地传递的形式化与实现

`论文

提交并于2025年11月14日公开答辩

以获得

洛林大学博士学位

（计算机科学专业）

par

Thomas BAGREL

评审委员会组成

Pr´esident:	Laurent VIGNERON	
报告人:	多米尼克 ORCHARD	肯特大学
	加布里埃莱 KELLER	乌得勒支大学
评审人:	黛莉娅 KESNER	IRIF – 巴黎城市大学
	Laurent VIGNERON	LORIA – 洛林大学
	Gabriel SCHERER	IRIF – 巴黎城市大学
指导教师:	Horatiu CIRSTEA	LORIA – 洛林大学（论文导师 `论文）
	Arnaud SPIWACK	Tweag（工业导师）

洛林计算机科学及其应用研究实验室 — UMR 7503

使用 thesul 类进行排版。

我谨以此论文献给 *Pascal* 和 *Gérard*，希望今天能让他们感到自豪。

摘要

目的地传递编程引入了目标这一概念，它表示一个尚未使用过的内存单元的地址，且只能向其中写入一次。这些目标可以作为函数参数传递，使调用者能够掌控内存管理：被调用的函数只需填充该内存单元，而不必为返回值分配空间。尽管目的地传递主要用于系统编程，但它在纯函数式编程中也有应用，可用于编写以往无法用常见的不可变数据结构表达的程序。

在本论文中，我们开发了一种带有目标的 λ演算，即 λ_d。这一新的理论系统比现有的类似研究更具表达力，因为其中的目的地传递被设计得尽可能灵活。这种表达力得益于一种模态类型系统，它结合了线性类型和年龄系统，以管理资源的词法控制，从而保证目的地传递的安全性。我们借助 Coq 证明助手，以机械化方式通过进展和类型保持这两个常见定理证明了系统的安全性。

接下来，我们将说明如何把在这一理论演算中形式化的目标传递机制集成到现有的纯函数式语言 Haskell 中；Haskell 的类型系统不如我们的理论系统强大。为了保证安全性，必须限制目标管理的灵活性。随后，我们进一步完善这一实现，以恢复大部分灵活性，但这会增加用户的复杂度。

Haskell 中的原型实现显示，在遍历或映射列表、树等大型数据结构时，采用目的地传递颇具前景。

摘要

目标传递风格编程引入了目标，它们表示一次写入内存单元的地址。这些目标可以作为函数参数传递，使调用方能够控制内存管理：被调用方只需填充该单元，而无需为返回值分配空间。虽然目的地传递通常用于系统编程，但它在纯函数式编程中也有应用，可使此前无法使用常见不可变数据结构表达的程序得以实现。

在这篇论文中，我们开发了一种带有目的地的核心 λ 演算，即 λ_d。与现有的类似系统相比，我们的新演算表达能力更强，其目的地传递的设计尽可能灵活。为使目标传递具备安全性，我们采用模态类型系统，将线性类型与用于管理作用域的年龄系统结合起来。我们使用 Coq 证明助手对核心演算的类型安全性进行了形式化证明。

随后，我们探讨如何将这一核心演算改造为现有的纯函数式语言 Haskell；其类型系统不如我们定制的理论类型系统强大。要保留安全性，就必须牺牲处理目的地时的部分灵活性。之后，我们进一步改进实现，以恢复其中大部分灵活性，但代价是增加用户使用的复杂度。

Haskell 中的原型实现表明，在遍历或映射大型数据结构（如列表或数据树）时，采用目标传递风格编程颇有成效。

致谢

怀着百感交集的心情，我终于走到了这段旅程的终点。这段旅程占据了我三年半的人生，既艰难又充满挑战，无论是对我自身，还是对所研究的课题而言都是如此。幸运的是，我有幸得到一群杰出人士的指导与支持。他们每个人都以自己的方式作出了贡献，才让我今天得以为这份手稿画上句号。事实上，这群人的支持远非我在此页所能尽述；因此，我谨向所有未能在此提及的人致以最诚挚的谢意。

我由衷感谢我的父母。早在这篇论文开始之前，他们就一直给予我指引与支持，并不断鼓励和肯定我，让我即使面对种种困难，也能追随自己选择的道路。我也想起 Marinette，如今她正走在一条与我相近的道路上，她的坚定与卓越令我钦佩，她的成功更是毋庸置疑。我也立刻想到了 Julien 和 Ocelline，他们包容我的情绪与烦恼，总是向我敞开家门，让我能够以比前一天更坚强、更有干劲的姿态重新出发。也要感谢 Luna，她如同一道阳光，照亮日常生活中那些较为灰暗的时刻；即使我失去希望，她也始终相信我。还要感谢 Tim，他看待世界的方式让我重拾几分乐观，也帮助我为自己的工作找到意义。

我还要向 Arnaud 致以诚挚的谢意。尽管肩负诸多责任，他仍每天准时出现，回答我的疑问与困惑，并认真跟进我那些天马行空的想法。他对科学的热情，让与他的每一次交流都引人入胜、令人受益匪浅。感谢他在这段旅程中与我分享的所有知识，也感谢他面对我一再固执时展现的耐心。我也要感谢 Horatiu 对我的支持、帮助我处理学术行政事务，以及为改进我的工作提供的宝贵建议。感谢他们二位信任我，并愿意与我携手踏上这个项目；这个项目是在 Dominique Mery 和 Clément Hurlin 的宝贵建议下启动的。

不过，还有许多人的陪伴，让我得以在日常生活中保持平衡。首先是几位乐观开朗的博士生同事 Sarah、Vincent、Ghislain 和 Alessio。我对他们的敬重，远远超出了我们在办公室里互相开玩笑时所表现出来的程度。衷心祝愿他们在各自的人生旅途中一切顺利。还有更广义上的 Mosel-Veridis团队，以及我们那些常常在科学与服饰之间切换的话题。我的*Tweag* 同事们也是如此。感谢 Clément（再次感谢），他一直照顾我；我很幸运能和他分享对战锤的热爱，以及对我们职业的某些相似看法。感谢 Camilla，她热爱生活、待人友善，为巴黎办公室带来了满满的活力；这些年来，她也总是设法让我在后勤方面有求必应。感谢 Florent，我最喜欢的愤世嫉俗搭档；感谢 Alex、Nicolas 和 Simeon，以及那些愉快的午餐时光；感谢 Yann，帮我走出了作用域逃逸的困境；感谢 Guillaume，他总是笑容满面，还会跳舞；也感谢 Benjamin，他负责管理巴黎这片适宜工作（也适宜生活）的地方。

在我的私人朋友圈中，我也要感谢 Julien、Florian 和 Alexandre，他们一直密切关注着我的进展，而我很遗憾常常不得不拒绝他们邀我外出或参加活动的提议。也感谢 Ysaline，她帮助我渡过了许多难关；我很钦佩她的坚定与洞察力。我还特别想到了 Marion，是她为我指引了学术之路，并一直密切关注我的学术历程。我很早就开始使用 LaTeX，也要归功于她；时至今日，我仍用它来制作这份文档。最后，我想把最后几句话献给祖父 Gérard，他将严谨的态度和对数学的热情传给了我；也献给叔叔 Pascal，我很早便和他一起探索工程学。尽管他们已无法读到这些文字，但他们一直非常关心我的进步与成就；我今天能走到这一步，部分也要归功于他们。

目录

前端 er ii

摘要.ii 致谢.iv 目录.vii 图目录.xi 代码清单目录.xii

引言 1

1 内存管理：编程语言的核心设计维度.1 2 目的地传递：植根于古老的命令式世界.2 3 函数式编程语言.3 4 带孔的函数式结构：Minamide 的奠基性工作.4 5 函数式目标传递的形式化.5 6 协调写指针与不可变性.6

第 1 章 – 线性 λ 演算与线性 Haskell 9

1.1 从 λ 演算到线性 λ 演算.9 1.2 线性 λ 演算：线性逻辑的计算诠释.11 1.3 非受限资源的便捷处理.13 1.4 借助分级模态方法回到单一上下文.16 1.5 深层模态：将模态投射到数据结构的字段上.19 1.6 线性 λ 演算（带深层模态）的操作语义.21 1.7 Haskell 中的线性类型.23 1.8 唯一性与线性作用域.24 1.9 结论.28

第2章——基于一等目的地的形式函数式语言： λ_d **31**

2.1 函数式环境中的目标传递.31 2.2 在 λ_d 中使用目标.32 2.2.1 构建词汇.33

2.2.2 尾递归映射.36 2.2.3 使用目标的函数式队列.37 2.3 目的地的作用域逃逸.38 2.4 广度优先树遍历.40 2.5 类型系统.42 2.5.1 模态与年龄半环.43 2.5.2 类型规则.44 2.6 操作语义.49 2.6.1 运行时值与新的类型上下文形式.49 2.6.2 求值上下文与命令.52 2.6.3 归约语义.55 2.6.4 类型安全.60 2.7 类型安全性的形式化证明.60 2.8 将先前工作转换为 λ_d 项和类型.63 2.9 使用原地内存变异实现 λ_d .64 2.10 结论.66

第3章 – Haskell 中目的地传递的首个实现 6 9

3.1 实现目标：线性 Haskell.69 3.2 仅使用线性类型实现安全目的地传递.70 3.3 DPS Haskell 编程一瞥.70 3.3.1 高效差分列表.70 3.3.2 广度优先树遍历.74 3.4 DPS Haskell API 与设计考量.75 3.4.1 UAmpar 类型.77 3.4.2 目的地填充函数.78 3.5 C

9 3.5.1 紧凑区域.80 3.5.2 紧凑区域中的实用 7 DPS 编程.81 3.5.3 紧凑区域中UAmpar的内存表示.83 3.5.4 使用Generics 为所有构造器推导填充 .85 3.5.5 GHC 内部机制及其运行时系统的变更.86 3.6 评估 DPS Haskell 的性能.91

3.6.1 列表拼接与差异列表.92 3.6.2 广度优先树重标记.94 3.6.3 解析 S 表达式.95 3.6.4 对列表应用函数.96

3.7 结论 . 97

第4章 – 扩展 DPSHaskell 以支持线性数据结构 99

4.1 线性数据的目的地面临的挑战：作用域逃逸、更新.99 4.2 补全单孔 ampar 的运算符族.101 4.3 存储线性数据的多孔 ampar.105 4.4 最终 API 和广度优先树遍历，更新.107 4.5 结论.111

相关工作 **113**

1 构造器模尾递归.113 2 用于高效内存管理的目标传递风格.114 3 半公理化序列演算.114 4 带孔数据结构的函数式表示.115 5 上下文模尾递归与命令式二叉搜索树的函数式本质 .....116 6 Oxidizing OCaml.116 7 使用分级模态类型进行定量程序推理 .117 8 在 Mezzo 中进行权限编程.117 9 Rust 生命周期.118

结论 **121**

贡献.121 未来展望.124

参考文献 **129**

图列表

1.1 自然演绎表示中的直觉主义线性逻辑，序列式（ILL）. . . . . 11 1.2 单子式表示中的线性 λ演算语法（λ_{L1}）.12 1.3 单子式表示中的线性 λ演算类型规则（λ_{L1}–ty）.12 1.4 二元表示中的线性 λ演算语法（λ_{L2}）.14 1.5 二元表示中的线性 λ演算类型规则（λ_{L2}–ty）.14 1.6 模态表示中的线性 λ演算语法（λ_{Lm}）.17 1.7 模态表示中的线性 λ演算类型规则（λ_{Lm}–ty）. 18 1.8 模态表示中的线性 λ演算深层模态的类型规则（λ_{Ldm}–ty）. . . . . 21 1.9 λ_{Ldm} 的小步语义（λ_{Ldm}–sem）.

2.1 λ_d 的项、类型和模态.42 2.2 λ_d 的类型规则（λ_d–ty）.45 2.3 语法糖的导出类型规则（λ_d–ty_{S}）.46 2.4 嵌套作用域中年龄的变化.47 2.5 运行时值、新建类型上下文形式和运算符.50 2.6 运行时值的类型规则（λ_d–ty_{V}）.51 2.7 求值上下文的语法.53 2.8 求值上下文（λ_d–ty_{E}）和命令（λ_d–ty_{CMD}）的类型规则.54 2.9 λ_d 的小步语义（λ_d–sem），第 1 部分.56 2.10 λ_d 的小步语义（λ_d–sem），第 2 部分.57

3.1 DPS Haskell 中的映射关系.71 3.2 DPS Haskell 中 `newUAmpar tok` 的内存表示.72 3.3 DPS Haskell 中 `append newUAmpar 1` 的内存行为.73 3.4 DPS Haskell 中 `concat dlist1 dlist2`的内存行为（基于 `fillComp`）. . . . . 73 3.5 DPS Haskell 中 `toList dlist`的内存行为.73 3.6 `fill @'(:):: UDest [t] ⊸ (UDest t, UDest [t])` 的内存行为.79 3.7 `fill @'[]:: UDest [t] ⊸ ()` 的内存行为.79 3.8 `fillLeaf:: t → UDest [t] ⊸ ()` 的内存行为.79 3.9 使用 `Ur`表示 `UAmpar r` （未采用）. 84

3.10 使用间接引用表示 UAmpar r .85 3.11 fill 的实现架构示意图.90 3.12 列表和差分列表上的迭代连接基准测试.93 3.13 S 表达式解析器基准测试.93 3.14 广度优先树重标记基准测试.95 3.15 列表映射函数基准测试.96

代码清单目录

2.1 列表上的尾递归`map` 函数（λ_d）.37
2 2 λ_d 38 . 差
2 3 λ_d
分列表与队列的实现（在.....中）.广度优先树遍历：在.....中使用目的地传递. 41
2 4 λ_d . 项
的去糖化规则（在.....中）.43 3 1 . DPS Haskell 中差分列表的实现.72 3 2 . Haskell 中 Hood-Melville 队列的实现.75 3 3 . DPS Haskell 中广度优先树遍历的实现.76 3 4 76 . DPS Haskell API，初始版本.3 81 3 5 6 . 紧凑区域的 DPS Haskell API.Haskell 中 S 表达式解析器的实现（无目的地）.82 3 7 . DPS Haskell 中 S 表达式解析器的实现.82 3 8 . `reifyInfoPtr#` 在 `compiler/GHC/StgToCmm/Prim.hs` 中的实现.88 3 9 . `compactAddHollow#` 在`rts/Compact.cmm` 中的实现.89 3 10 . `LCtorToSymbol` 在`compiler/GHC/Builtin/Types/Literal.hs` 中的实现. . . . . 90 4 1 . 扩展 DPS Haskell 中用于线性数据的高效队列.108 4 2 . 扩展 DPS Haskell 中使用高效队列进行广度优先树遍历 . . . . . 109 4 3 110 . 支持线性目标的扩展 DPS Haskell API.

引言

1 内存管理：编程语言的核心设计维度

在过去五十年里，编程语言已发展成为不可或缺的工具，深刻影响着我们与计算机交互的方式，以及我们在几乎所有领域处理信息的方式。与人类语言颇为相似，种类繁多的编程语言反映了计算需求、设计理念以及开发者和研究人员所追求目标的多样性。这种多样性源于专业化应用、用户偏好，以及对速度或表达能力持续改进的追求。如今，已有数百种编程语言，每种语言都有其独特特性和专长，因此开发者在选择语言时需要综合考量。

从宏观层面来看，编程语言在数据表示和程序执行等核心方面的处理方式各不相同，并可依据若干关键维度进行分类。这些维度往往揭示了语言的基本原则，以及它对不同类型应用的适用性。区分编程语言的一些主要特征包括：

· 用户向计算机表达意图的方式——是通过过程式语言中明确的逐步指令，还是采用更强调“计算什么”而非“如何计算”的函数式/声明式风格；

· 数据的组织与操作——采用面向对象范式，将领域数据封装在能够彼此交互的对象中；或者采用更简单的数据结构，由函数和过程对其进行操作与拆解；

· 抽象层次——高级语言抽象掉技术细节，使开发者能够用更少的代码行实现更复杂的功能；低级语言则提供更多对环境和任务具体执行过程的控制；

· 内存管理——内存分配和内存释放由语言自动处理，还是需要程序员明确介入；

· 副作用和异常——它们是否能够用该语言表示、捕获和／或操作。

在这些方面中，内存管理是编程语言设计最关键的方面之一。每个程序都需要内存来存储数据和指令，而必须谨慎管理这些内存，以避免错误并维持性能。通常，程序处理的输入规模和数据结构可能差异很大，因此需要动态内存分配来确保程序顺利执行。因此，所选的内存管理方案将深刻影响程序员编写和组织代码的方式，也会影响哪些功能能够以合理的成本纳入语言中。

Python 或 Java 等高级语言通常通过垃圾回收自动管理内存。借助自动垃圾回收（由追踪式或引用计数式垃圾回收器实现），内存分配和内存释放都在后台完成，使开发者不必处理

手动内存控制的复杂性。这种自动内存管理简化了编程，让开发者能够专注于功能，并使语言更易于使用，从而加快开发速度。尽管垃圾回收总体上相当快速，但在某些特定用例中可能会很慢；其不可预测的开销或程序执行暂停也令人望而生畏，因此不适用于实时应用。

相比之下，C语言或 Zig 这类低级语言通常允许开发者直接控制内存分配和内存释放。这确实有助于进行更充分的优化并高效使用资源，在系统编程或对性能敏感的应用中尤其有用。然而，这种控制也伴随着更高的风险；如果不仔细检测并处理，内存泄漏、缓冲区溢出和悬空指针等错误可能导致系统不稳定和安全漏洞。

有趣的是，有些语言打破了这些常见分类，通过安全的高级抽象提供精确的内存控制。这类语言允许用户显式管理资源的生命周期，同时由用户负责在每项资源生命周期的开始和结束时分配及释放内存。最广为人知的例子是 C++ 中的智能指针，以及 Rust 中的所有权模型；其基本原则也称为 作用域绑定资源管理 (SBRM) 或 资源获取即初始化 (RAII)。SBRM 最初仅适用于早期 C++中的栈分配对象；随着 2000 年代初期用于堆分配对象的智能指针问世，SBRM 也随之发展。此后，这些工具已成为现代 C++ 和 Rust 的基础，大幅提升了内存安全性。尤其是，Rust 无需承担垃圾回收的开销，便能提供与垃圾回收语言相当的安全保证，代价是学习曲线较为陡峭。因此，Rust 既适用于高级应用程序编程，也适用于低级系统开发，弥合了传统上因内存管理方式不同而产生的鸿沟。

2 目的地传递：在古老的命令式世界中扎根

在系统编程中，精细控制内存至关重要，同一内存缓冲区往往会被重复使用多次，以节省内存空间并减少对内存分配器的昂贵调用次数。因此，我们不希望中间函数为其计算结果分配内存。相反，我们希望为它们提供一个内存地址，让它们将结果写入其中，这样我们就可以决定重用已分配的内存槽位，或者写入内存映射文件或 RDMA 缓冲区。同样，我们可以为函数提供多个输出槽位，让它写入多个结果。

在实践中，这意味着父作用域不仅管理函数输入的分配和内存释放，也管理函数输出的分配和内存释放。输出槽位以指针（或在高级语言中以可变引用）的形式传递给函数，使函数能够直接写入由调用方管理的内存。这些指针称为输出参数或目标，它们指定了函数应将输出存储到的确切内存位置。

这一理念构成了目标传递风格编程（DPS）的基础：调用方不再让函数为输出分配内存，而是向其提供一个预先分配的内存空间的写入权限，用于存放函数的结果。将内存管理责任交给调用方而非被调用方，使函数的使用更加灵活：这正是目标传递风格编程（DPS）的核心优势。

在低层系统编程环境中，DPS 还能直接带来性能优势。例如，可以预先分配一块较大的内存块，再将其划分为较小的段，并将每一段指定为特定函数调用的输出槽位。这样可以最大限度地减少单独分配的次数，而单独分配往往是性能瓶颈，尤其是在内存分配密集的上下文中。

在下文将介绍的 DPS 函数式解释中，节省内存的效果会更加微妙；我们将转而利用 DPS 在灵活性和表达能力方面的优势，突破函数式编程的一些常见限制。

3 函数式编程语言

函数式编程语言通常被视为与系统语言处于光谱两端的相反一端。

函数式编程没有单一的定义，但大多数人都认同，函数式语言：

- 支持 λ 抽象（又称作为一等值的函数），它们能够捕获父级环境的部分内容（闭包）、存储起来，并作为参数传递；
- 强调表达式而非语句，其中每条指令都会产生一个值；
- 通过应用和组合函数来构建复杂表达式，而不是串联语句。

基于这些原则，函数式语言往往倾向于采用不可变数据结构和声明式风格：通过指定新数据结构与现有数据结构之间的差异来定义新数据结构，并且通常依赖结构共享和持久化数据结构，而不是直接进行变异。

由于函数式语言中“一切皆为表达式”，它们尤其适合进行数学形式化。这并非偶然：函数式编程中的许多核心概念都源自数学，例如 λ演算——一种极简却具有完整表达能力的计算模型，同时还通过 Curry-Howard 同构与形式化证明紧密相连。

尽管如此，许多函数式语言仍然允许包含副作用的表达式。副作用涵盖程序对其环境所能采取的所有可观察操作，例如写入文件或控制台，或修改内存。副作用很难推理，尤其是在该语言的任何表达式都可能产生副作用的情况下；但事实上，副作用是任何编程语言中都至关重要的一部分：没有副作用，程序就无法将其结果传达给“外部”。

纯函数式编程语言 有些函数式语言采取更严格的方法，禁止副作用，除非是在主程序的边界处。这一概念称为纯函数式编程。在这种模式下，程序员定义的所有表达式都会计算并返回值，而不产生副作用。相反，执行副作用（如向控制台打印内容）的意图 可以表示为一个值，只有语言运行时才会将其转化为预期的副作用。

这项限制带来了显著的好处。程序可以完全建模为数学函数，从而更容易推理其行为，也能对程序行为进行更强大的分析。

纯函数式语言的一个关键属性是引用透明性：任何函数调用都可以替换为其结果，而不改变程序的行为。显然，C语言等语言不具备这一属性；例如，将`写入(我的文件，字符串，12)` 替换为值`12` （写入成功时的返回值），并不能产生预期行为：后者根本不会产生任何写入副作用。

这种将程序视为纯函数进行推理的能力，极大地提高了程序的可预测性，尤其是在跨越库边界时，并全面提升了软件安全性。

函数式语言中的内存管理 显式内存管理往往是非纯的，因为以一种之后可以检查的方式修改内存属于副作用。因此，大多数函数式语言（即使是像 OCaml 或 Scala 这样并非完全纯的语言）也倾向于依赖垃圾回收器，为用户抽象掉内存管理和内存访问。

不过，这并不意味着函数式语言必须完全放弃对内存控制的任何方面；这只是意味着内存控制不能涉及像 `malloc` 或 `free` 这样的显式副作用语句。实际上，这要求内存管理不应影响语言中正在计算的表达式的值。实现这一点的一种方式，是在特定函数或代码块上附加注解或编译指示，指明应如何或在何处分配内存。另一种方法是使用带有模态的自定义类型系统，在类型层跟踪值在内存中的管理方式，例如近期针对 OCaml 的模态开发 ⟦Lorenzen、White 等人，2024⟧。此外，有些语言会使用特殊的内置函数，例如 Haskell 中的`oneShot`；这些函数不会影响其所包装表达式的结果，但对编译器管理内存具有特殊意义。

还有另一种可能的解决方案。我们先前提到，纯函数式语言中禁止的副作用，是指用户能够观察到 的对环境或内存的任何修改。如果我们允许显式的内存管理表达式，同时确保这些对内存或环境的（临时）更改仍不可被观察到，会怎样?

这就是我们将采取的路径——此前已有其他人走过这条路——它能在坚守纯性原则的同时，实现更细粒度的内存控制。

4 带孔的函数式结构：Minamide 的奠基性工作

在大多数上下文中，带垃圾回收器的函数式语言无需程序员介入，就能高效管理内存。然而，大多数函数式语言共有的不可变性 这一特性带来了一个主要限制：我们只能直接构造最终形式的数据结构。这意味着，即使结构中的某些字段尚未计算出有意义的值，程序员也必须为每个字段提供初始值。之后，任何超出简单扩展原始结构的更新，都需要创建该结构的（部分）副本。这会给分配器和垃圾回收器带来沉重负担。

因此，无论是从外部来源读取数据，还是转换现有结构，生成大型结构的算法都可能需要进行许多中间分配，每次分配都代表一个临时处理状态。尽管不可变性有其优点，但在这种情况下，它可能会成为优化性能的障碍。

为克服这一限制，Minamide ⟦1998⟧ 为函数式语言引入了一种扩展，使其能够表示尚未完成的数据结构，并在之后对其进行扩展和补全。在不完整结构完成之前，不允许读取它们——这一点由类型系统强制保证——因此，用户无法察觉在将结构从不完整状态更新至完成状态的过程中发生的底层变异，用户只能观察到最终结果。这样既保留了不可变函数式语言的特性，也不会破坏纯性本身。通过类型级保证，确保命令式或非纯计算的影响对用户不可观察，从而使这些计算对用户不透明；这种方法是本文所述方案的核心。

在 Minamide 的工作中，不完整结构，即带有空洞的结构，由孔抽象表示——本质上是以结构的缺失组件为参数并返回完整结构的纯线性函数。换言之，它表示结构待完成的构造；理论上，只有在所有缺失部分都已提供后，才能完成该构造。孔抽象

也可以像函数一样进行组合：$(\lambda x \mapsto 1 :: x) \circ (\lambda y \mapsto 2 :: y) \rightsquigarrow \lambda y \mapsto 1 :: 2 :: y$。然而在底层，每个孔抽象在内存中并不由函数对象表示，而是由一对指针表示：一个读指针，指向该孔抽象所描述的数据结构的根节点；另一个写指针，指向数据结构中尚未指定的字段。两个孔抽象的组合可以立即执行，无需等待第二个孔抽象的缺失值；从内存角度看，其结果是将第一个孔抽象此前缺失的字段变异为指向第二个孔抽象的根节点。

在 Minamide 系统中，空洞（的指针）与包含它的结构不可分割。要与结构中的空洞交互，就必须与结构本身交互，这使得不完整结构只能包含一个空洞。因此，我们认为 Minamide 的方法尚不能算作目标传递风格编程。尽管如此，它仍是最早期的示例之一，展示了纯函数式语言如何允许数据结构以写入权限传递，同时保留内存安全性和纯性等关键语言保证。

这一思想最近由 ⟦Leijen 和 Lorenzen、 2025⟧以及 ⟦Lorenzen、Leijen、Swierstra 和 Lindley 2024⟧, 重新提出并加以扩展，其中带孔结构被称为一等上下文。

然而，要实现真正的目标传递风格编程，我们需要一种方法，在不必传递包含空洞的不完整结构的情况下，引用其中的空洞。这一点迄今在函数式环境中实际上鲜少得到探讨。⟦Bour、Clément 和 Scherer 2021⟧ 的工作或许是现有研究中最接近这一问题的，但目标传递风格仅用于编译器中的中间语言层面，以进行优化，因此用户无法在自己的代码中使用 DPS。

5 函数式目的地传递的形式化

我们将在这篇论文中开发一种函数式语言，其中 带孔结构 是语言中的一等类型，类似于 ⟦Minamide, 1998⟧, ，但同时也将指向这些空洞的一等指针（即目的地）集成到面向用户的语言中。

我们将看到，结合这两项特性，能像 ⟦Bour、Clément 和 Scherer, 2021⟧ 或 ⟦Leijen 和 Lorenzen, 2025⟧, 那样提升某些函数式算法的性能；此外，我们还能获得额外的表达能力，而这通常只是命令式语言才有的优势。事实上，这应该是首个在前端（而非仅在中间表示中，如 ⟦Bour、Clément 和 Scherer, 2021年；Leijen 和 Lorenzen, 2025⟧）支持写指针，从而能够灵活高效地构建数据结构的函数式语言！

事实上，我们将设计一种基于线性类型方法的形式语言，其中带孔结构 作为构造数据结构的核心抽象，取代函数式编程中传统的数据构造器用法。随后，我们将展示如何将这一理论演算中的许多理念和特性应用于实用的函数式语言——即 Haskell——同时尽可能保留其表达能力。我们还将展示，这种方法所预期的优势确实体现在 Haskell 原型实现中。

这种形式语言还旨在提供一个理论框架，涵盖现有的大多数函数式 DPS 研究，并可用于推理早期研究中的安全性和正确性属性。

6 协调写指针与不可变性

在函数式环境中引入无限制的写指针并不可取，因为这会破坏函数式语言易于推理的许多特性。相反，我们需要工具来确保只发生受控的变异。我们的目标是允许结构暂时保持部分未初始化状态，并让写指针指向这些未初始化部分。不过，一旦结构填充完毕，每个字段都分配了一个值，就应使其不可变，以维护函数式范式的完整性。简言之，我们需要对结构中的字段实行一次性写入规则，字段可通过与其关联的写指针（又称 目的地）进行变异。虽然在运行时强制目的地只能使用一次并不容易，但我们可以改用类型系统提供静态保证，确保每个目的地恰好使用一次。

在编程语言理论中，当需要新的保证时，通常会设计一种类型系统来强制执行这些保证，确保程序保持类型正确。我们在这里采用的正是这种方法，并借助一种成熟的类型系统——线性类型——来监控资源使用，尤其是资源被使用的次数。利用线性类型系统，我们可以设计一种语言，使带孔结构成为一等公民，并且指向这些空洞的写指针只能使用一次。这样可以确保每个空洞都恰好被填充一次，从而保持不可变性，并确保结构确实在读取之前完成。这也是 ⟦Minamide、1998⟧ 以及后续研究所采用的方法。

第 1 章

线性 λ演算与线性 Haskell

下文假设读者熟悉简单类型的 λ演算、其类型规则以及通常的语义。我们还假设读者对自然演绎的直觉主义片段有一定了解。关于这些概念的恰当介绍，敬请读者参阅 ⟦Pierce、 2002⟧ 以及 ⟦Sørensen 和 Urzyczyn 的著作 2006⟧。

1.1 从 λ演算到线性 λ演算

20 世纪 30 年代末，阿隆佐·丘奇提出了作为计算的形式数学模型的无类型 λ演算 。无类型 λ演算以函数抽象和应用为基础，并且是图灵完备的：换句话说，它的表达能力与同一年代提出的、相当命令式的图灵机模型相同。无类型 λ演算中唯一可用的对象是纯函数；不过，巧妙地利用它们可以编码许多其他类型的数据，因此它们足以构成一个坚实的计算模型。

1940年，Church 定义了其原始演算的一种类型化变体，称为简单类型 λ演算，简称 STLC。它舍弃了图灵完备性，但实现了强规范化：每个类型正确的项最终都会归约为范式。STLC 为项指定类型，并限制函数只能应用于类型匹配的项。这种限制由该演算的类型规则强制执行。

Howard 在 1969 ⟦Howard，1969⟧ 中指出，简单类型的 λ-演算实际上与一种称为自然演绎的逻辑框架同构。这一发现与 Curry 早期的研究有关；Curry 也曾注意到计算领域的系统与逻辑领域的系统之间存在类似的对称关系。这些见解后来被形式化为*Curry-Howard* 同构，该同构指出，类型化 λ-演算中的类型与证明系统中的公式相对应，而项则与这些公式的证明相对应。此后，Curry-Howard 同构被扩展到其他逻辑和演算中，并成为逻辑与计算领域研究的重要灵感来源。例如，Jean-Yves Girard 基于其关于 λ-演算模型的研究中的一个想法，首次提出了线性逻辑 ⟦Jean-Yves Girard，1987⟧, ，之后又提出了相应的演算，即线性 λ-演算。

什么是线性逻辑? 在某些逻辑表述中，尤其是序列演算中[1]，，假设会通过系统的结构规则显式地复制或丢弃，分别称为收缩和弱化（与自然演绎不同，在自然演绎中，这些操作都是隐式进行的，因为它们属于元理论）。因此，可以通过统计这些结构规则的使用次数来追踪某个公式被使用的次数。在线性逻辑中，收缩和弱化受到有意限制，因此默认情况下，每个假设都必须恰好使用一次。因此，逻辑蕴涵$\mathtt{T} \to \mathtt{U}$不属于线性逻辑，而由线性蕴涵$\mathtt{T} \multimap \mathtt{U}$取代；要证明$\mathtt{U}$，必须恰好使用$\mathtt{T}$一次。线性逻辑还引入了一种模态!，读作当然或*bang*，允许对特定公式进行弱化和收缩：$!\mathtt{T}$表示$\mathtt{T}$可以被任意次地使用（我们称之为无限制的）。我们称线性逻辑为次结构逻辑，因为它限制了通常逻辑中结构规则的使用。

我们在图 1.1 中给出了直觉主义线性逻辑（直觉主义线性逻辑，ILL）的一种自然演绎表述[2]，因为它很适合被计算性地解释为具有通常语法的线性 λ-演算。

除了与 ! 模态相关的规则外，直觉主义线性逻辑的所有规则都直接取自自然演绎（并进行了少量调整）。1 表示真值常元，$\oplus$ 表示析取，$\otimes$ 表示合取[3]。假设由多重集表示 Γ。这些多重集记录每个公式在其中出现的次数。Γ_1, Γ_2 中的逗号算子表示多重集并，因此会将 Γ_1 和 Γ_2中每个公式的出现次数相加。我们用 •表示空多重集。

现在让我们聚焦于 ! 模态的四条规则。提升规则 直觉主义线性逻辑/!P 规定，如果公式 $\mathtt{T}$ 只依赖于本身也不受限制的公式，那么它就可以变为不受限制的公式 $!\mathtt{T}$ 。为此，我们引入记号 $!\Gamma$ ，表示一个包含与Γ 相同假设、但全部包裹在 ! 模态中的上下文；换言之，潜在为空的多重集 $!\Gamma$ 中的每个假设都是不受限制的。降格规则 直觉主义线性逻辑/!D 规定，不受限制的公式 $!\mathtt{T}$ 可以用于需要线性公式 $\mathtt{T}$的位置。收缩规则 直觉主义线性逻辑/!C 和弱化规则 直觉主义线性逻辑/!W 分别规定，不受限制的公式 $!\mathtt{T}$ 可以随时被复制或丢弃。

线性逻辑系统看起来可能限制很多，但我们总能通过! 模态，在直觉主义线性逻辑中模拟常见的非线性自然演绎。Girard 在 [J.-Y. Girard, 1995], 的第 2.2.6 节中给出了这种转换的精确规则，其主要思想是给大多数公式加上前缀!，并将非线性蕴涵$\mathtt{T} \to \mathtt{U}$编码为$!\mathtt{T} \multimap \mathtt{U}$。

为了进一步熟悉这个系统，让我们来看看线性逻辑如何通过其规则有效地限制线性假设的复制。例如，让我们尝试（并失败地）从空环境中推导出蕴含式$\mathtt{T} \multimap (\mathtt{T} \otimes \mathtt{T})$：

$$\dfrac{\dfrac{\dfrac{}{\mathtt{T} \vdash \mathtt{T}}\,\text{ID} \quad \dfrac{}{\mathtt{T} \vdash \mathtt{T}}\,\text{ID}}{\mathtt{T},\ \mathtt{T} \vdash \mathtt{T} \otimes \mathtt{T}}\,\otimes\text{I}}{\mathtt{T} \vdash \mathtt{T} \multimap (\mathtt{T} \otimes \mathtt{T})}\,\multimap\text{I}$$

[1]序列演算是一种非常流行的逻辑框架，可作为自然演绎的替代方案；它也是 Gentzen 在 20 世纪 30 年代提出的。[2]不过，为了便于之后过渡到项的类型规则，我们借用了序列演算的序列风格记法。然而，由于我们处于直觉主义设置中，规则只能从一个多重集的公式推导出单个结论。[3]线性逻辑区分乘法合取与加法合取，以及乘法析取与加法析取；它们与否定和德摩根对偶性的交互方式各不相同：乘法联结词（例如 $\otimes$、）互为对偶，加法联结词（例如 $\oplus$、&）也互为对偶。在我们的设置中，我们聚焦于 $\otimes$ （乘法合取）和 $\oplus$ （加法析取），因为它们与标准逻辑联结词 $\wedge$ 和 $\vee$ 的对应关系更为密切。

$\boxed{\Gamma \vdash T}$ （推理规则）

$$\frac{}{T \vdash T}\,\text{ID} \qquad \frac{\Gamma,\ T \vdash U}{\Gamma \vdash T \multimap U}\,\multimap\text{I} \qquad \frac{}{\cdot \vdash 1}\,1\text{I} \qquad \frac{\Gamma \vdash T_1}{\Gamma \vdash T_1 \oplus T_2}\,\oplus\text{I}_1 \qquad \frac{\Gamma \vdash T_2}{\Gamma \vdash T_1 \oplus T_2}\,\oplus\text{I}_2$$

$$\frac{\Gamma_1 \vdash T_1 \qquad \Gamma_2 \vdash T_2}{\Gamma_1,\ \Gamma_2 \vdash T_1 \otimes T_2}\,\otimes\text{I} \qquad \frac{\Gamma_1 \vdash T \qquad \Gamma_2 \vdash T \multimap U}{\Gamma_1,\ \Gamma_2 \vdash U}\,\multimap\text{E} \qquad \frac{\Gamma_1 \vdash 1 \qquad \Gamma_2 \vdash U}{\Gamma_1,\ \Gamma_2 \vdash U}\,1\text{E}$$

$$\frac{\begin{array}{l}\Gamma_1 \vdash T_1 \oplus T_2 \\ \Gamma_2,\ T_1 \vdash U \\ \Gamma_2,\ T_2 \vdash U\end{array}}{\Gamma_1,\ \Gamma_2 \vdash U}\,\oplus\text{E} \qquad \frac{\Gamma_1 \vdash T_1 \otimes T_2}{\Gamma_2,\ T_1,\ T_2 \vdash U}\,\otimes\text{E} \qquad \frac{!\Gamma \vdash T}{!\Gamma \vdash\ !T}\,!\text{P} \qquad \frac{\Gamma \vdash\ !T}{\Gamma \vdash T}\,!\text{D} \qquad \frac{\Gamma_1 \vdash\ !T}{\Gamma_2,\ !T,\ !T \vdash U}\,!\text{C}$$

$$\frac{\Gamma_1 \vdash\ !T \qquad \Gamma_2 \vdash U}{\Gamma_1,\ \Gamma_2 \vdash U}\,!\text{W}$$

图 1 .1：自然演绎表示中的直觉主义线性逻辑，序列式（IL L)

在序列演算中，我们会在 ⊗I 和 —∘I 之间插入一个收缩步骤，以便从 T, T ⊢ T⊗T 推导出 T ⊢ T⊗T 。但是，在线性逻辑中，线性假设不允许收缩。ILL—∘I 规则的每次应用都恰好消耗一个 T 副本，因此，要使这个蕴涵成立，上下文中仍需要额外一个 T 。

但并非毫无办法；我们可以复制一个值，只要它被包裹在! 模态中。这正是!T 的用途：它是允许复制的T 的无限制版本。那么，让我们看看如何从空环境中推导出蕴含式!T —∘ (T⊗T) :

$$\dfrac{\dfrac{\dfrac{}{!T \vdash\ !T}\,\text{ID}}{!T \vdash T}\,!\text{D} \qquad \dfrac{\dfrac{}{!T \vdash\ !T}\,\text{ID}}{!T \vdash T}\,!\text{D}}{\dfrac{\dfrac{}{!T \vdash\ !T}\,\text{ID} \qquad !T,\ !T \vdash T \otimes T}{!T \vdash T \otimes T}\,\otimes\text{I}}\,!\text{C}}{\cdot \vdash\ !T \multimap (T \otimes T)}\,\multimap\text{I}$$

在这里，我们使用恒等和降格，从无限制的T中提取线性!T。积T⊗T需要两个副本，因此上下文中需要两个!T；这恰好对应于应用收缩规则 ILL/!C所需的设置。在下一节中，我们将看到这一推导如何对应于一个类型良好的程序。

1.2 线性 λ演算：线性逻辑的计算解释 c

线性逻辑可以有几种不同的线性 λ演算解释。第一种——沿用 [Andreoli 1992] 的术语，称为单子形式——源自对图 1.1 所示直觉主义线性逻辑自然演绎规则的直接项指派。这种单子形式的语法和类型规则记为 λ_{L1}，见图 1.2 和图 1.3，其设计深受 Bierman [1994] 工作的启发。

$$
\begin{array}{rcl}
v & ::= & \boldsymbol{\lambda} x \mapsto u \mid () \mid \mathsf{Inl}\, v \mid \mathsf{Inr}\, v \mid (v_1 , v_2) \mid \mathsf{Many}\, v \\
t, u & ::= & v \mid x \mid \mathsf{Inl}\, t \mid \mathsf{Inr}\, t \mid (t_1 , t_2) \mid \mathsf{Many}\, t \mid t\ t' \mid t \fatsemi t' \\
& \mid & \mathsf{case}\ t\ \mathsf{of}\ \{\mathsf{Inl}\, x_1 \mapsto u_1 , \mathsf{Inr}\, x_2 \mapsto u_2\} \mid \mathsf{case}\ t\ \mathsf{of}\ (x_1 , x_2) \mapsto u \\
& \mid & \mathsf{dup}\ t\ \mathsf{as}\ x_1 , x_2\ \mathsf{in}\ u \mid \mathsf{drop}\ t\ \mathsf{in}\ u \mid \mathsf{derelict}\ t \\
\\
\mathsf{T}, \mathsf{U} & ::= & \mathsf{T} \multimap \mathsf{U} \mid \mathsf{1} \mid \mathsf{T}_1 \oplus \mathsf{T}_2 \mid \mathsf{T}_1 \otimes \mathsf{T}_2 \mid\ !\mathsf{T} \\
\\
\Gamma & ::= & \cdot \mid x : \mathsf{T} \mid \Gamma_1 , \Gamma_2
\end{array}
$$

图 1.2：单子式表示中的线性 λ-演算语法（λ_{L1}）

$\boxed{\Gamma \vdash t : \mathsf{T}}$ （项的类型判断）

$$\frac{}{x : \mathsf{T} \vdash x : \mathsf{T}}\ \mathrm{ID} \qquad \frac{\Gamma,\ x : \mathsf{T} \vdash u : \mathsf{U}}{\Gamma \vdash \boldsymbol{\lambda} x \mapsto u : \mathsf{T} \multimap \mathsf{U}} \multimap\mathrm{I} \qquad \frac{}{\cdot \vdash () : \mathsf{1}}\ \mathsf{1}\mathrm{I} \qquad \frac{\Gamma \vdash t_1 : \mathsf{T}_1}{\Gamma \vdash \mathsf{Inl}\, t_1 : \mathsf{T}_1 \oplus \mathsf{T}_2} \oplus\mathrm{I}_1$$

$$\frac{\Gamma \vdash t_2 : \mathsf{T}_2}{\Gamma \vdash \mathsf{Inr}\, t_2 : \mathsf{T}_1 \oplus \mathsf{T}_2} \oplus\mathrm{I}_2 \qquad \frac{\Gamma_1 \vdash t_1 : \mathsf{T}_1 \quad \Gamma_2 \vdash t_2 : \mathsf{T}_2}{\Gamma_1,\ \Gamma_2 \vdash (t_1 , t_2) : \mathsf{T}_1 \otimes \mathsf{T}_2} \otimes\mathrm{I} \qquad \frac{\Gamma_1 \vdash t : \mathsf{T} \quad \Gamma_2 \vdash t' : \mathsf{T} \multimap \mathsf{U}}{\Gamma_1,\ \Gamma_2 \vdash t'\ t : \mathsf{U}} \multimap\mathrm{E}$$

$$\frac{\Gamma_1 \vdash t : \mathsf{1} \qquad \Gamma_2 \vdash u : \mathsf{U}}{\Gamma_1,\ \Gamma_2 \vdash t \fatsemi u : \mathsf{U}}\ \mathsf{1}\mathrm{E} \qquad \frac{\begin{array}{c}\Gamma_1 \vdash t : \mathsf{T}_1 \oplus \mathsf{T}_2 \\ \Gamma_2,\ x_1 : \mathsf{T}_1 \vdash u_1 : \mathsf{U} \\ \Gamma_2,\ x_2 : \mathsf{T}_2 \vdash u_2 : \mathsf{U}\end{array}}{\Gamma_1,\ \Gamma_2 \vdash \mathsf{case}\ t\ \mathsf{of}\ \{\mathsf{Inl}\, x_1 \mapsto u_1 , \mathsf{Inr}\, x_2 \mapsto u_2\} : \mathsf{U}} \oplus\mathrm{E}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash t : \mathsf{T}_1 \otimes \mathsf{T}_2 \\ \Gamma_2,\ x_1 : \mathsf{T}_1,\ x_2 : \mathsf{T}_2 \vdash u : \mathsf{U}\end{array}}{\Gamma_1,\ \Gamma_2 \vdash \mathsf{case}\ t\ \mathsf{of}\ (x_1 , x_2) \mapsto u : \mathsf{U}} \otimes\mathrm{E} \qquad \frac{!\Gamma \vdash t : \mathsf{T}}{!\Gamma \vdash \mathsf{Many}\, t : !\mathsf{T}}\ !\mathrm{P} \qquad \frac{\Gamma \vdash t : !\mathsf{T}}{\Gamma \vdash \mathsf{derelict}\ t : \mathsf{T}}\ !\mathrm{D}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash t : !\mathsf{T} \\ \Gamma_2,\ x_1 : !\mathsf{T},\ x_2 : !\mathsf{T} \vdash u : \mathsf{U}\end{array}}{\Gamma_1,\ \Gamma_2 \vdash \mathsf{dup}\ t\ \mathsf{as}\ x_1 , x_2\ \mathsf{in}\ u : \mathsf{U}}\ !\mathrm{C} \qquad \frac{\Gamma_1 \vdash t : !\mathsf{T} \quad \Gamma_2 \vdash u : \mathsf{U}}{\Gamma_1,\ \Gamma_2 \vdash \mathsf{drop}\ t\ \mathsf{in}\ u : \mathsf{U}}\ !\mathrm{W}$$

图 1.3：单子式表示中的线性 λ-演算类型规则（λ_{L1}–ty）

在 λ_{L1} 中， Γ 现在是从变量到类型的有限映射，可表示为变量绑定的集合 $x : \mathsf{T}$。与往常一样，上下文 Γ 中不允许出现重复的变量名。逗号运算符现在表示有限映射的不交并。

λ_{L1} 的项语法大部分沿用了简单类型 λ 演算的语法。单位类型 1 的消去通过 ; 运算符实现。和积类型的模式匹配通过 case 关键字实现。最后，我们新增了运算符 dup、drop 和 derelict，分别用于对类型为 !T 的非限制项进行收缩、弱化和降格。此外，语言还包含一个用于非限制项和值的数据构造器，分别记作 Many t 和 Many v。将项提升为非限制形式，可直接应用构造器 Many。为便于说明，我们使用语法糖 let x :=t in u，并将其编码为 (λx ↦ u) (t)。

上一节中线性蕴涵 $!\mathsf{T} \multimap (\mathsf{T} \otimes \mathsf{T})$ 的推导，现在对应于一个类型为 $!\mathsf{T} \multimap (\mathsf{T} \otimes \mathsf{T})$的类型正确项的类型推导。看看具体是怎样的：

$$\dfrac{\dfrac{}{x : !\mathsf{T} \vdash x : !\mathsf{T}}\,\text{ID} \qquad \dfrac{\dfrac{\dfrac{}{x_1 : !\mathsf{T} \vdash x_1 : !\mathsf{T}}\,\text{ID}}{x_1 : !\mathsf{T} \vdash \mathsf{derelict}\ x_1 : \mathsf{T}}\,!\text{D} \qquad \dfrac{\dfrac{}{x_2 : !\mathsf{T} \vdash x_2 : !\mathsf{T}}\,\text{ID}}{x_2 : !\mathsf{T} \vdash \mathsf{derelict}\ x_2 : \mathsf{T}}\,!\text{D}}{x_1 : !\mathsf{T},\ x_2 : !\mathsf{T} \vdash (\mathsf{derelict}\ x_1\ ,\ \mathsf{derelict}\ x_2) : \mathsf{T} \otimes \mathsf{T}}\,\otimes\text{I}}{x : !\mathsf{T} \vdash \mathsf{dup}\ x\ \mathsf{as}\ x_1\ ,\ x_2\ \mathsf{in}\ (\mathsf{derelict}\ x_1\ ,\ \mathsf{derelict}\ x_2) : \mathsf{T} \otimes \mathsf{T}}\,!\text{C}}{\bullet \vdash \lambda x \mapsto \mathsf{dup}\ x\ \mathsf{as}\ x_1\ ,\ x_2\ \mathsf{in}\ (\mathsf{derelict}\ x_1\ ,\ \mathsf{derelict}\ x_2) : !\mathsf{T} \multimap (\mathsf{T} \otimes \mathsf{T})}\,\multimap\text{I}$$

这种转换非常直接。唯一的变化是，我们必须为类型上下文中类型相同的假设指定不同的变量名；而在直觉主义线性逻辑中，同一公式的多个副本并没有命名这一概念。我们还可以看到，由于需要手动进行复制和降格，处理无限制假设可能相当繁琐。最后，在这个系统中，替换并不具有可容许性，详见 ⟦Wadler，1992⟧。下一节将介绍一种规避这些问题的方法。

1.3 对无限制资源的便捷处理

在单子式表示 λ_{L1}中，由于需要显式地进行收缩、弱化和降格，使用无限制项可能变得非常冗长而繁琐。线性 λ演算的第二种、更强大的表示法称为二元 表示法，或称 λ_{L2}；它通过在每个判断中使用两个类型上下文来减轻这一负担，一个用于线性变量，另一个用于无限制变量。λ_{L2} 的语法和类型规则见图 1.4 和图 1.5。

在 λ_{L2} 中，不再有针对非受限资源的收缩、弱化和降格规则。取而代之的是，每个判断都配有第二个上下文 υ，其中保存着可以不受限制地使用的变量绑定。构造 ↦ 让我们能够将类型为 !T 的项 t 作为类型为 T、存在于非受限上下文 υ 中的变量 x 来访问。需要注意的是， ↦ 并非降格：在主体 u 中仍可以多次使用 x，也可以将 x 重新包装为 Many x，从而重建 t；而对于 =，这就不可能了。从本质上说，我们可以将 $\Gamma;\upsilon$ （属于 λ_{L2} ）这对上下文理解为单一上下文 Γ，$!\upsilon$ （属于 λ_{L1}），其中 $!\upsilon$ 是一个上下文，它与 υ 具有相同的变量绑定，只不过所有类型前面都加上了 !。这种表示方式的优点在于，它使替换对 λ_{L2} 成为可容许的，而对 λ_{L1} 则并非如此。

在 λ_{L2} 中，只要某条规则的前提中有两个子项，就会隐式地收缩非受限资源。事实上，非受限上下文 υ 会在两个前提中重复出现，特别是在规则 λ_{L2}–ty/⊗I 中可以看到；而线性上下文 Γ 则必须拆分为两个不相交部分。所有非受限

$$v ::= \lambda x \mapsto u \mid () \mid Inl\ v \mid Inr\ v \mid (v_1, v_2) \mid Many\ v \quad t, u ::= v \mid x \mid Inl\ t \mid Inr\ t \mid (t_1, t_2) \mid Many\ t \mid t\ t' \mid t\ ;\ t' \mid 分情况\ t\ of\ \{Inl x_1 \mapsto u_1, Inr\ x_2 \mapsto u_2\} \mid 分情况\ t\ of\ (x_1, x_2) \mapsto u \mid 分情况\ t\ of\ Many\ x \mapsto u$$

$$\mathsf{T}, \mathsf{U} ::= \mathsf{T} \multimap \mathsf{U} \mid \mathsf{1} \mid \mathsf{T}_1 \oplus \mathsf{T}_2 \mid \mathsf{T}_1 \otimes \mathsf{T}_2 \mid \mathsf{!T}$$

$$\Gamma ::= \cdot \mid x : \mathsf{T} \mid \Gamma_1, \Gamma_2$$
$$\mho ::= \cdot \mid x : \mathsf{T} \mid \mho_1, \mho_2$$

图1.4：二元表示中的线性 λ-演算语法（λ_{L2}）

$\boxed{\Gamma\ ;\ \mho \vdash t : \mathsf{T}}$ （项的类型判断）

$$\frac{}{x : \mathsf{T}\ ;\ \mho \vdash x : \mathsf{T}}\ \mathrm{ID}_{\mathrm{LIN}} \qquad \frac{}{\cdot\ ;\ \mho,\ x : \mathsf{T} \vdash x : \mathsf{T}}\ \mathrm{ID}_{\mathrm{UR}} \qquad \frac{\Gamma,\ x : \mathsf{T}\ ;\ \mho \vdash u : \mathsf{U}}{\Gamma\ ;\ \mho \vdash \lambda x \mapsto u : \mathsf{T} \multimap \mathsf{U}} \multimap \mathrm{I}$$

$$\frac{}{\cdot\ ;\ \mho \vdash () : \mathsf{1}}\ \mathsf{1}\mathrm{I} \qquad \frac{\Gamma\ ;\ \mho \vdash t_1 : \mathsf{T}_1}{\Gamma\ ;\ \mho \vdash \mathsf{Inl}\ t_1 : \mathsf{T}_1 \oplus \mathsf{T}_2} \oplus \mathrm{I}_1 \qquad \frac{\Gamma\ ;\ \mho \vdash t_2 : \mathsf{T}_2}{\Gamma\ ;\ \mho \vdash \mathsf{Inr}\ t_2 : \mathsf{T}_1 \oplus \mathsf{T}_2} \oplus \mathrm{I}_2$$

$$\frac{\Gamma_1\ ;\ \mho \vdash t_1 : \mathsf{T}_1 \quad \Gamma_2\ ;\ \mho \vdash t_2 : \mathsf{T}_2}{\Gamma_1,\ \Gamma_2\ ;\ \mho \vdash (t_1, t_2) : \mathsf{T}_1 \otimes \mathsf{T}_2} \otimes \mathrm{I} \qquad \frac{\cdot\ ;\ \mho \vdash t : \mathsf{T}}{\cdot\ ;\ \mho \vdash \mathsf{Many}\ t : \mathsf{!T}}\ \mathsf{!}\mathrm{I} \qquad \frac{\Gamma_1\ ;\ \mho \vdash t : \mathsf{T} \quad \Gamma_2\ ;\ \mho \vdash t' : \mathsf{T} \multimap \mathsf{U}}{\Gamma_1,\ \Gamma_2\ ;\ \mho \vdash t'\ t : \mathsf{U}} \multimap \mathrm{E}$$

$$\frac{\Gamma_1\ ;\ \mho \vdash t : \mathsf{1} \quad \Gamma_2\ ;\ \mho \vdash u : \mathsf{U}}{\Gamma_1,\ \Gamma_2\ ;\ \mho \vdash t\ ;\ u : \mathsf{U}}\ \mathsf{1}\mathrm{E} \qquad \frac{\Gamma_1\ ;\ \mho \vdash t : \mathsf{T}_1 \oplus \mathsf{T}_2 \quad \Gamma_2,\ x_1 : \mathsf{T}_1\ ;\ \mho \vdash u_1 : \mathsf{U} \quad \Gamma_2,\ x_2 : \mathsf{T}_2\ ;\ \mho \vdash u_2 : \mathsf{U}}{\Gamma_1,\ \Gamma_2\ ;\ \mho \vdash \mathsf{case}\ t\ \mathsf{of}\ \{\mathsf{Inl}\ x_1 \mapsto u_1,\ \mathsf{Inr}\ x_2 \mapsto u_2\} : \mathsf{U}} \oplus \mathrm{E}$$

$$\frac{\Gamma_1\ ;\ \mho \vdash t : \mathsf{T}_1 \otimes \mathsf{T}_2 \quad \Gamma_2,\ x_1 : \mathsf{T}_1,\ x_2 : \mathsf{T}_2\ ;\ \mho \vdash u : \mathsf{U}}{\Gamma_1,\ \Gamma_2\ ;\ \mho \vdash \mathsf{case}\ t\ \mathsf{of}\ (x_1, x_2) \mapsto u : \mathsf{U}} \otimes \mathrm{E} \qquad \frac{\Gamma_1\ ;\ \mho \vdash t : \mathsf{!T} \quad \Gamma_2\ ;\ \mho,\ x : \mathsf{T} \vdash u : \mathsf{U}}{\Gamma_1,\ \Gamma_2\ ;\ \mho \vdash \mathsf{case}\ t\ \mathsf{of}\ \mathsf{Many}\ x \mapsto u : \mathsf{U}}\ \mathsf{!}\mathrm{E}$$

图1.5：二元表示中的线性 λ-演算类型规则（λ_{L2}–ty）

因此，变量绑定会传播到类型树的叶节点，也就是没有前提的规则 λ_{L2}–ty/id$_{\text{Lin}}$、 λ_{L2}–ty/id$_{\text{Ur}}$ 和 λ_{L2}–ty/1I。这三条规则会丢弃非受限上下文 υ 中所有未使用的绑定，并执行若干次隐式弱化。

最后，这个系统有两条恒等规则。第一条， λ_{L2}–ty/id$_{\text{Lin}}$，是通常的线性恒等规则：它会消耗线性类型上下文中存在的变量x。由于线性类型上下文必须在兄弟子项之间拆分为不相交部分（当一条规则有多个前提时，例如乘积规则），这意味着x只能在其中一个子项中使用。第二条恒等规则， λ_{L2}–ty/id$_{\text{Ur}}$，是非受限恒等规则：它允许我们在需要线性变量的位置使用来自非受限类型上下文的变量x，从而执行某种隐式降格。由于非受限类型上下文会在兄弟子项之间复制，因此一个非受限变量可以在多个子项中使用。例如，下面的推导无效，因为我们没有遵守对的结果类型上下文中逗号算子的不相交条件：

$$\frac{\dfrac{}{x' : \mathsf{T} \,;\, \bullet \vdash x' : \mathsf{T}}\,\mathrm{ID}_{\text{Lin}} \qquad \dfrac{}{x' : \mathsf{T} \,;\, \bullet \vdash x' : \mathsf{T}}\,\mathrm{ID}_{\text{Lin}}}{x' : \mathsf{T},\ x' : \mathsf{T} \,;\, \bullet \vdash (x', x') : \mathsf{T} \otimes \mathsf{T}}\,\otimes\mathrm{I}$$

但是，下面的推导是有效的，因为非受限类型上下文会在兄弟子项之间复制：

$$\frac{\dfrac{}{\bullet \,;\, x' : \mathsf{T} \vdash x' : \mathsf{T}}\,\mathrm{ID}_{\text{Ur}} \qquad \dfrac{}{\bullet \,;\, x' : \mathsf{T} \vdash x' : \mathsf{T}}\,\mathrm{ID}_{\text{Ur}}}{\bullet \,;\, x' : \mathsf{T} \vdash (x', x') : \mathsf{T} \otimes \mathsf{T}}\,\otimes\mathrm{I}$$

此时，我们已经非常接近于重新构造一个类型为 $!\mathsf{T} \multimap (\mathsf{T} \otimes \mathsf{T})$ 的项，就像我们在上一节中所做的那样。让我们完成这个示例：

$$\frac{\dfrac{\dfrac{}{x : !\mathsf{T} \,;\, \bullet \vdash x : !\mathsf{T}}\,\mathrm{ID}_{\text{Lin}} \qquad \dfrac{\dfrac{}{\bullet \,;\, x' : \mathsf{T} \vdash x' : \mathsf{T}}\,\mathrm{ID}_{\text{Ur}} \qquad \dfrac{}{\bullet \,;\, x' : \mathsf{T} \vdash x' : \mathsf{T}}\,\mathrm{ID}_{\text{Ur}}}{\bullet \,;\, x' : \mathsf{T} \vdash (x', x') : \mathsf{T} \otimes \mathsf{T}}\,\otimes\mathrm{I}}{x : !\mathsf{T} \,;\, \bullet \vdash \mathsf{case}\ x\ \mathsf{of}\ \mathsf{Many}\ x' \mapsto (x', x') : \mathsf{T} \otimes \mathsf{T}}\,!\mathrm{E}}{\bullet \,;\, \bullet \vdash \boldsymbol{\lambda} x \mapsto \mathsf{case}\ x\ \mathsf{of}\ \mathsf{Many}\ x' \mapsto (x', x') : !\mathsf{T} \multimap \mathsf{T} \otimes \mathsf{T}}\,\multimap\mathrm{I}$$

如果我们的非受限类型上下文中存在任何无用的变量绑定（比如 $\upsilon = y :\ !\mathsf{U}$），推导仍然成立，在项级别无需做任何更改；这些无用的绑定只会在整个类型树中持续保留，并在叶节点处被消除（通过叶规则中发生的隐式弱化）：

$$\frac{\dfrac{\dfrac{}{x : !\mathsf{T} \,;\, y : !\mathsf{U} \vdash x : !\mathsf{T}}\,\mathrm{ID}_{\text{Lin}} \qquad \dfrac{\dfrac{}{\bullet \,;\, y : !\mathsf{U},\ x' : \mathsf{T} \vdash x' : \mathsf{T}}\,\mathrm{ID}_{\text{Ur}} \qquad \dfrac{}{\bullet \,;\, y : !\mathsf{U},\ x' : \mathsf{T} \vdash x' : \mathsf{T}}\,\mathrm{ID}_{\text{Ur}}}{\bullet \,;\, y : !\mathsf{U},\ x' : \mathsf{T} \vdash (x', x') : \mathsf{T} \otimes \mathsf{T}}\,\otimes\mathrm{I}}{x : !\mathsf{T} \,;\, y : !\mathsf{U} \vdash \mathsf{case}\ x\ \mathsf{of}\ \mathsf{Many}\ x' \mapsto (x', x') : \mathsf{T} \otimes \mathsf{T}}\,!\mathrm{E}}{\bullet \,;\, y : !\mathsf{U} \vdash \boldsymbol{\lambda} x \mapsto \mathsf{case}\ x\ \mathsf{of}\ \mathsf{Many}\ x' \mapsto (x', x') : !\mathsf{T} \multimap \mathsf{T} \otimes \mathsf{T}}\,\multimap\mathrm{I}$$

正如我们刚才所演示的，在二元系统 λ_{L2} 中管理无限制的假设要容易得多。类型树更短，或许也更易于阅读。

此外，从 λ_{L1} 表示法转到 λ_{L2}：Andreoli ⟦1992⟧ 给出了详细证明，说明 λ_{L1}–$_{\text{TY}}$ 与 λ_{L2}–$_{\text{TY}}$ 等价；换言之，程序 t 在上下文对 $\Gamma;\upsilon$ 中的 λ_{L2}–$_{\text{TY}}$ 类型成立，当且仅当它在上下文 $\Gamma, !$ λ_{L1}–$_{\text{TY}}$中类型成立。

1.4 通过分级模态方法回到单一上下文

到目前为止，我们只考虑了线性类型系统；除此之外，它们与常见的简单类型 λ 演算相当标准。考虑到未来的需求，本文后续部分需要在整个类型系统中携带比单纯线性性更多的信息和限制。

最直接的做法是，对于每个需要的新模态，将类型上下文进行相乘。问题在于，这种方法并不真正具备可扩展性：最终，我们需要为模态的每种可能组合各设置一个上下文（如果模态表示正交原则），因此规模会迅速增长。

事实上，不必想得太远，如果我们希望对线性性进行更精细的控制，就已经会遇到一个问题。如果我们希望允许变量被使用固定次数，而不仅仅是一次或无限次，该怎么办？在 ⟦Jean-Yves Girard、Scedrov 和 Scott， 1992⟧, Girard 考虑了线性逻辑的有界扩展，其中假设的使用次数可以被限制为任意值 m——称为重数——而不再仅仅是线性或无限制的。为此，他为 ! 模态添加了一个索引 m，用于指定一个假设应被使用多少次。我们称 $!_m$ 为分级模态。

采用包含任意数量元素的模态族（$!_m$）$_{m\in M}$，而不是原始线性逻辑中的单一!模态，这意味着我们无法像二元表示那样，为每个模态分别设置一个上下文。一种解决方案是回到单子式表示，在这种表示中，变量的类型会一直携带其模态，直到最终使用它们时才不再如此。我们还需要一个新运算符，将$!_m T$ 转换为一对$!_{m-1}T \otimes T$，这样便能从一个由允许多次使用的模态包装的值中提取出一次使用权。正如所想，这种方法很快就会变得不切实际。

幸运的是，我们有办法解决这个问题。我们不再让m 成为模态的一部分，进而成为项类型的一部分，而是将它作为注解嵌入变量绑定中。我们可以不用$x : S,\ y : !_m T,\ z : !_n U \vdash \ldots$，而改用 $x :_1 S,\ y :_m T,\ z :_n U \vdash$。我们将称绑定上的这些新注解为模态。注意，每个绑定都会配有一个模式 m，包括此前类型上没有模态的线性绑定[4]。除了为绑定添加模态外，我们还将定义若干模式上的代数运算（之后再推广到绑定和类型上下文），以更简单、更全面的方式编码系统的（子）结构规则，无需再为它们显式设置语法，例如弃用 或dup 。

通过模态方法，我们重新得到了一种类似二元系统的系统，其中收缩、弱化和降格都可以方便地隐式处理，同时不会失去对资源使用的任何控制能力，也不会导致上下文数量激增！

此外，我们基于一项关键洞见——最初由 ⟦Brunel 等人、 2014⟧ 以及 ⟦Ghica 和 Smith 提出 2014⟧——即为模态集合赋予半环结构，就足以用代数方式表达我们所需的所有类型上下文操作。其思路是定义模态上的两个主要操作：乘法 $\cdot$，用于建模模态如何在嵌套表达式或函数调用中复合（例如，若函数 f 以模态 2使用其参数，而 g 也以模态2使用其参数，那么 $f\,(g\,x)$ 就以模态 $2 \cdot 2$ 使用 x）；以及加法 $+$，用于描述变量在多个子项中使用时的情况（例如，若子项 t 以模态 2 使用 x，而u 以模态3使用 x，那么$t \ ; u$ 就以模态 $2 + 3$ 使用 x）。通过将类型系统的结构限制抽象为具有半环结构的模态系统，我们使类型系统更具可扩展性：如果要扩展类型系统，并且能够将扩展表示为一个新半环，那么几乎无需修改类型规则，只需调整底层的模态半环。接着，我们按以下方式将模态运算符提升到变量绑定和类型上下文：

[4]这种为每个绑定都指定一个模式的统一方法似乎源自 ⟦Ghica 和 Smith、 2014⟧ 以及 ⟦Petricek、Orchard 和 Mycroft 2014⟧。

$$v ::= \lambda x_{\,m} \mapsto u \mid () \mid Inl\, v \mid Inr\, v \mid (v_1, v_2) \mid Mod_n\, v \qquad t, u ::= v \mid x \mid Inl\, t \mid Inr\, t \mid (t_1, t_2) \mid Mod_n\, t$$
$$\mid t\, t' \mid t \mathbin{;} t' \mid \text{对 } t \text{ 作如下分情况：} \{Inl x_1 \mapsto u_1, Inr\, x_2 \mapsto u_2\} \mid \text{对 } t \text{ 作如下分情况：} (x_1, x_2) \mapsto u \mid \text{对 } t \text{ 作如下分情况：} Mod_n\, x \mapsto u$$

$$\mathsf{T}, \mathsf{U} ::= \mathsf{T}_{\,m\!\multimap}\mathsf{U} \mid \mathbf{1} \mid \mathsf{T}_1 \oplus \mathsf{T}_2 \mid \mathsf{T}_1 \otimes \mathsf{T}_2 \mid !_n \mathsf{T}$$
$$m, n ::= 1 \mid \omega$$

$$\Gamma ::= \cdot \mid x :_m \mathsf{T} \mid \Gamma_1, \Gamma_2$$

图1.6：模态表示中的线性 λ演算语法（λ_{Lm}）

$$\begin{array}{lcll} n \cdot \cdot & \triangleq & \cdot & \\ n \cdot (x :_m \mathsf{T}, \Gamma) & \triangleq & (x :_{n \cdot m} \mathsf{T}), n \cdot \Gamma & \\ \\ \cdot + \Gamma & \triangleq & \Gamma & \\ (x :_m \mathsf{T}, \Gamma_1) + \Gamma_2 & \triangleq & x :_m \mathsf{T}, (\Gamma_1 + \Gamma_2) & \text{if } x \notin \Gamma_2 \\ (x :_m \mathsf{T}, \Gamma_1) + (x :_{m'} \mathsf{T}, \Gamma_2) & \triangleq & x :_{m+m'} \mathsf{T}, (\Gamma_1 + \Gamma_2) & \end{array}$$

现在，我们将介绍直觉主义 λ-演算的具体模态表述，并将其记为 λ_{Lm}。事实上，这里我们并未使用完整的半环，因为我们不需要零元素来表示线性性，这一点与 λ_{L1} 和 λ_{L2} 中不同。相反，我们使用无零元半环来表示模态——下文仍称之为半环——它依然保留了 [Ghica 和 Smith, 2014] 中所述的所有代数优势。我们采用的半环仅包含两个元素：1，表示必须以线性方式管理的变量；以及 ω，表示无限制变量。下面给出相应的运算表：

+	1	ω
1	ω	ω
ω	ω	ω

·	1	ω
1	1	ω
ω	ω	ω

现在，模态的半环已经完全定义，我们可以转向 λ_{Lm} 的语法和类型规则，见图 1.6 和图 1.7。与 λ_{L2} 相比，主要变化在于，$!_m \mathsf{T}$ 的构造器现在是 $\mathsf{Mod}_m\, t$（而不是 $\mathsf{Many} t$，对应于 $!\mathsf{T}$）。

在 λ_{Lm} 中，我们回到了单一的恒等规则 λ_{Lm}–ty/id，该规则要求 x 以模式 m 出现在类型上下文中，而该模式必须与一次线性使用兼容（我们记作 $1 \leqslant m$）。在我们的半环中，只有两个元素，因此 $1 \leqslant 1$ 和 $1 \leqslant \omega$ 都成立，所以 x 实际上可以具有任意模式（线性或非线性，因而涵盖 λ_{L2}–ty/$\mathrm{id}_{\mathrm{Lin}}$ 和 λ_{L2}–ty/$\mathrm{id}_{\mathrm{Ur}}$）；但在更复杂的模态系统中，并非所有模式都必须与半环的单位兼容[5]。规则 λ_{Lm}–ty/id 和 λ_{Lm}–ty/1I 还允许丢弃仅由非线性绑定组成的任意上下文，记作 $\omega \cdot \Gamma$（等同于 λ_{L1}中的记号 $!\Gamma$），因此它们执行了与 λ_{L2}中相同的隐式弱化。

λ_{Lm} 中提到两个子项的每条规则，都会在规则的结论中对类型上下文使用新定义的 $+$ 运算符。如果同一个变量x 在 Γ_1 和 Γ_2 中都需要（模式为1 或 ω），那么 $\Gamma_1 + \Gamma_2$ 将包含绑定$x :_\omega \mathsf{T}$。换句话说，得益于 $+$ 运算符，超项会根据子项使用x的次数，自动推断x 需要是线性的还是不受限的。这种做法

[5]关于并非所有模式都与单位兼容的半环示例，请参见第 2.5.2 节。

$\boxed{\Gamma \vdash t : \mathsf{T}}$ *(Typing judgment for terms)*

$$\frac{1 \leq \mathsf{m}}{\omega \cdot \Gamma,\ x :_{\mathsf{m}} \mathsf{T} \vdash x : \mathsf{T}}\ \text{ID} \qquad \frac{\Gamma,\ x :_{\mathsf{m}} \mathsf{T} \vdash u : \mathsf{U}}{\Gamma \vdash \boldsymbol{\lambda} x \,{}_{\mathsf{m}}\!\mapsto u : \mathsf{T} \multimap_{\mathsf{m}} \mathsf{U}} \multimap\text{I} \qquad \frac{}{\omega \cdot \Gamma \vdash () : 1}\ 1\text{I}$$

$$\frac{\Gamma \vdash t_1 : \mathsf{T}_1}{\Gamma \vdash \mathsf{Inl}\, t_1 : \mathsf{T}_1 \oplus \mathsf{T}_2} \oplus\text{I}_1 \qquad \frac{\Gamma \vdash t_2 : \mathsf{T}_2}{\Gamma \vdash \mathsf{Inr}\, t_2 : \mathsf{T}_1 \oplus \mathsf{T}_2} \oplus\text{I}_2 \qquad \frac{\Gamma_1 \vdash t_1 : \mathsf{T}_1 \quad \Gamma_2 \vdash t_2 : \mathsf{T}_2}{\Gamma_1 + \Gamma_2 \vdash (t_1, t_2) : \mathsf{T}_1 \otimes \mathsf{T}_2} \otimes\text{I}$$

$$\frac{\Gamma \vdash t : \mathsf{T}}{\mathsf{n} \cdot \Gamma \vdash \mathsf{Mod}_{\mathsf{n}}\, t : !_{\mathsf{n}} \mathsf{T}}\ !\text{I} \qquad \frac{\Gamma_1 \vdash t : \mathsf{T} \quad \Gamma_2 \vdash t' : \mathsf{T} \multimap_{\mathsf{m}} \mathsf{U}}{\mathsf{m} \cdot \Gamma_1 + \Gamma_2 \vdash t'\ t : \mathsf{U}} \multimap\text{E} \qquad \frac{\Gamma_1 \vdash t : 1 \quad \Gamma_2 \vdash u : \mathsf{U}}{\Gamma_1 + \Gamma_2 \vdash t \fatsemi u : \mathsf{U}}\ 1\text{E}$$

$$\frac{\Gamma_1 \vdash t : \mathsf{T}_1 \oplus \mathsf{T}_2 \quad \Gamma_2,\ x_1 :_1 \mathsf{T}_1 \vdash u_1 : \mathsf{U} \quad \Gamma_2,\ x_2 :_1 \mathsf{T}_2 \vdash u_2 : \mathsf{U}}{\Gamma_1 + \Gamma_2 \vdash \mathsf{case}\ t\ \mathsf{of}\ \{\mathsf{Inl}\, x_1 \mapsto u_1\,;\ \mathsf{Inr}\, x_2 \mapsto u_2\} : \mathsf{U}} \oplus\text{E}$$

$$\frac{\Gamma_1 \vdash t : \mathsf{T}_1 \otimes \mathsf{T}_2 \quad \Gamma_2,\ x_1 :_1 \mathsf{T}_1,\ x_2 :_1 \mathsf{T}_2 \vdash u : \mathsf{U}}{\Gamma_1 + \Gamma_2 \vdash \mathsf{case}\ t\ \mathsf{of}\ (x_1, x_2) \mapsto u : \mathsf{U}} \otimes\text{E}$$

$$\frac{\Gamma_1 \vdash t : !_{\mathsf{n}} \mathsf{T} \quad \Gamma_2,\ x :_{\mathsf{n}} \mathsf{T} \vdash u : \mathsf{U}}{\Gamma_1 + \Gamma_2 \vdash \mathsf{case}\ t\ \mathsf{of}\ \mathsf{Mod}_{\mathsf{n}}\, x \mapsto u : \mathsf{U}}\ !\text{E}$$

图 1.7：模态表示中的线性 λ-演算类型规则（λ_{Lm}–ty）

与$\lambda_{L1,2}$ 中对子项进行线性上下文拆分，以及 λ_{L2}中对子项进行非受限上下文复制的做法截然不同。我们认为，得益于模态半环的代数性质，这使得系统的表述更加流畅（至少在理论层面如此；为这种线性类型系统实现类型检查可能会更棘手）。

与 λ_{L2}中大致相同，指数模态$!_{\mathsf{m}}$ 通过分情况$t\ \mathsf{of}\ \mathsf{Mod}_{\mathsf{m}}\, x \mapsto u$ 表达式消除，该表达式将判别式绑定到一个新变量x，其模式为m。若有需要，可以用$\mathsf{Mod}_{\mathsf{m}}\, x$重新构造原始装箱值；实际上，与 λ_{L2}中一样，$!_{\mathsf{m}}$ 的消除规则并非降格。

该系统最后一项具体化是，函数箭头 $\multimap$ 现在携带一个模式 m，并据此绑定其参数。此前，在 λ_{L1},2 中，函数参数总是线性的，这意味着函数体中对应的变量绑定也是线性的。而在 λ_{Lm} 中，我们可以在任意模式 m 下绑定参数——函数应用的规则 λ_{Lm}–ty/$\multimap$E 体现了这一点：在该规则的结论中，传递给函数的参数所对应的类型上下文 Γ_1 会乘以 m。实际上，与先前的表述相比，这并不会改变系统的表达能力。例如，函数$\boldsymbol{\lambda} x \,{}_{\mathsf{m}}\!\mapsto u ::: \mathsf{T} \multimap_{\mathsf{m}} \mathsf{U}$ 可以编码为 $\boldsymbol{\lambda} x' \mapsto \mathsf{case}\ x'\ \mathsf{of}\ \mathsf{Mod}_{\mathsf{m}}\, x \mapsto u ::: !_{\mathsf{m}} \mathsf{T} \multimap \mathsf{U}$，只使用纯线性函数箭头（在调用处，我们只需将参数显式地包装在指数模态中，即将 $f\ x$ 替换为 $f'\ (\mathsf{Mod}_{\mathsf{m}}\, x)$）。但由于在这种表述中，每个变量绑定无论如何都必须指定一个模式，因此允许程序员自行选择任意模式 m，而不只是默认使用 1，便顺理成章——这样可以减少显式包装和拆包模态所带来的冗余。

让我们更新一下贯穿全文的示例。项语法没有变化，我们只看到类型上下文有所演变（现在只有一个带模式注释的类型上下文，而不是两个）：

$$\dfrac{\dfrac{\dfrac{}{y :_{\omega} \mathsf{U},\ x :_{1} !\mathsf{T} \vdash x : !\mathsf{T}}\ \mathrm{ID}\quad \dfrac{\dfrac{1\leqslant 1}{y :_{\omega} \mathsf{U},\ x' :_{1} \mathsf{T} \vdash x' : \mathsf{T}}\ \mathrm{ID}\quad \dfrac{1\leqslant 1}{y :_{\omega} \mathsf{U},\ x' :_{1} \mathsf{T} \vdash x' : \mathsf{T}}\ \mathrm{ID}}{y :_{\omega} \mathsf{U},\ x' :_{\omega} \mathsf{T} \vdash (x', x') : \mathsf{T}\otimes\mathsf{T}}\ \otimes\mathrm{I}}{y :_{\omega} \mathsf{U},\ x :_{1} !\mathsf{T} \vdash \mathsf{case}\ x\ \mathsf{of}\ \mathsf{Mod}_{\omega}\ x' \mapsto (x', x') : \mathsf{T}\otimes\mathsf{T}}\ !\mathrm{E}}{y :_{\omega} \mathsf{U} \vdash \boldsymbol{\lambda} x\ {}_{1}\!\mapsto \mathsf{case}\ x\ \mathsf{of}\ \mathsf{Mod}_{\omega}\ x' \mapsto (x', x') : !\mathsf{T}\ {}_{1}\!\multimap \mathsf{T}\otimes\mathsf{T}}\ \multimap\mathrm{I}$$

在 λ_{Lm} 中，得益于 $+$ 作用于上下文的方式，我们只需将类型上下文中无限制且未使用的部分（在此例中为 $y :_{\omega} \mathsf{U}$ ）带到至少一个叶节点，而不是所有叶节点。比如，下面的写法同样有效：

$$\dfrac{\dfrac{\dfrac{}{y :_{\omega} \mathsf{U},\ x :_{1} !\mathsf{T} \vdash x : !\mathsf{T}}\ \mathrm{ID}\quad \dfrac{\dfrac{1\leqslant 1}{x' :_{1} \mathsf{T} \vdash x' : \mathsf{T}}\ \mathrm{ID}\quad \dfrac{1\leqslant 1}{x' :_{1} \mathsf{T} \vdash x' : \mathsf{T}}\ \mathrm{ID}}{x' :_{\omega} \mathsf{T} \vdash (x', x') : \mathsf{T}\otimes\mathsf{T}}\ \otimes\mathrm{I}}{y :_{\omega} \mathsf{U},\ x :_{1} !\mathsf{T} \vdash \mathsf{case}\ x\ \mathsf{of}\ \mathsf{Mod}_{\omega}\ x' \mapsto (x', x') : \mathsf{T}\otimes\mathsf{T}}\ !\mathrm{E}}{y :_{\omega} \mathsf{U} \vdash \boldsymbol{\lambda} x\ {}_{1}\!\mapsto \mathsf{case}\ x\ \mathsf{of}\ \mathsf{Mod}_{\omega}\ x' \mapsto (x', x') : !\mathsf{T}\ {}_{1}\!\multimap \mathsf{T}\otimes\mathsf{T}}\ \multimap\mathrm{I}$$

我们也可以选择在树顶端 λ_{Lm}–ty/id 规则的两个相同实例中，以模态 ω 为 x' 赋予类型；这不会影响推导的其余部分，因为 $1+1$ 和 $\omega+\omega$ 最终都会给出 ω。现在，在 λ_{L1}–ty 中为一个项赋予类型时，我们有了几个需要作出的选择，而在 λ_{L2}–ty 中并没有这些选择；这正是试图证明这两个系统等价时唯一不平凡的方面。

最后，利用我们刚才讨论的函数箭头上的模态特性，我们可以给出这个函数的一个替代性且更简洁的版本。这个新版本的类型为$\mathsf{T}\ {}_{\omega}\!\multimap \mathsf{T}\otimes\mathsf{T}$，而不是$!\mathsf{T}\ {}_{1}\!\multimap \mathsf{T}\otimes\mathsf{T}$：

$$\dfrac{\dfrac{\dfrac{1\leqslant 1}{y :_{\omega} \mathsf{U},\ x' :_{1} \mathsf{T} \vdash x' : \mathsf{T}}\ \mathrm{ID}\quad \dfrac{1\leqslant 1}{y :_{\omega} \mathsf{U},\ x' :_{1} \mathsf{T} \vdash x' : \mathsf{T}}\ \mathrm{ID}}{y :_{\omega} \mathsf{U},\ x' :_{\omega} \mathsf{T} \vdash (x', x') : \mathsf{T}\otimes\mathsf{T}}\ \otimes\mathrm{I}}{y :_{\omega} \mathsf{U} \vdash \boldsymbol{\lambda} x'\ {}_{\omega}\!\mapsto (x', x') : \mathsf{T}\ {}_{\omega}\!\multimap \mathsf{T}\otimes\mathsf{T}}\ \multimap\mathrm{I}$$

1.5 深层模态：通过数据结构的字段投影模态

在 Girard 最初提出的线性逻辑中（参见直觉主义线性逻辑及上文的介绍 λ_{L1}），$!\mathsf{T}\otimes!\mathsf{U}$ 与$!(\mathsf{T}\otimes\mathsf{U})$之间只有一个单向态射；我们无法从$!(\mathsf{T}\otimes\mathsf{U})$ 转换为$!\mathsf{T}\otimes!\mathsf{U}$。换句话说，一个无限制对$!(\mathsf{T}\otimes\mathsf{U})$并不允许其分量被无限制地使用。这个对可以任意复制或丢弃，但要使用它，必须先解除其指数模态，使其变为$\mathsf{T}\otimes\mathsf{U}$；此时，$\mathsf{T}$ 或U都不再能被复制或丢弃。因此，T 和U 必须恰好使用相同的次数，尽管它们属于一个无限制对$!$。

在 λ_{L2}（分别在 λ_{Lm}）中，情况并无不同：尽管类型为 $!(\mathsf{T}\otimes\mathsf{U})$ 的对可以绑定到无限制绑定 $x :\ \mathsf{T}\otimes\mathsf{U} \in \mho$（分别为 $x :\ {}_{\omega}\mathsf{T}\otimes\mathsf{U}$，位于 λ_{Lm} 中），并且无需降格即可这样使用，但当对其进行模式匹配时（通过规则 $\lambda_{L2.\ m}$–ty/$\otimes$E），仍会发生隐式降格，并且会在该对的分量上建立仅限线性的绑定：$x_1 :\ \mathsf{T}$，$x_2 :\ \mathsf{U}$，位于线性类型上下文中（分别为 $x_1 :\ {}_{1}\mathsf{T}$，$x_2 :\ {}_{1}\mathsf{U}$，位于 λ_{Lm} 中）。

假设我们想编写 `fst` 函数，用于提取对的第一个元素。在非线性语言中，我们可能会这样编写：

$\mathsf{fst}_{\mathsf{nl}} ::: \mathsf{T} \otimes \mathsf{U} \to \mathsf{T}$
$\mathsf{fst}_{\mathsf{nl}} \triangleq$ 分情况 $\boldsymbol{\lambda} x \mapsto$
x of $(x_1 , x_2) \mapsto x_1$

但是在 λ_{Lm} 中，甚至在 λ_{L1} 中，我们不允许这样做。一个直接的想法是为该函数指定类型签名 $\mathsf{T} \otimes \mathsf{U} \multimap_{\omega} \mathsf{T}$，但这仍然没有帮助：正如我们刚才所说，即使使用非受限对，我们也必须以相同的次数消耗该对的两个元素。唯一有效的解决方案是在类型签名中表明，该对的第二个元素——我们将其丢弃——不会以线性方式使用，方法是将其包装在一个 $!_{\omega}$ 模态中：

$\mathsf{fst}_{\mathsf{m}} ::: \mathsf{T} \otimes (!_{\omega} \mathsf{U}) \multimap_{1} \mathsf{T}$ $\mathsf{fst}_{\mathsf{m}} \triangleq$ 分情况 $\boldsymbol{\lambda}$
$x \mapsto_{1} x$ of $(x_1 , ux_2) \mapsto$分情况ux_2 of
$\mathsf{Mod}_{\omega}\ x_2 \mapsto x_1$

在实际编程场景中，线性性并非总是首要关注点，因此不得不在类型中到处插入 $!_{\omega}$ 模态，以及在项中到处插入分情况 t `of` $\mathsf{Mod}_{\mathsf{m}}\ x \mapsto \ldots$，会相当令人厌烦。理想情况下，我们希望在通常的 λ演算中有效的程序，几乎无需修改，尤其是语法方面，就能在我们的线性 λ演算中得到有效解释。

我们可以通过深层模态做到这一点。深层模态是实用模态语言中的常见特性，其做法是允许模态或相应的分级模态与其他数据类型（如和类型和积类型）交换顺序。例如，线性 Haskell ⟦Bernardy 等人、2018⟧, Granule ⟦Orchard、Liepelt 和 Eades III，2019a⟧, 以及较新的 Modal OCaml ⟦Lorenzen、White 等人，2024⟧都已实现这一特性。相关讨论还见于 ⟦Jack Hughes、Daniel Marshall 等，2021；Jack Hughes、Vollmer 和 Orchard，2021⟧。

借助深层模态，我们得到以下等价关系：

$!_{\mathsf{m}}(\mathsf{T} \otimes \mathsf{U}) \simeq (!_{\mathsf{m}}\mathsf{T}) \otimes (!_{\mathsf{m}}\mathsf{U})\ !_{\mathsf{m}}$
$(\mathsf{T} \oplus \mathsf{U}) \simeq (!_{\mathsf{m}}\mathsf{T}) \oplus (!_{\mathsf{m}}\mathsf{U})\ !_{\mathsf{m}}$
$(!_{\mathsf{n}}\mathsf{T}) \simeq !_{(\mathsf{m} \cdot \mathsf{n})}\mathsf{T}$

在我们加入深层模态扩展的模态线性 λ-演算（下文记作 λ_{Ldm}）中，类型为 $\mathsf{T} \multimap_{\omega} \mathsf{U}$ 的函数可以表示所有在简单类型 λ 演算中类型为 $\mathsf{T} \to \mathsf{U}$ 的函数。若我们在两个系统中为项和类型采用兼容的语法，那么将简单类型 λ 演算中有效的函数转换为 λ_{Ldm} 中有效的函数几乎无需改动；只需添加少量模态注释（同时加在项和类型中）即可。这与对吉拉德线性逻辑的忠实解释形成对比，正如我们刚才通过 λ_{Lm} 展示的那样：每当出现非线性控制流时（例如 $\mathsf{fst}_{\mathsf{m}}$），都需要显式引入（之后再消解）模态。

重新审视这个 `fst` 函数，借助 λ_{Ldm}中的深层模态，我们能够写出：

$\mathsf{fst}_{\mathsf{dm}} ::: \mathsf{T} \otimes \mathsf{U} \multimap_{\omega} \mathsf{T}$
$\mathsf{fst}_{\mathsf{dm}} \triangleq$ 分情况$_{\omega}$ $\boldsymbol{\lambda} x \multimap_{\omega} x$ 的 $(x_1 , x_2) \mapsto x_1$

这与$\mathsf{fst}_{\mathsf{nl}}$的实现完全相同，只是增加了模式注释！

$\boxed{\Gamma \vdash t : \mathsf{T}}$ （项的类型判断）

$$\frac{\Gamma_1 \vdash t : \mathsf{T}_1 \oplus \mathsf{T}_2 \quad \Gamma_2,\, x_1 :_{\mathsf{m}} \mathsf{T}_1 \vdash u_1 : \mathsf{U} \quad \Gamma_2,\, x_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash u_2 : \mathsf{U}}{\mathsf{m} \cdot \Gamma_1 + \Gamma_2 \vdash \mathsf{case}_{\mathsf{m}}\ t\ \mathsf{of}\ \{\mathsf{Inl}\, x_1 \mapsto u_1\ ;\ \mathsf{Inr}\, x_2 \mapsto u_2\} : \mathsf{U}}\ \oplus\mathrm{E}$$

$$\frac{\Gamma_1 \vdash t : \mathsf{T}_1 \otimes \mathsf{T}_2 \quad \Gamma_2,\, x_1 :_{\mathsf{m}} \mathsf{T}_1,\, x_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash u : \mathsf{U}}{\mathsf{m} \cdot \Gamma_1 + \Gamma_2 \vdash \mathsf{case}_{\mathsf{m}}\ t\ \mathsf{of}\ (x_1\ ,\ x_2) \mapsto u : \mathsf{U}}\ \otimes\mathrm{E} \qquad \frac{\Gamma_1 \vdash t : !_{\mathsf{n}} \mathsf{T} \quad \Gamma_2,\, x :_{\mathsf{m}\cdot\mathsf{n}} \mathsf{T} \vdash u : \mathsf{U}}{\mathsf{m} \cdot \Gamma_1 + \Gamma_2 \vdash \mathsf{case}_{\mathsf{m}}\ t\ \mathsf{of}\ \mathsf{Mod}_{\mathsf{n}}\ x \mapsto u : \mathsf{U}}\ !\mathrm{E}$$

图 1.8：模态表示中线性 λ-演算的深层模态类型规则（λ_{Ldm}–ty）

深层模态下语法和类型规则的形式上的变更 在 λ_{Lm} 和 λ_{Ldm} 之间，语法所需的唯一变更是：λ_{Ldm} 中的分情况 构造采用一个m 模式，用于指定它们以何种模式消耗判别式；该模式会传播到分情况主体中判别式各字段的变量绑定上。分情况 的新类型规则见图 1.8；带深层模态的线性 λ演算 λ_{Ldm}的其他所有规则均与图 1.7 中的规则相同。

对于这些新的 case 规则，我们注意到，在这些规则的结论中，判别式类型所在的类型上下文 Γ_1 会乘以 m。这与应用规则 λ_{Lm}–ty/$\multimap$E 非常相似：它确保如果我们要在模式 m 下使用 t，那么对 t 进行类型检查所需的资源就会被消耗 m 次。给定 $x_2 :_1 T_2 \vdash (() , x_2) ::: 1 \otimes T_2$（对恰好使用 x_2 一次），如果我们想以模式 ω 提取对的分量并丢弃 x_2（如 $\mathrm{fst}_{\mathsf{dm}}$ 中那样），那么 case$_\omega$ (() , x_2) of (x_1 , x_2) $\mapsto x_2$ 将需要上下文 $\omega \cdot (x_2 :_1 T_2)$，也就是 $x_2 :_\omega T_2$。换句话说，我们不能以无限制的方式使用依赖于线性变量的结构的部分（否则会破坏线性性）。

带有深层模态的线性 λ-演算 λ_{Ldm} 将作为我们下一章核心贡献——目的地演算 λ_d——的基础。

1.6 带有深层模态的线性 λ-演算的操作语义

关于 λ-演算，已有许多种语义表述。在图 1.9 中，我们为 λ_{Ldm}给出一个小步归约系统，采用归约语义风格，其灵感源自 [Felleisen、 1987] 以及后续的 [Biernacka 和 Danvy（2007）；Danvy 和 Nielsen， 2004], 也就是说，这是一种将求值上下文 E 显式地以语法形式作为栈来操作的语义。我们将求值上下文 E（即一个栈）定义为以下两者之一：空栈 $[]$, ，或由 $E' \circ e$ 组成的组合，其中 e——位于栈顶——表示最内层的 求值上下文分量（即缺少一个子项才能完整的项）。

我们用一个对 $E[t]$ 表示一个运行中的程序，其中包括求值上下文 E和处于焦点下的项 t。我们将这样的对 $E[t]$ 称为命令，借用了 Curien 和 Herbelin 的术语 [2000]。

小步求值规则分为三类：

· 聚焦规则（F）将焦点下的当前项拆分为两部分：一个被压入栈中以供稍后使用的求值上下文分量E， 以及一个置于焦点下的子项；

· 去聚焦规则（U）： 当焦点项成为值时，通过从栈中弹出最近的求值分量，并将其与该值合并，重新构造出一个更大的项；

$E\,[t\,] \longrightarrow E'\,[t'\,]$ (小步求值 ion)

$E\,[\,Inl\ t\,] \longrightarrow (E \circ Inl\ [\,])\,[\,t\,]$ $\star\ \oplus I_1$

$F(E \circ Inl\ [\,])\,[\,v\,] \longrightarrow E\,[\,Inl\ v\,]$ $\oplus$

$I_1 U\ E\,[\,Inr\ t\,] \longrightarrow (E \circ Inr\ [\,])\,[\,t$

$]$ $\star\ \oplus I_2 F(E \circ Inr\ [\,])\,[\,v\,] \longrightarrow E\,[\,Inr$

$v\,]$ $\oplus I_2 U\ E\,[\,(t_1, t_2)\,] \longrightarrow (E \circ ([\,],$

$t_2))\,[\,t_1\,]$ $\star\ \otimes IF_1(E \circ ([\,], t_2))\,[\,v_1\,] \longrightarrow E\,[\,($

$v_1, t_2)\,]$ $\otimes IU_1 E\,[\,(v_1, t_2)\,] \longrightarrow (E \circ (v_1, [\,])$

$)\,[\,t_2\,]$ $\star\ \otimes IF_2(E \circ (v_1, [\,]))\,[\,v_2\,] \longrightarrow E\,[\,(v_1,$

$v_2)\,]$ $\otimes IU_2 E\,[\,Mod_n\ t\,] \longrightarrow (E \circ Mod$

$_n\,[\,])\,[\,t\,]$ $\star\ !IF(E \circ Mod_n\,[\,])\,[\,v\,] \longrightarrow E\,[$

$Mod_n\ v\,]\ !IU\ E\,[\,t'\,t\,] \longrightarrow (E \circ t'\,[\,])\,[\,t$

$]$ $\star\ \multimap EF_1(E \circ t'\,[\,])\,[\,v\,] \longrightarrow E\,[\,t'$

$v\,]$ $\multimap EU_1 E\,[\,t'\,v\,] \longrightarrow (E \circ [\,]$

$v)\,[\,t'\,]$ $\star\ \multimap EF_2(E \circ [\,]\,v)\,[\,v'\,] \longrightarrow$

$E\,[\,v'\,v\,]$ $\multimap EU_2 E\,[\,(\lambda x\ m \mapsto u)\ v$

$] \longrightarrow E\,[\,u[x := v]\,]$ $\multimap EC\ E\,[\,t\ ;\ u$

$] \longrightarrow (E \circ [\,]\ ;\ u)\,[\,t\,]$ $\star\ 1EF(E \circ [\,]\ ;$

$u)\,[\,v\,] \longrightarrow E\,[\,v\ ;\ u\,]$ $1EU\ E\,[\,()\ ;$

$u\,] \longrightarrow E\,[\,u\,]$ $1EC\ E\,[\,$分情

况$_m\ t\ of\ \{Inl\ x_1 \mapsto u_1, Inr\ x_2 \mapsto u_2\}\,] \longrightarrow (E \circ$分情况$_m\,[\,]\ of\ \{Inl\ x_1 \mapsto u_1, Inr\ x_2 \mapsto u_2\})\,[\,t\,]$ $\star\ \oplus$

$EF(E \circ$分情况$_m\,[\,]\ of\ \{Inl\ x_1 \mapsto u_1, Inr\ x_2 \mapsto u_2\})\,[\,v\,] \longrightarrow E\,[\,$分情况$_m\ v\ of\ \{Inl\ x_1 \mapsto u_1, Inr\ x_2 \mapsto u$

$_2\}\,]$ $\oplus EU\ E\,[\,$分情况$_m\ (Inl\ v_1)\ of\ \{Inl\ x_1 \mapsto u_1, Inr\ x_2 \mapsto u_2\}\,] \longrightarrow E\,[\,u_1[x_1 := v$

$_1]\,]$ $\oplus EC_1 E\,[\,$分情况$_m\ (Inr\ v_2)\ of\ \{Inl\ x_1 \mapsto u_1, Inr\ x_2 \mapsto u_2\}\,] \longrightarrow E\,[\,u_2[$

$x_2 := v_2]\,]$ $\oplus EC_2 E\,[\,$分情况$_m\ t\ of\ (x_1, x_2) \mapsto u\,] \longrightarrow (E \circ$分情况$_m\,[\,]\ of\ ($

$x_1, x_2) \mapsto u)\,[\,t\,]$ $\star\ \otimes EF(E \circ$分情况$_m\,[\,]\ of\ (x_1, x_2) \mapsto u)\,[\,v\,] \longrightarrow E\,[\,$分

情况$_m\ v\ of\ (x_1, x_2) \mapsto u\,]$ $\otimes EU\ E\,[\,$分情况$_m\ (v_1, v_2)\ of\ (x_1, x_2) \mapsto u\,] \longrightarrow$

$E\,[\,u[x_1 := v_1][x_2 := v_2]\,]$ $\otimes EC\ E\,[\,$分情况$_m\ t\ of\ Mod_n\ x \mapsto u\,] \longrightarrow ($

$E \circ$分情况$_m\,[\,]\ of\ Mod_n\ x \mapsto u)\,[\,t\,]$ $\star\ !EF(E \circ$分情况$_m\,[\,]\ of\ Mod_n\ x \mapsto$

$u)\,[\,v\,] \longrightarrow E\,[\,$分情况$_m\ v\ of\ Mod_n\ x \mapsto u\,]\ !EU\ E\,[\,$分情况$_m\ Mod_n\ v\ of\ Mod_n\ x \mapsto u\,] \longrightarrow E\,[\,u[x := v]\,]\ !$

$EC\star$: 仅当将成为新焦点的项尚不是值时才允许

图 1.9: λ_{Ldm} (λ_{Ldm}–语义)的小步语义

· 不作用于栈的收缩规则（C）E 但当焦点项是可约表达式时，只转换焦点项（沿用以下文献引入的术语：⟦Biernacka 和 Danvy，2007年；Danvy 和 Nielsen， 2004⟧）。

为了得到一个完全确定且直观的归约系统，只有当将要聚焦的子项还不是值时，聚焦规则才会触发（如图 1.9 中的 $\star$ 所示）。

数据构造器只有聚焦规则和去聚焦规则，因为它们并不描述可以进行归约的计算。就此而言，$\mathsf{Mod}_m t$ 被视为一元数据构造器。一旦数据构造器获得焦点，就会被完全求值为值形式。

在大多数情况下，去聚焦规则和收缩规则可以合并为一个步骤。例如，一旦得到 $\big(\mathrm{E}\ \circ$ 分情况$_m\ []$ of $(\mathrm{x}_1, \mathrm{x}_2) \mapsto \mathrm{u}\big)\,[(\mathrm{v}_1, \mathrm{v}_2)]$，就可以立即触发收缩，而无需重新构造项 $\mathsf{case}_m\ (\mathrm{v}_1, \mathrm{v}_2)$ of $(\mathrm{x}_1, \mathrm{x}_2) \mapsto \mathrm{u}$。不过，我们更倾向于采用三个彼此独立的步骤进行展示；尽管这种方式更加冗长，却能清楚地表明收缩规则不会修改求值上下文。

系统的其余部分都非常标准。特别是，λ_{Ldm} 的收缩规则——它们体现了该演算的本质——与带有和类型、积类型（以及严格求值策略）的 λ演算中的收缩规则非常相似。

例如，让我们看看函数应用的归约。我们沿用之前那个类型签名为 $!\mathsf{T} \multimap (\mathsf{T} \otimes \mathsf{T})$ 的函数示例，此处将它应用于参数 $\mathsf{Mod}_\omega\,(\mathsf{Inl}\,())$：

$$
\begin{array}{lll}
& []\big[(\boldsymbol{\lambda} x_{\,1} \mapsto \mathsf{case}\ x\ \mathsf{of}\ \mathsf{Mod}_\omega\ x' \mapsto (x', x'))\ (\mathsf{Mod}_\omega\ (\mathsf{Inl}\,()))\big] & \\
\longrightarrow & []\big[(\mathsf{case}\ \mathsf{Mod}_\omega\ (\mathsf{Inl}\,())\ \mathsf{of}\ \mathsf{Mod}_\omega\ x' \mapsto (x', x'))\big] & \multimap\mathrm{EC} \quad \text{with substitution } x := \mathsf{Mod}_\omega\ (\mathsf{Inl}\,()) \\
\longrightarrow & []\big[(\mathsf{Inl}\,(), \mathsf{Inl}\,())\big] & !\mathrm{EC} \quad \text{with substitution } x' := \mathsf{Inl}\,()
\end{array}
$$

由于我们已有一个值形式的函数，其函数体很简单，并且它应用于一个值，因此无需聚焦到某个子项中，再在之后取消聚焦；我们只会看到一系列收缩步骤。

形式为 $[]\big[v\big]$ with v 的命令中，若 t 是 $[]\big[t\big]$，且 E 是 $E\big[v\big]$，那么对于所有类型良好的程序而言，以值为 v 的命令都是归约的终点。若我们有一个非值项，就会触发聚焦或收缩步骤；若我们有一个非空的求值上下文，就会触发取消聚焦步骤。只有格式错误的项或值才会导致归约停滞，这是程序类型错误的表现。

1.7 Haskell 中的线性类型

作为这项博士研究的一部分，我们计划在一种工业级函数式编程语言中实现原型，以探索目标传递领域。得益于对线性类型的支持，Haskell 是合适的目标语言。接下来，我们将向读者简要介绍线性 Haskell，以及前文讨论的概念如何映射到 Haskell 世界中。

线性 Haskell ⟦Bernardy 等人， 2018⟧ 是格拉斯哥 Haskell 编译器（GHC）的一种语言扩展，它引入了线性函数箭头 `t ⊸ u`，并修改 GHC 的类型检查器，使其能够强制执行线性性要求。类型为 `t ⊸ u` 的线性函数保证：当函数的结果恰好被消耗一次时，其参数也恰好被消耗一次。启用线性 Haskell 后，仍可使用常规函数箭头 `t → u`；但当其结果被消耗一次时，它并不保证参数会被消耗多少次。实际上，线性 Haskell 基于一种模态形式主义，例如 λ_{Ldm}，因此也有重数多态箭头 `t %m → u`，它与 $\mathsf{T}\ {}_m\!\!\multimap \mathsf{U}$非常相似；因此，`t ⊸ u` 和 `t → u` 分别是 `t %1 → u` 和 `t %ω → u` 的语法糖。

当一个值被模式匹配，且其子分量各自恰好被消耗一次时，或者当它作为参数传递给一个结果恰好被消耗一次的线性函数时，就称该值被消耗一次（或线性消耗）。当一个函数应用于某个参数，且其结果恰好被消耗一次时，就称该函数被消耗一次。如果消耗表达式 `u` 一次意味着变量 `x` 恰好被消耗一次，我们就说变量 `x` 在表达式 `u` 中被线性使用。反之，没有被线性使用的值称为非线性使用，或以无限制的方式使用。

非受限值 线性 Haskell 引入了一个名为 `Ur` 的包装器，用于表示线性上下文中的值不必以线性方式使用。`Ur t` 是 Haskell 中对应于 $\lambda_{L1,2}$ 中的$!T$ 或 λ_{Ldm}中的$!_{\omega}T$ 的形式，并且 `Ur t ⊸ u` 与`t → u` 等价。与 λ_{L2} 和 λ_{Ldm}中一样，我们可以使用 `case term of Ur x ⊸ term'` 对类型为`Ur t` 的项进行模式匹配，从而得到一个类型为`t`的非受限变量绑定`x` 。

与线性类型系统中的常见情况一样，只有已经包装在`Ur` 中的值，或来自非线性箭头左侧的值，才能被放入另一个`Ur` 中而不破坏线性性。这与规则 λ_{L1}–ty/!P 或 $\lambda_{L2,m}$–ty/!I 相呼应：被 `Many` 或`Mod`$_{\omega}$ 包装的项只能依赖于非受限上下文。此规则唯一的例外是实现`Movable` 类型类[6] 的类型的项，例如`Int` 或`()`。`Movable` 提供函数`move ::t ⊸ Ur t`，即对于实现此类型类的类型，可以无条件地提升为`Ur` 。

运算符 本文其余部分经常会用到一些 Haskell 运算符和记法：

`(⨾) :: () ⊸ u ⊸ u` 用于串联一个返回 `()` 的线性操作和一个返回 `u` 类型值的操作，同时不破坏线性性。它相当于 Haskell 中的 λ_{Ldm}–TY/1E。它并非 Haskell 核心语法的一部分，但只需用以下分情况表达式即可定义： `分情况 term1 of () -> term2`。

`类 ⇒ ...` 是类型类约束的记法（由编译器隐式解析）。

`@t` 在 `f @t ...`中表示类型应用；它允许为多态函数指定显式类型。

在代码摘录中，类型以蓝色显示，而项、变量和数据构造器（有时其名称与所属类型相同，例如 `Ur` （类型） / `Ur` （数据构造器））以黑色显示，以尽可能与前几章保持一致。遗憾的是，尽管在我们的形式体系中类型变量 `T`,`U` 使用大写字母，但在 Haskell 中我们只能使用小写字母 `t`、`u`。

1.8 唯一性与线性作用域

在引言中，我们曾说过，将使用线性类型系统来控制资源（尤其是目标）的使用次数。函数式编程中，资源管理更常见的做法是将资源封装在单子中。虽然单子可以确保受控访问，但它们会造成传染：一旦某项计算接触到单子值，外围代码通常就不得不采用相同的单子结构，即使它并未直接操作该资源也是如此。这往往会模糊纯代码与有效果代码之间的界限，使程序更难阅读、组合和重构。相比之下，线性类型系统承诺能够精确控制资源使用，同时避免无处不在的开销，因为程序可以保持“直接风格”。

然而，线性类型系统乍看之下似乎无法充分胜任这项工作。在一个线性类型系统中，我们只能承诺自己会如何处理收到的值。对于此前已经对它做过什么，我们无法多说。因此，我们无法确保它在整体上恰好被使用一次。

[6] The`Movable` 类型类并非线性 Haskell 语言扩展的一部分，而是在`linear-base` 库中定义的。

例如，给定一个类型为`t ⊸ u`的函数，如果我们将一个类型为`t`的参数传给它，就知道不会复制这个类型为`t` 的值（只要我们恰好使用一次类型为`u` 的结果）。但我们无法判断这个`t` 此前是否已被复制：这是个重大问题。

陈述有关某个值过去 的信息，属于唯一性类型系统 或唯一类型系统的范畴。在唯一类型系统中，核心属性并非函数使用某个值多少次来生成输出，而是该值是否存在别名。唯一性（或非唯一性）是值的属性，而线性性是函数的属性[7]。值具有唯一性或非唯一性属性，可以更轻松地控制那些在其生命周期内应始终保持唯一的资源。最常见的例子是高效实现函数式数组。如果数组以唯一方式处理且未被共享，那么我们就可以像 ⟦Danielle Marshall、Vollmer 和 Orchard， 2022⟧ 中那样，通过原地内存修改来实现数组更新。借助类型系统静态地声明值的唯一性，我们便能实现可预测的编译器优化。

然而，唯一性类型系统的理论基础并不像线性类型那样丰富。与线性性不同，唯一性属性最初是以非形式化的方式加以研究的，之后才从逻辑层面进行研究，见 ⟦Harrington， 2006⟧。此外，对于我们之后要做的事情，唯一性类型系统还有两个缺点。首先，在唯一类型系统中，强制使用某个资源并不容易（而我们确实需要一种方法，强制使用目标，而不只是将其丢弃）。使用唯一性类型系统，我们只能声明或证明该资源没有别名。其次，也许更重要的是，高阶函数（捕获 λ 抽象）在唯一类型系统中会带来许多边界情况，详见 ⟦Harrington 的第 5 节， 2006⟧； ⟦Danielle Marshall、Vollmer 和 Orchard， 2022⟧ 中也对此有所简述（这也是他们偏好以线性系统为基础、再添加唯一性扩展，而非以唯一性系统为基础、再添加线性性扩展的原因之一）。

事实上，Danielle Marshall、Vollmer 和 Orchard ⟦2022⟧认为，现代编程语言若能同时整合线性类型系统和唯一性类型系统，将会受益匪浅，因为线性性与唯一性似乎是半对偶、半互补的概念，二者结合起来足以对语言中的资源使用进行非常精确的控制。近来，Lorenzen、White 等人 ⟦2024⟧ 也以类似的思路，在其模态系统中纳入了仿射性 （资源至多使用一次）和唯一性，并研究了二者之间的相互作用。

然而，在第3章中，我们的主要目标语言将是 Haskell。正如第1.7节所详述的，Haskell 没有唯一性类型系统，只有线性类型。幸运的是，我们可以在支持线性类型的语言中，借助线性作用域 或作用域函数技巧（姑且这样称呼它）来模拟唯一性控制。

线性作用域 如前所述，在线性 Haskell 中，与大多数线性类型系统一样，可以规定函数是否必须线性地使用其参数。换句话说，线性性是一种约定，用来关联函数的输入与输出。但我们无法预先断定，传入函数的值在函数调用之前经历了什么，也无法断定函数结果之后会发生什么。线性控制只能施加于变量[8]。

假设我们想要设计一个不使用单子的文件 API（这实际上是线性类型／唯一类型的一个有趣用途）。

[7]至少在上述介绍中，这一点是由吉拉德线性逻辑推导出的。也存在一些更侧重编程方面的线性类型系统方法，在这些方法中，线性性是值及其类型的属性，而不是函数的属性。[8]顶层绑定／定义不被视为普通变量，而是全局常量。因此，它们不受线性约束的限制，这使得我们接下来要介绍的问题示例成为可能。

假设采用以下简化的 API，我们可以编写这个小程序：

```
openFile:: 字符串 ⊸ 文件 closeFile:: 文件 ⊸ () 文件 ::文件 -- 无法强制此值恰好使用一次
文件 = openFile "test.txt" nonLinearUseButValid:: () =
closeFile file ; closeFile file -- 即使 file 被消耗两次也有效
```

按值调用或按需调用语义下，openFile 产生的副作用只会发生一次，而closeFile产生的副作用会发生两次，从而导致错误。

解决方法是禁止使用者创建并直接访问我们希望对其施加线性性/唯一性约束的资源类型的值。

相反，我们强制资源的使用者传入一个延续，表示他们希望对该资源执行什么操作，这样我们就可以通过其类型签名检查它确实是一个线性延续：

```
withFile:: 字符串 ⊸ (文件 ⊸ t) ⊸ t closeFile:: 文件 ⊸ ()
nonLinearUseAndEffectivelyRejected:: () = withFile "test.txt" (\文件 →
closeFile 文件 ; closeFile 文件) -- 非线性
```

类型 文件 在 withFile 的类型签名中处于正位置，因此函数 withFile 应该以某种方式知道如何生成一个 文件，但这对用户来说是不透明的。重要的是，只有向 withFile 提供线性延续，才能访问文件。

不过，这还不够；由于 \文件 → 文件 确实是线性延续，因此可以使用withFile "test.txt" (\文件 → 文件)来泄露一个文件，然后在外部世界中以非线性方式使用它。因此，我们必须禁止文件 出现在延续的返回类型中的任何位置。

为此，我们要求返回类型由非限制模态 Ur 包裹（这相当于线性 Haskell 中的 !模态）：由于类型为 File 的值来自线性箭头的左侧，因此无法将其提升为 Ur，否则就会破坏线性性，所以它无法离开作用域，并且必须在此作用域中恰好使用一次。这就是我们所说的线性作用域。下面是更新后的示例：

```
withFile' :: String ⊸ (File ⊸ Ur t) ⊸ Ur t 2 closeFile :: File ⊸ () 3
exampleOk :: Ur () = withFile' "test.txt" (\file → closeFile file ; Ur ()) 4
exampleFail:: Ur File =5 withFile' "test.txt" (\file → Ur file) -- 泄漏文件，因此实际上会被拒绝
```

不过，这个解决方案并不完美。实际上，我们阻止了任何线性资源离开作用域，即使这些资源与文件 API 毫无关系，且是在外层作用域中创建的！Spiwack 〚2023b〛对这个问题进行了详细讨论。

这个问题体现在下面这个被拒绝的示例中：

```
1 readLine :: File ⊸ (Ur 字符串, File) 2
writeLine File ⊸ 字符串 ⊸ File 3 4
exampleFail' ::Ur () =5 withFile' "test.txt" (\file
1 →6 … 7 withFile' "test2.txt" (\file2 →8 let 表
达式 (Ur line, file1') = 读取行 file1 9 file2' = 写
入行 file2 line 10 in closeFile file2' ⨟ 1
Ur file ') 11 … -- 此处对 file1' 的其他操作 12 )
```

这里的问题在于，我们无法从内部作用域返回 `file1'`，因为它是无法提升为 `Ur` 的线性资源。不过，这样做实际上完全安全；只是我们在类型层没有足够精确的工具来让编译器确信这一事实。在这种情况下，最明显的变通方法是，只要我们还需要对 `file1'` 进行操作，就扩展 `withFile' "test2.txt" (file2 → …)` 的作用域。确实非常遗憾，也很不实用：我们仍然需要对外部作用域中的资源进行操作，却不得不延长内部作用域！（参见 ⟦Spiwack, 2023b⟧ 中关于线性作用域的黏滞结尾 的讨论）

一种简单的改进方法是使用单个线性作用域，提供一种线性令牌类型，并让这种令牌类型用于实例化任何我们想要强制满足线性性／唯一性的资源。这样一来，我们只需使用一个大的线性作用域，就能以直接风格编写大部分代码，而且仍然不需要单子！

```
1 类型 Token 2 dup ::Token ⊸ (Token, Token) 3 丢弃 :: Token ⊸ () 4 withToken :: (Token ⊸
Ur t) ⊸ Ur t 5 6 -- 第一个线性 API 7 openFile' :: 字符串 ⊸ Token ⊸ 文件 8 closeFile :: 文件 ⊸ () 9
10 -- 第二个线性 API 11 openDB :: 字符串 ⊸ 字符串 ⊸ Token ⊸ 数据库 12 closeDB :: 数据库 ⊸ ()
13 14 exampleOk1 :: Ur () =15 withToken (\tok → let 表达式 file = openFile' "test.txt" tok in
closeFile file ⨟ Ur ()) 16 exampleOk2 :: Ur () =17 withToken (\tok → let 表达式 (tok1, tok2) = dup
tok 18 file = openFile' "test.txt" tok1 19 db = openDB "localhost" 3306 tok2 20 in closeFile file ⨟
closeDB db ⨟ Ur ()) 21 exampleRejected :: Ur File =22 withToken (\tok → Ur (openFile' "test.txt"
tok)) -- 泄漏了一个在线性上依赖于 tok 的文件
```

k

这里可能令人惊讶的是，我们可以复制或丢弃令牌（使用 dup 或 drop），尽管我们之前宣称令牌是一种线性资源！

事实上，Token 类型的目的是将每种资源都与一项线性要求绑定起来。关键在于，Token 类型不应有途径被提升为 Ur，否则令牌和其他线性资源就可能泄漏，并存在于 withToken的线性作用域之外。

线性 dup 函数会将一个令牌变成两个，但仍通过线性箭头将生成的令牌与输入令牌关联起来。允许复制令牌，并不意味着使用者就能复制我们想要控制的资源（文件、数据库句柄等）。因此，在这方面我们是安全的。

采用这种方法，处理多种线性资源类型会统一得多。我们可以使用任何剩余的令牌来实例化一个或多个新的线性资源。由于我们只有一个包含所有线性资源的大型线性作用域，因此无需担心必须扩展嵌套作用域的问题。当程序被拆分成许多子函数时，这些函数甚至不必处理或了解线性作用域；它们只需传递和接收令牌，其他部分则可以采用直接风格编写。

虽然这超出了本文档的范围，但 Spiwack 等人 ⟦2022⟧ 进一步探讨了线性令牌和作用域的概念，并提出了线性约束。它大致是一种以隐式方式传递令牌的方法，从而让线性类型的编程更加便利[9]。

在接下来的章节中，我们将再次采用带令牌的线性作用域 策略，对线性资源实施线性性和唯一性约束，从而在纯函数式的直接风格 API 中安全地使用副作用。

1.9 结论

本章介绍了线性 λ-演算，从其在线性逻辑中的根源，到更注重实践和计算的表述。特别是，我们为模态线性 λ-演算奠定了理论基础，后续章节将再次用到它。

此外，我们还探讨了线性 Haskell，将其作为一种用于试验线性类型的具体目标。特别是，我们展示了线性类型如何帮助管理唯一资源：这些资源必须使用，但又不能复制；不过，仍需稍加谨慎，以避免线性类型固有的局限（也就是说，它们本身无法保证过去对某个值做过什么）。

现在我们已进入熟悉的领域，具备了探索函数式目标传递理论和实践层面所需的全部工具。

[9]撰写本文档时，GHC 正在认真考虑将线性约束纳入其中。GHC 是主要的 Haskell 编译器 ⟦Spiwack, 2023a⟧。

第2章

基于一等目的地的形式函数式语言：λ_d

在上一章中，我们介绍了构建此处所开发的函数式语言所需的重要基础构件，即 λ_d。接下来，我们来看看为什么目的地传递在纯函数式编程环境中很有价值，以及 λ_d 为何能为此提供良好的基础层。

本章介绍的成果也已发表于 OOPSLA 2025 会议 [Bagrel 和 Spiwack, 2025a]。

2.1 函数式环境中的目的地传递

在目标传递风格中，函数不返回值，而是接收一个位置作为参数——该位置称为目标——函数的输出应写入其中。在本章中，我们用 $\lfloor \mathtt{T} \rfloor$ 表示目的地，其中 $\mathtt{T}$ 是目的地中可存储内容的类型。类型为 $\mathtt{T} \to \mathtt{U}$ 的函数，转换为目标传递风格后，其类型签名为 $\mathtt{T} \to \lfloor \mathtt{U} \rfloor \to \mathtt{1}$。它不再返回 $\mathtt{U}$，而是接收一个类型为 $\lfloor \mathtt{U} \rfloor$ 的目的地，并仅返回单位（类型 $\mathtt{1}$）。这种风格在系统编程中很常见，其中目的地 $\lfloor \mathtt{U} \rfloor$ 通常称为输出参数（在 C语言中，$\lfloor \mathtt{U} \rfloor$ 通常是一个类型为$\mathtt{U}*$的指针）。

系统程序之所以如此依赖目标，是因为在每一点性能提升都弥足珍贵的情况下，使用目标可以减少对内存分配器的调用，而这些调用代价高昂。如果函数返回一个 $\mathtt{U}$，就必须为 $\mathtt{U}$ 分配空间。但使用目标时，由调用方负责为 $\mathtt{U}$ 找到空间。调用方可以直接向内存分配器申请空间，这样一来我们并没有节省任何东西；但也可以重用现有 $\mathtt{U}$ 所占用的空间（如果已经不再需要它），或数组中的空间，甚至是分配器无法访问的内存区域中的空间，例如内存映射文件。事实上，正如我们将在第 2.5 节中看到的，目标传递风格总是比对应的直接风格更通用（尽管可能稍显冗长）；使用它没有任何缺点，因为我们总能以系统化的方式，从目标传递风格的函数中还原出常规的直接风格函数。

到目前为止，我们主要考察了命令式上下文中的目的地传递，例如系统编程，但我们认为，目的地传递也能为函数式编程语言带来诸多益处，即使是纯函数式编程语言也不例外。目的地在函数式编程中真正发挥优势之处，在于它们让我们能够借鉴

类命令式编程技术，从而打造表达能力更强、对处理过程有更多控制的语言；只要谨慎处理，我们仍然可以保留函数式语言的优良特性，例如纯性和（表面上的）不可变性。

因此，这里的目标是对标准函数式编程语言进行适度修改，使其能够通过目的地传递构建不可变结构，同时不损害纯性和内存安全性。在这种特定设定中，目的地变成了指向含有空洞、尚不可读取的结构的只能写入一次的引用。这种做法比 C 指针（可不受限制地用于读写）严格得多，但仍足以将新的编程技术引入函数式编程领域。

更准确地说，有两个关键元素提升了目的地传递在函数式上下文中的表达能力：

- 结构可以按任意顺序构建。不仅可以像普通函数式编程那样从叶节点到根节点构建，也可以从根节点到叶节点构建，或以这两种方式的任意组合进行构建。在普通函数式编程中，可以通过函数复合并采用延续传递的形式来实现这一点；不过，目的地是对这种方案的一种非常直接的优化。这一研究方向由 Minamide ⟦1998⟧开创；

- 当目的地是一等值时，它们可以像普通值一样被传递和存储。其结果不仅是结构的构建顺序可以任意，还意味着该顺序可以在程序运行时动态确定。与早期工作相比，这正是我们方法的主要创新之处。

在本章中，我们介绍 λ_d，这是一种纯函数式演算，其核心正是目的地这一概念。我们希望 λ_d 将其作为一种基础理论演算，用于研究函数式环境中的安全目的地；因此，本章将仅非常简略地讨论实现问题，因为 λ_d 并非真正意图作为一种实际的编程语言来实现。在现有函数式语言中实际实现目的地传递，将是后续章节的重点。

λ_d 涵盖了文献中所有带有目的地的函数式系统，从 ⟦Minamide， 1998⟧ 到 ⟦Lorenzen、Leijen、Swierstra 和 Lindley， 2024⟧。因此，我们预计，只需将这些系统或其扩展翻译为 λ_d，便可以为它们提供正当性，从而免费获得 λ_d 的全部安全性结果和元理论。事实上，我们使用 Coq证明助手证明了 λ_d 的类型安全定理，具体描述见第 2.7 节。

2.2 在 λ_d中处理目标

让我们来介绍并熟悉 λ_d，这是一种带有目的地的模态、线性、简单类型化 λ演算。我们借用了上一章的大部分语法，尤其是 λ_{Ldm}中的语法。我们仍然使用线性逻辑中的 `T⊕U` 和 `T⊗U` 来表示和类型与积类型，并使用线性函数箭头 $\multimap$，因为 λ_d 是线性类型化的。与之前的演算相比，主要区别在于 λ_d 没有一等数据构造器，例如 $\mathtt{Inl}\,t$ 或 (t_1 , t_2)，因为它们被更通用的目的地填充算子所取代，我们将在接下来的段落中介绍这些算子。

2.2.1 构建词汇

最简单的目的地传递形式与续延传递很相似：将作为参数接收的位置用于写入输出值，而不是返回该值。例如，下面是恒等函数的目的地传递版本：

```
dId : T ⊸ ⌊T⌋ ⊸ 1
dId x d ≜ d ◀ x
```

我们可以将目标视为对未初始化内存位置的引用，而将 $d \blacktriangleleft x$ （读作“用 x 填充 d”）视为将 x 写入 d所指向的内存位置。

执行 $d \blacktriangleleft x$ 是使用目标的最简单方式：这样我们可以直接用一个值填充它。但我们也可以通过仅指定想要用于填充它的最外层构造器，来分段填充目标：

```
用 Inl 填充 :::⌊T⊕U⌋ ⊸ ⌊T⌋
fillWithInl d ≜ d ◁ Inl
```

在此示例中，我们通过将最外层构造器设置为左变体 Inl，为类型 T⊕U 填充一个目标。我们将 $d \triangleleft$ Inl （读作“用 Inl 填充 d ”）理解为：分配内存以存储一个形如 Inl 的块，并将该块的地址写入 d 所指向的位置。由于我们刚刚创建了一个尚无参数的 Inl 构造器，因此会返回一个类型为 ⌊T⌋ 的新目标，它指向 Inl 的未初始化参数。当未初始化内存作为结构或值的一部分（例如 Inl）存在时，我们称其为一个空洞。

请注意，借助 `fillWithInl`，我们是从最外层构造器向内构建结构的：我们已将一个形如 Inl 的值写入空洞，但还没有描述新空洞中应放入什么。这类带有未初始化参数的数据构造器称为空心构造器[10]。这与函数式编程通常的工作方式相反；在函数式编程中，值是从最内层构造器向外构建的：我们先构建一个值 v，然后才能使用 Inl 构造出 Inl$v$。能够从最外层构造器向内构建结构，是我们之前所承诺的额外表达能力的关键要素。

然而，原则上，我们迄今所展示的一切都可以用延续来表达。因此，值得探讨的是：目标与延续究竟有何不同？答案的一部分在于，我们打算将目标有效地实现为指向未初始化内存的指针（见第 2.9 节以及第 3 章和第 4 章）。但目标与延续真正不同之处在于，我们可以轻易地将一个数据结构的目标拆分为对应其各个部分的多个目标。接着，用户必须填充所有由此产生的目标。这种拆分的一个例子是`fillWithPair` 函数，它会将一个空心对构造器写入目标：

```
fillWithPair :::⌊T⊗U⌋ ⊸ ⌊T⌋⊗⌊
U⌋fillWithPair d ≜ d ◁(,)
```

使用 `fillWithPair` 后，必须分别使用返回的、类型为 ⌊T⌋ 和 ⌊U⌋ 的目标，填充对的第一个字段和第二个字段。

[10]完整三角形 ◀ 用于用一个完整构造的值填充目标，而空心三角形 ◁ 则用于用一个空心构造器填充目标。

相比之下，fillWithPair 的延续传递风格等价形式需要类型签名 (T⊗U ⊸ S) ⊸ (T ⊸ S)⊗(U ⊸ S)。但无法定义 fillWithPairCPS ::: (T⊗U ⊸ S) ⊸ (T ⊸S)⊗(U ⊸ S)，因为我们需要先收集两个已创建延续的结果，才能使用类型为 (T⊗U ⊸S) 的输入延续（因而需要某种形式的同步和状态传递）。

不过，与延续传递变换类似，我们已经可以注意到，目的地填充算子与相应的数据构造器之间存在某种对偶关系。常见的 Inl 构造器的类型签名为 T ⊸ T⊕U，而目的地填充 ◁Inl 的类型签名为 ⌊T⊕U⌋ ⊸ ⌊T⌋。类似地，对构造器 (,) 的类型签名为 T ⊸ U ⊸ T⊗U，而目的地填充 ◁(,) 的类型签名为 ⌊T⊗U⌋ ⊸ ⌊T⌋⊗⌊U⌋。假设柯里化方式有一定灵活性，我们可以看到，参数和结果的类型会在箭头两侧互换；从构造器变为目的地填充算子，反之亦然时，它们也会从 ⌊•⌋ 中解包或封装。这个观察结果可以推广到所有目的地填充算子及其对应的构造器。

带孔结构 需要特别注意的是，尽管目标用于构造一个结构，但目标仅指向该结构中尚未定义的某个特定部分，而不是整个结构。因此，正在构造的结构的（根）类型通常会不同于当前目标的类型。类型为 ⌊T⌋ 的目标只表示，某个更大的结构中至少存在一个类型为 T 的空洞。结构本身的类型从不出现在目的地填充函数的类型签名中（例如，使用 `fillWithPair` 只表示正在操作的结构中有一个类型为 T⊗U 的空洞，并且该空洞正被写入）。

因此，我们仍然需要一种类型，将正在构造的结构——由目的地填充原语隐式确定——与表示其空洞的目的地联系起来。为表示这一点，λ_d 引入了一种类型 S ⋉ ⌊T⌋，用于表示一个类型为 S、缺少一个类型为 T 的值因而尚未完成的结构。在 S 中可以存在多个空洞——从而在右侧产生多个目的地——只要 S 中仍有空洞，就无法读取它。例如，S ⋉(⌊T⌋⊗⌊U⌋) 表示一个 S，它同时缺少一个 T 和一个 U 才能完成（从而变得可读）。 ⋉ 的右侧并不局限于对和目标类型；它可以具有任意复杂的类型。

一般形式 S ⋉T 读作“S ampar T”。“ampar”一名代表“非对称记忆 par”。“par”运算符最初源自（经典）线性逻辑，是一种满足结合律和交换律的运算符，它涵盖线性蕴涵，表达能力也更强：T ⊸S 等价于 T$^\perp$ S 或 S T$^\perp$。类似地，T_1 ⊸ T_2 ⊸ S 等价于 T1$^\perp$ T2$^\perp$ S。函数输入的类型处于负位置，当函数被写成“par”形式时，会用对偶化算子 •$^\perp$ 包裹。这里我们采用类似的方法：Minamide ⟦1998⟧首先指出，带孔结构类似于线性函数T ⊸ S，其中 T 表示缺失的部分；因此，我们决定以更具表达力的形式 S ⋉ ⌊T⌋ 来表示它们（呼应 S T$^\perp$）。

在此呈现中，类型 T 缺失的部分以一等类型 ⌊T⌋ 的形式出现，也称为目标类型。沿用 Danielle Marshall 和 Orchard ⟦2022⟧, 的术语，我们可以说 ⌊T⌋ 是 T 的松弛逆，其对应 ⊗；因为给定 d ::: ⌊T⌋ 和 v ::: T，有 d ◀ v ::: 1[11]。

我们的记忆 par 具有非对称性，这是为了实现简单的顺序语义而产生的不幸副作用。事实上，似乎只有采用并发语义，才能实现对称的记忆 par，因为正在构造的结构和指向它的目的地都可能被同等地消耗，这意味着我们可能不得不暂停并等待空洞被填充[12]。

[11]反过来，⌊T⌋ 则是T 的余松弛逆，其对应 ⋉；因为 **λ**un ↦ un ⨟新建⋉ 的类型为 1 ⊸ T ⋉ ⌊T⌋[12]⟦DeYoung、Pfenning 和 Pruiksma， 2020⟧ 提出了这样一种具有目的地和并发执行模型的系统

目的地虽然是一等值，但始终存在于带孔结构的上下文中。目的地既是结构中存在空洞的见证，也是向其中写入内容的句柄。关键在于，目的地在其他方面仍是普通值，位于相应 ampar 的右侧。要访问 ampar 的目的地，λ_d 提供了一种 $\mathtt{upd}_{\ltimes}$ 构造，使我们能够将函数应用于 ampar 的右侧。正是在 $\mathtt{upd}_{\ltimes} ::: \mathtt{S} \ltimes \mathtt{T} \multimap (\mathtt{T} \multimap \mathtt{U}) \multimap \mathtt{S} \ltimes \mathtt{U}$ 的主体中，可以调用操作目的地的函数来更新结构：

$$\mathrm{fillWithInl}' ::: \mathrm{S} \ltimes \lfloor \mathrm{T} \oplus \mathrm{U} \rfloor \multimap \mathrm{S} \ltimes \lfloor \mathrm{T} \rfloor \mathrm{fillWithInl}'\ x \triangleq \mathrm{upd}_{\ltimes} x \text{ with } d \mapsto d \triangleleft \mathrm{Inl}\ \mathrm{fillWithPair}' ::: \mathrm{S} \ltimes \lfloor \mathrm{T} \otimes \mathrm{U} \rfloor \multimap \mathrm{S} \ltimes (\lfloor \mathrm{T} \rfloor \otimes \lfloor \mathrm{U} \rfloor)\ \mathrm{fillWithPair}'\ x \triangleq \mathrm{upd}_{\ltimes} x \text{ with } d \mapsto \mathrm{fillWithPair}\ d$$

总而言之，我们需要一种引入和消去带孔结构的方法。带孔结构通过 $\mathtt{新建}_{\ltimes}$ 引入，它会创建一个类型为 $\mathtt{T} \ltimes \lfloor \mathtt{T} \rfloor$ 的值。$\mathtt{新建}_{\ltimes}$ 与线性恒等函数有些相似：它是一个（类型为 $\mathtt{T}$ 的）空洞，需要一个类型为 $\mathtt{T}$ 的值才能成为类型为 $\mathtt{T}$ 的完整值。从内存角度看，它是一个足以容纳类型为 $\mathtt{T}$ 的值的未初始化块，以及一个指向该块的目标。相反，带孔结构通过[13] $\mathtt{from}'_{\ltimes}$ 消去：$\mathtt{S} \ltimes \mathtt{1} \multimap \mathtt{S}$：如果所有目标都已被使用，且右侧只剩下单位，那么 $\mathtt{S}$ 就不再有孔，因此只是一个普通的数据结构。

有了这些新运算符，我们终于可以展示如何通过逐步填充来导出传统构造器。事实上，正如前面所说，λ_d 没有原始构造形式；λ_d 中的构造器都是语法糖。这里我们给出 $\mathtt{Inl}$ 和 (,) 的定义，其他构造器也可以用类似方式导出。第二个示例中出现的运算符 $\fatsemi ::: \mathtt{1} \multimap \mathtt{U} \multimap \mathtt{U}$，用于串联返回单位类型 $\mathtt{1}$ 的操作。为便于阅读，我们还在使用 $\mathtt{新建}_{\ltimes}$ 时提供了类型注释：

$$\mathrm{Inl} ::: \mathrm{T} \multimap \mathrm{T} \oplus \mathrm{U}$$
$$\mathrm{Inl}\ x \triangleq \mathrm{from}'_{\ltimes}(\mathrm{upd}_{\ltimes}\ (\mathtt{新建}_{\ltimes} ::: (\mathrm{T} \oplus \mathrm{U}) \ltimes \lfloor \mathrm{T} \oplus \mathrm{U} \rfloor) \text{ with } d \mapsto d \triangleleft \mathrm{Inl} \blacktriangleleft x)$$
$$(,) ::: \mathrm{T} \multimap \mathrm{U} \multimap \mathrm{T} \otimes \mathrm{U}\ (x, y) \triangleq \mathrm{from}'_{\ltimes}(\mathrm{upd}_{\ltimes}\ (\mathtt{新建}_{\ltimes} ::: (\mathrm{T} \otimes \mathrm{U}) \ltimes \lfloor \mathrm{T} \otimes \mathrm{U} \rfloor) \text{ with } d \mapsto \mathtt{分情况}\ (d \triangleleft (,))\ \mathrm{of}\ (d_1, d_2) \mapsto d_1 \blacktriangleleft x \fatsemi d_2 \blacktriangleleft y)$$

内存安全与纯性 这里我们必须让读者放心。当然，以无限制的方式使用目标并不具备内存安全性。为了确保安全，我们需要对目标施加线性纪律。否则，我们可能会遇到两类问题：

- 如果目的地没有至少被写入一次，例如：

  $$\mathtt{忘记} ::: \mathtt{T}\mathtt{忘记} \triangleq \mathtt{from}'_{\ltimes}(\mathtt{更新}_{\ltimes}\ (\mathtt{新建}_{\ltimes} ::: \mathtt{T} \ltimes \lfloor \mathtt{T} \rfloor)\ \mathtt{使用}\ d \mapsto ())$$

  那么，读取 $\mathtt{forget}$ 的结果会导致读取由一个我们从未使用过的目标所指向的位置，换言之，就是读取未初始化内存。这必须不惜一切代价加以避免；

[13]顾名思义，还有一种更一般的消去形式 $\mathtt{from}_{\ltimes}$。我们将在第 2.5 节中讨论它。

· 如果多次写入目的地，如下所示：

ambiguous1 ::: 布尔型 ambiguous1 ≜ from′⅄(upd⅄ (新建⅄ ::: 布尔型 ⅄ ⌊布尔型⌋) with d ↦ d ◀ true ⨟ d ◀ false) ambiguous2 ::: 布尔型 ambiguous2 ≜ from′⅄(upd⅄ (新建⅄ ::: 布尔型 ⅄ ⌊布尔型⌋) with d ↦ let x:=(d ◀ false) in d ◀ true ⨟ x)

于是，由于求值顺序，我们有 `ambiguous1` 返回 `false` ，而 `ambiguous2` 返回 `true` ，因此我们打破了纯函数式语言（包括 λ_d）中本应有效的*let* 展开 。

实际上，我们将在第 2.3 节看到，仅有线性纪律甚至不足以确保在 λ_d 中安全地使用目标。不过，在处理这个问题之前，让我们通过更复杂的示例进一步熟悉 λ_d 。

2.2.2 尾递归映射

让我们看看如何使用目标来构建常见的数据结构。在这些示例中，我们假设 λ_d 具有等递归类型和不动点算子（尽管第 2.5 节的形式系统中并未包含该算子）。

链表 我们像往常一样将列表定义为一个不动点： T 列表 ≜ 1⊕(T⊗(T 列表))。为方便起见，我们还将填充算子 ◁[] 和 ◁(::)定义为宏，它们使用用于和类型、积类型和单位类型的原语目标填充算子：

◁[] ::: ⌊T 列表⌋ ⊸ 1d
◁ [] ≜ d ◁ Inl ◁ ()

◁(::) : ⌊List T⌋ ⊸ ⌊T⌋⊗⌊List T⌋
d ◁ (::) ≜ d ◁ Inr ◁ (,)

就像我们在第 2.2.1 节中所做的那样，我们可以系统地从目标填充算子中还原传统构造器，使用 新建⅄ 、 upd⅄ 和 from′⅄：

(::) ::: T⊗(T 列表) ⊸T 列表
x :: xs ≜ from′⅄(upd⅄ (new⅄ ::: (T 列表)⅄⌊T 列表⌋) ，其中 d ↦分情况 (d ◁ (::)) of (dx， dxs) ↦ dx ◀ x ⨟ dxs ◀ xs)

尾递归映射函数 列表在函数式编程中无处不在，因此，在列表上编写映射函数最自然的方式并非尾递归（因而会消耗无界的栈空间），这一点相当令人遗憾。通过两遍处理，可以让映射变为尾递归：首先以逆序构建结果列表，然后将其反转。不过，多亏了目标，我们完全可以避免这一两遍处理过程，因为它们允许我们直接扩展结果列表的尾部。我们在清单2.1中给出了完整的实现。

这个尾递归函数是 `map′`，其类型为 (T ⊸ U) ω⊸T 列表 ⊸ ⌊U 列表⌋ ⊸ 1[14]。也就是说，它不返回结果列表，而是将目标作为输入，并用结果填充它。每次递归调用中，`map′` 都会新建一个空心 cons 单元来填充目标。一个指向该列表尾部的目标

[14]事实上，我们将从第 2.5 节开始开发一个更复杂的模态系统，以确保对目的地进行作用域控制，从而正确的类型签名应为 (T ⊸ U) ω⊙⊸T 列表 1↑⊸ ⌊U 列表⌋ ⊸ 1。

```
T 列表 ≜ 1⊕(T⊗(T 列表)) map′ ::: (T ⊸ U) ω⊸ T 列表 ⊸ ⌊U 列表
⌋ ⊸ 1map′ f l dl ≜分情况 l of {[] ↦ dl ◁ [], x :: xs ↦ 分情况 (dl ◁
(::)) of (dx , dxs)↦ dx ◀ f x ⨟ map′ f xs dxs}map ::: (T ⊸ U) ω⊸ T
列表 ⊸ U 列表 map f l ≜ from′⊗(upd⊗ (新建⊗ ::: (U 列表) ⊗ ⌊U 列表⌋)
with dl ↦ map′ f l dl)
```

清单2.1： λ_d 中列表上的尾递归 映射函数

还会新建一个 cons 单元，并在其上递归（尾递归）调用 映射′。这实际上与在命令式语言中为可变列表实现 map 时编写的算法相同。然而， λ_d 是一种纯函数式语言，只有不可变类型[15]。

要获得常规的 映射 函数，只需调用 新建⊗ 来创建一个初始目的地，并在所有目的地都已填充后调用 $\mathtt{from}'_{\otimes}$ 提取完成的列表；这与我们将填充算子（例如上文的 (::)）转化为构造器时的做法非常相似。

2.2.3 带目的地的函数式队列

大多数先前的工作中都提供了尾递归映射的实现，从 [Minamide，到较新的研究 [Bour、Clément 和 Scherer（2021年）；Leijen 和 Lorenzen， 2025]。尾递归映射不需要 λ_d 的一等目的地所具备的全部能力：它只需要一种带有（单个）空洞的结构。我们将逐步构造一个能充分发挥一等目的地 λ_d潜力的示例。

差分列表 与任何编程语言中一样，链表的反复串接 $((xs_1 +\!\!+ xs_2) +\!\!+ \ldots) +\!\!+ xs_n$ 在 λ_d 中是二次复杂度。通常，改进这一复杂度的做法是使用差分列表。差分列表 这一名称涵盖了许多相关实现，其概念是“缺少一个队列的链表”。差分列表包含的元素与普通列表相同，但由于我们保留了一种稍后为其队列设置值的方式，因此可以在常数时间内轻松地从后端扩展。在纯函数式语言中，差分列表通常表示为一个函数，而不是 [John Hughes，因为我们通常没有写指针。于是，单例差分列表为 $\boldsymbol{\lambda} ys \mapsto x :: ys$，而差分列表的串接就是函数复合。要将差分列表转换为列表，可以将其队列设为空列表；在函数式的情形中，则是将它应用于空列表。结果是，无论进行多少次函数复合，每个 cons 单元都只会分配一次，因此反复串接确实是线性复杂度。

问题在于，函数式实现中，每次串接仍然会分配一个闭包。如果我们从单例和组合构建差分列表，那么每个 cons 单元大致对应一次组合，因此反复组合实际上会对列表遍历两次。在 λ_d 中，我们实际上有写指针，即 目的地，因此可以做得更好：将差分列表表示为带有空洞的列表，这与命令式环境中的做法很相似。单例差分列表会变成 $x ::$ ，而

[15]我们认为， λ_d 中的数据结构是不可变的，这是因为带孔结构在尚未完全构造完成之前无法被检查，因此与目标相关的变异完全是不透明且不可观测的。

```
DList T ≜ (List T) ⅋ ⌊List T⌋追加 ::: DList T ⊸
T ⊸DList T ys 追加 y ≜upd⅋ ys with dys
 ↦ 分情况 (dys ◁ (::)) of (dy, dys′)↦ dy ◀
y ⨟ dys′concat ::: DList T ⊸DList T ⊸
DList T ys concat ys′ ≜ upd⅋ ys with d ↦
d ◁⊙ys′toList ::: DList T ⊸List T toList ys ≜
from′⅋(upd⅋ys with d ↦ d ◁ [])
```

```
队列 T ≜ (T 列表)⊗(DList T) 单例 ::: T ⊸队
列 T 单例 x ≜ (Inr (x :: []), (新建⅋ ::: DList
T)) 入队 ::: 队列 T ⊸ T ⊸队列 T q 入队 y ≜
分情况 q of (xs, ys) ↦ (xs, ys 追加 y) 出
队 ::: 队列 T ⊸ 1⊕(T⊗(队列 T)) 出队 q ≜分
情况 q of {((x :: xs), ys) ↦ Inr (x, (xs,
ys)), ([], ys) ↦ 分情况 (toList ys) of {[] ↦
Inl (), x :: xs ↦ Inr (x, (xs, (新建⅋ :::
DList T)))}}
```

代码清单 2.2： λ_d 中差分列表和队列的实现

串接就是用另一个差分列表填充空洞，使用复合算子 ◁⊙。详细实现见代码清单 2.2 左侧。差分列表的这种编码不会产生多余的遍历：串接只需进行一次 $O(1)$ 原地更新。

使用差分列表实现高效队列 在不可变函数式语言中，队列可以实现为一对列表（前端、后端）［Hood 和 Melville， 1981］。后端列表以反向顺序存储新元素（$O(1)$ 前置）。我们从前端弹出元素；如果它为空，则将队列设为（反转后端，[]），再从新的前端弹出元素。

尽管实现简单，Hood-Melville 队列的效率却出奇地高：在单线程使用队列时，反转操作的摊销成本为 $O(1)$。不过，最好还是避免对后端列表进行完整遍历。退一步看，这个必须在访问前反转的后端列表，其实只是一个可以从后端扩展的列表的表示。我们已经知道这种列表的高效实现方式：差分列表。

因此，我们可以使用目标给出简单函数式队列的改进版本。该实现见代码清单 2.2 右侧。请注意，与命令式编程语言不同，我们无法将队列实现为单个差分列表：如前所述，我们的类型系统不允许读取差分列表的前端元素。与简单函数式队列一样，我们需要一对列表，一个用于读取，另一个用于扩展。尽管如此，这种队列实现既是纯的（由 λ_d 类型系统保证），效率也几乎与命令式编程语言所能提供的一样高。

2.3 目的地的作用域逃逸

在第 2.2 节中，我们作了一个隐含假设：对目标施加线性纪律，就能确保所有目标最终都会到达填充算子 ◀ 或 ◁的左侧，从而将相应的空洞写入内容。事实证明，这一假设并不完全周全。

要理解其中原因，我们来看看类型 $\lfloor \lfloor \mathsf{T} \rfloor \rfloor$：这是一个指向空洞的目标的类型，而该空洞需要一个目标。可以把它看作 C 语言中指针类型 $\mathsf{T} *$ 的对应类型。目标确实是普通值，因此可以存储在数据结构中；在它们被存储之前，空洞会在结构中占据其位置。例如，假设我们有 $d ::: \lfloor \mathsf{T} \rfloor$ 和 $dd ::: \lfloor \lfloor \mathsf{T} \rfloor \rfloor$，就可以构造 $dd \blacktriangleleft d$：d 将被存储在 dd所指向的结构中。

这里我们是否应将 d 视为线性使用？其他方案似乎都不太可取：

- 如果我们将此视为对 d 的非线性使用，那么 $dd \blacktriangleleft d$ 就会被拒绝，因为目的地（例如这里的 d）只能线性使用。这种选择限制颇多，因为它会使我们无法将目的地存储在带孔结构中，而我们将在第 2.4 节中至关重要地这样做。
- 如果我们完全不将对 d 的这次使用计入，那么我们就可以写出 $dd \blacktriangleleft d \fatsemi d \blacktriangleleft v$，使得 d 既被存储以供后续使用and 又被立即填充（导致对应的空洞可能被写入两次）。正如第 2.2.1 节末尾所讨论的，这样做是不可靠的。

因此，我们选择线性使用。但这带来了一个问题：在线性类型系统中，我们无法区分“一个目标已在三角形左侧使用，因此其对应的空洞已被填充”和“一个目标已被存储，而其空洞此刻仍然存在”。这一疏漏可能会让我们读取未初始化内存！

我们来比较两个例子。暂且假设采用一种简单的存储语义：带孔结构在完成之前会一直保留在存储中。归约步骤从由存储和项组成的一对 $\mathcal{S}|t$ 出发，得到一个新的一对 $\mathcal{S}'|t'$；存储由形如 $\{l :=v\}$ 的命名绑定组成，其中v 是一个可能包含空洞的值。我们还需要$\mathtt{alloc} ::: (\lfloor \mathsf{T} \rfloor \multimap 1) \multimap \mathsf{T}$ 运算符。$\mathtt{alloc}$ 的语义是：在存储中分配一个带有单个根洞的结构，并以指向该根洞的目标作为参数调用给定函数；当函数消耗掉所有目标（因此只剩下单位）时，将该结构从存储中取出，从而得到一个完整的T。$\mathtt{alloc}\ f$ 在类型和行为上对应于$\mathtt{from}'_{\times}(\mathtt{upd}_{\times}\mathtt{new}_{\times}\ \mathtt{with}\ d \mapsto f\ d)$。

在以下代码片段中，带孔结构在存储中绑定到名称 l 和 ld；空洞也有名称，分别记作 h 和 hd，具体目标则记作 $\to h$和 $\to hd$。 □ □

当$l ::: $ 布尔型 的构建作用域是$ld ::: \lfloor$布尔型$\rfloor$构建作用域的父级时，一切都能正常运行，因为ld包含指向h的目标，因此必须先使用ld，然后才能读取l。换句话说，将较旧的目的地存入较新的目的地并不是问题：

$$\{\}\,|\,\mathtt{alloc}\ (\lambda d \mapsto (\mathtt{alloc}\ (\lambda dd \mapsto dd \blacktriangleleft d) ::: \lfloor \text{布尔型} \rfloor) \blacktriangleleft \mathtt{true})$$
$$\longrightarrow \quad \{l := h\}\,|\,(\mathtt{alloc}\ (\lambda dd \mapsto dd \blacktriangleleft \to h) ::: \lfloor \text{布尔型} \rfloor) \blacktriangleleft \mathtt{true} \fatsemi \mathtt{deref}\ l$$
$$\longrightarrow \{l := h, ld := hd\}\,|\,(\to hd \blacktriangleleft \to h \fatsemi \mathtt{deref}\ ld) \blacktriangleleft \mathtt{true} \fatsemi \mathtt{deref}\ l$$
$$\longrightarrow \{l := h, ld := \to h\}\,|\,\mathtt{deref}\ ld \blacktriangleleft \mathtt{true} \fatsemi \mathtt{deref}\ l \longrightarrow \{l := h\}\,|\,\to h \blacktriangleleft \mathtt{true} \fatsemi \mathtt{deref}\ l$$
$$\longrightarrow \square \quad \{l := \mathtt{true}\}\,|\,\mathtt{deref}\ l \longrightarrow \{\}\,|\,\mathtt{true}$$

然而，当ld的作用域是l的父级时，也就是说，当较新的目的地被存储到较旧的目的地中时，我们就能写出一个线性类型化但不健全的程序：

$$\{\}\,|\,alloc\ (\lambda dd \mapsto \text{分情况}\ (alloc\ (\lambda d \mapsto dd \blacktriangleleft d) ::: \text{布尔型})\ of\ \{true \mapsto (), false \mapsto ()\}) \longrightarrow \{ld := hd\}\,|\,\text{分情况}\ (alloc\ (\lambda d \mapsto \to hd \blacktriangleleft d) ::: \text{布尔型})\ of\ \{true \mapsto (), false \mapsto ()\} \fatsemi deref\ ld \longrightarrow \{ld := hd, l := h\}\,|\,\text{分情况}\ (\to hd \blacktriangleleft \to h \fatsemi deref\ l)\ of\ \{true \mapsto (), false \mapsto ()\} \fatsemi deref\ ld \longrightarrow \{ld := \to h, l := h\}\,|\,\text{分情况}\ (deref\ l)\ of\ \{true \mapsto (), false \mapsto ()\} \fatsemi deref\ ld \longrightarrow \{ld := \to h\}\,|\,\text{分情况}\ h\ of\ \{true \mapsto (), false \mapsto ()\} \fatsemi deref\ ld$$ □ 🐛🐛🐛

这里，表达式 $dd \blacktriangleleft d$ 会导致d 逃逸出其作用域，进入父级作用域，因此l 只是未初始化内存（空洞h），在对其进行解引用时便会如此。我们的类型系统必须拒绝此示例。

一个相关问题是兄弟目标之间如何相互作用：

$$\{\}\mid \text{分配}\ (\lambda d \mapsto \text{分情况}\ (d \triangleleft (,))\ \text{of}\ (d' ::: \lfloor \text{布尔型} \rfloor, dd' ::: \lfloor\lfloor \text{布尔型} \rfloor\rfloor) \mapsto dd' \blacktriangleleft d') ::: \text{布尔型} \otimes \lfloor \text{布尔型} \rfloor \longrightarrow \{l := h\} \mid \text{分情况}\ (\rightarrow h \triangleleft (,))\ \text{of}\ (d' ::: \lfloor \text{布尔型} \rfloor, dd' ::: \lfloor\lfloor \text{布尔型} \rfloor\rfloor) \mapsto dd' \blacktriangleleft d' \fatsemi \text{解引用}\ l \longrightarrow \{l := (h', hd')\} \mid \text{分情况}\ (\rightarrow h', \rightarrow hd')\ \text{of}\ (d' ::: \lfloor \text{布尔型} \rfloor, dd' ::: \lfloor\lfloor \text{布尔型} \rfloor\rfloor) \mapsto dd' \blacktriangleleft d' \fatsemi \text{解引用}\ l \longrightarrow \{l := (h', \rightarrow hd')\} \mid \rightarrow hd' \blacktriangleleft \rightarrow h' \fatsemi \text{解引用}\ l \longrightarrow \{l := (h', \rightarrow h')\} \mid \text{解引用}\ l \longrightarrow \{\} \mid (h', \rightarrow h')$$

这里同样有一个空洞 h' 从存储中逃逸，这意味着，如果我们对上面代码块的结果进行模式匹配，最终也会读到未初始化内存！

问题依旧在于，如果只使用线性类型系统，我们就只能通过同时拒绝第一个正确的例子，来拒绝最后两个例子。在这种情况下，类型 $\lfloor\lfloor \mathsf{T} \rfloor\rfloor$ 实际上会变得毫无用处：这样的目标永远无法被填充。这不是我们想要的方向：我们确实希望能够将目标存储在带孔结构中。因此，我们希望允许 t in $d \blacktriangleleft t$ 具有线性类型。若没有额外的限制，这样做并不健全；为了解决这个问题，我们需要额外的控制系统来阻止作用域逃逸；仅靠线性性是不够的。更具体地说，我们必须确保：如果一个目标 d 被存储在另一个目标 dd中，那么 d 必须源自一个严格更早的作用域，也就是说，该作用域必须在 dd的作用域开启之前开启。这个条件排除了我们刚才讨论的两个有问题的例子。

对于 λ_d，我们决定使用一套年龄来表示资源源自哪个作用域。年龄将在第 2.5 节中详细介绍。在本文档的其余部分，我们经常使用作用域逃逸来笼统指代上文提出的两个问题：一种情况是$dd \blacktriangleleft d$ 来自严格更年轻的作用域，或者另一种情况是d 与dd来自同一作用域，因为这两种情况通常都可以使用相同的技术或技巧加以避免。

2.4 广度优先树遍历

在转入形式化细节之前，让我们先看一个完整的示例，该示例使用了 λ_d 的全部表达能力。这里我们关注广度优先树重标记：“给定一棵树，创建一棵形状相同的新树，但将节点中的值替换为按照 $1 \ldots |T|$ 广度优先顺序排列的数字。”

这并不是一个非常自然的函数式编程问题，因为广度优先遍历意味着构建结构的顺序（从左到右、从上到下）不同于函数式树的结构顺序——先构建叶节点，再向上构建到根节点。因此，它通常需要巧妙的函数式变通方法 [Gibbons, 1993; Gibbons 等人, 2023; Okasaki, 2000]。

我们可能会倾向于用高效、类似命令式编程的方式解决这个练习，让队列决定处理顺序。这是大学里教授的标准算法：从队列前端出队下一个待处理节点，并将其子节点入队到队列后端，以便稍后处理，从而实现广度优先遍历。为此，Minamide 系统 [Minamide, 1998] 其中带孔结构表示为线性函数，并无助益。我们确实需要将目的地作为一等值，才能借助这种命令式编程的强大能力。

```
go ::: (S ⊸ T₁ ⊸ (!S)⊗T₂) ⊸ S ⊸ Queue (Tree T₁⊗⌊Tree T₂⌋) ⊸ 1 递
归 go f st q ≜ 分情况 (dequeue q) of {Inl () ↦ () , Inr ((tree , dtree) , q′) ↦ 分情
况 tree of {Nil ↦ dtree ◁ Nil ⨟ go f st q′, Node x tl tr ↦ 分情况 (dtree ◁
Node) of (dy , (dtl , dtr)) ↦ 分情况 (f st x) of (Mod⊸st′ , y) ↦dy ◀ y ⨟go f st′ (q′
enqueue (tl , dtl) enqueue (tr , dtr))}}mapAccumBfs ::: (S ⊸ T₁ ⊸ (!S)⊗T₂)
⊸ S ⊸ Tree T₁ 1⊸ Tree T₂mapAccumBfs f st tree ≜ from′ₗ(updₗ (新建ₗ :::
(Tree T₂)ₗ ⌊Tree T₂⌋) with dtree ↦relabelDps ::: Tree 1 1⊸ Tree 自然数
relabelDps tree ≜ mapAccumBfs (λst ⊸↦ λun ↦ un ⨟(Mod⊸ (succ st) ,
st)) 1
```

go *f st* (**singleton** (*tree* , *dtree*)))

tree

列表 2.3：使用 λ_d 中的目的地传递进行广度优先树遍历

列表 2.3 展示了借助一等目的地，如何在 λ_d 中实现这种广度优先树遍历。我们假设数据类型 `Tree T` 已定义为 `Tree T` ≜ $1\oplus(T\otimes((\texttt{Tree}\ T)\otimes(\texttt{Tree}\ T)))$；并将 `Tree T` 的构造器称为 `Nil` 和 `Node`，其定义方式与之前其他构造器相同，使用了语法糖。我们还假设自然数类型 `Nat` 采用了某种编码。请记住，`Queue T` 是第 2.2.3 节中的高效队列类型。

我们算法的核心思想是维护一个由对组成的队列，每个对都存储一个输入子树以及（指向）其对应输出子树的一个目标。队列前端的元素（*tree*, *dtree*）处理完毕后，*tree* 的子节点及 *dtree* 的子目标会入队，以便稍后处理，这与原始命令式算法的做法大致相同。

我们将实际的广度优先重标记 **relabelDps** 实现为更通用的广度优先遍历函数 **mapAccumBfs** 的一个实例。该函数会按广度优先顺序对标签应用任意状态传递风格变换。在 **mapAccumBfs** 中，我们新建一个目标 *dtree*，用于写入遍历结果，然后调用 **go**。**go** 函数采用目标传递风格，但值得注意的是，**go** 的参数中可以包含任意多个目标，因为队列中有多少个项目，就有多少个目标。这正是我们利用目标属于普通值这一事实的地方。

不过，这种新获得的表达能力也有代价：我们需要一种结合线性控制和年龄控制的类型系统，才能保证系统的健全性，否则就会遇到上一节所述的作用域逃逸问题。我们将在同一个模态半环[16]中结合线性性和前述年龄系统。你已经在第 1 章了解过线性注解1 和 ω；在此基础上，我们还引入新的年龄注解 ∞，用于表示相应的参数不能携带目标。未显示任何模态的参数，或未标注模态的函数箭头，默认取半环的单位；特别地，它们是线性的，并且可以捕获目标。

[16]我们在第 1.4 节中提到，采用基于半环的模态方法可以让类型系统更容易扩展。以下是证明。

第2章。基于一等目的地的形式函数式语言： λ_d

项的核心语法：

$$
\begin{aligned}
t, u \ ::= \ & x \mid t'\ t \mid t \fatsemi t' \\
& \mid \mathsf{case}_{\mathsf{m}}\ t\ \mathsf{of}\ \{\mathsf{Inl}\ x_1 \mapsto u_1\ ,\ \mathsf{Inr}\ x_2 \mapsto u_2\} \mid \mathsf{case}_{\mathsf{m}}\ t\ \mathsf{of}\ (x_1\ ,\ x_2) \mapsto u \mid \mathsf{case}_{\mathsf{m}}\ t\ \mathsf{of}\ \mathsf{Mod}_{\mathsf{n}}\ x \mapsto u \\
& \mid \mathsf{upd}_{\ltimes}\ t\ \mathsf{with}\ x \mapsto t' \mid \mathsf{to}_{\ltimes}\ t \mid \mathsf{from}_{\ltimes}\ t \mid \mathsf{new}_{\ltimes} \\
& \mid t \lhd () \mid t \lhd \mathsf{Inl} \mid t \lhd \mathsf{Inr} \mid t \lhd (,) \mid t \lhd \mathsf{Mod}_{\mathsf{m}} \mid t \lhd (\boldsymbol{\lambda} x\ {}_{\mathsf{m}}\!\!\mapsto u) \mid t \lhd\!\!\circ\, t' \mid t \blacktriangleleft t'
\end{aligned}
$$

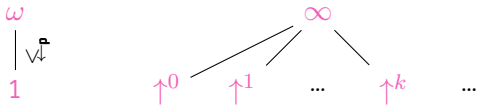
类型、模态和上下文的语法：

$$
\begin{aligned}
\mathsf{T}, \mathsf{U}, \mathsf{S} \ ::= \ & \lfloor_{\mathsf{n}} \mathsf{T} \rfloor \quad \textit{(destination)} \\
& \mid \mathsf{S} \ltimes \mathsf{T} \quad \textit{(ampar)} \\
& \mid \mathsf{1} \mid \mathsf{T}_1 \oplus \mathsf{T}_2 \mid \mathsf{T}_1 \otimes \mathsf{T}_2 \mid !_{\mathsf{m}} \mathsf{T} \mid \mathsf{T}\ {}_{\mathsf{m}}\!\!\multimap \mathsf{U}
\end{aligned}
\qquad
\begin{aligned}
\mathsf{m}, \mathsf{n} \ ::= \ & \mathsf{pa} \quad \textit{(pair of multiplicity and age)} \\
\mathsf{p} \ ::= \ & 1 \mid \omega \\
\mathsf{a} \ ::= \ & \uparrow^k \mid \infty
\end{aligned}
$$

$$
\Gamma \ ::= \ \bullet \mid x :_{\mathsf{m}} \mathsf{T} \mid \Gamma_1, \Gamma_2
\qquad
\nu \triangleq \uparrow^0 \quad \uparrow \triangleq \uparrow^1
$$

模态上的序：

$$
\mathsf{pa} \leqslant \mathsf{p'a'} \iff \mathsf{p} \leqslant_{\mathsf{p}} \mathsf{p'} \wedge \mathsf{a} \leqslant_{\mathsf{a}} \mathsf{a'}
$$



模态上的操作：

$+$	1	ω
1	ω	ω
ω	ω	ω

$\cdot$	1	ω
1	1	ω
ω	ω	ω

$(\text{爸}) \cdot (\mathsf{p'a'}) \triangleq (\mathsf{p} \cdot \mathsf{p'})(\mathsf{a} \cdot \mathsf{a'})$

$+$	$\uparrow^k$	∞
$\uparrow^j$	如果 $k = j$ 那么 $\uparrow^k$ 否则 ∞	∞
∞	∞	∞

$\cdot$	$\uparrow^k$	∞
$\uparrow^j$	$\uparrow^{k+j}$	∞
∞	∞	∞

$(\mathsf{pa}) + (\mathsf{p'a'}) \triangleq (\mathsf{p} + \mathsf{p'})(\mathsf{a} + \mathsf{a'})$

类型上下文上的操作：

$$
\mathsf{n} \cdot \bullet \triangleq \bullet
\qquad
\mathsf{n} \cdot (x :_{\mathsf{m}} \mathsf{T},\ \Gamma) \triangleq (x :_{\mathsf{n} \cdot \mathsf{m}} \mathsf{T}),\ \mathsf{n} \cdot \Gamma
$$

$$
\begin{aligned}
\bullet + \Gamma &\triangleq \Gamma \\
(x :_{\mathsf{m}} \mathsf{T}\ \ \Gamma_1) + \Gamma_2 &\triangleq x :_{\mathsf{m}} \mathsf{T}, (\Gamma_1 + \Gamma_2) \qquad \text{如果} x \not\in \Gamma_2 \\
(x :_{\mathsf{m}} \mathsf{T},\ \Gamma_1) + (x :_{\mathsf{m}'} \mathsf{T}\ \ \Gamma_2) &\triangleq x :_{\mathsf{m}+\mathsf{m}'} \mathsf{T}, (\Gamma_1 + \Gamma_2)
\end{aligned}
$$

图 2.1： λ_d 的项、类型和模态

2.5 类型系统

λ_d 是一个带有单位（1）、积（$\otimes$）和和类型（$\oplus$）的简单类型 λ 演算，与第 1 章中 λ_{Ldm} 所述类似。它还具有模态多态函数箭头 $_{\mathsf{m}}\!\!\multimap$。然而，它最显著的特点是我们在第 2.2 至 2.4 节中引入的目标 $\lfloor_{\mathsf{m}} \mathsf{T} \rfloor$ 和 ampar $\mathsf{S} \ltimes \mathsf{T}$ 类型。目标类型 $\lfloor_{\mathsf{m}} \mathsf{T} \rfloor$ 上的模态注释 m 表示写入相应空洞的值必须具有的模态；迄今为止，示例中一直隐藏着这一注释，因为我们只考虑了能够接受单位模态值的目标。

为了确保目的地得到合理使用，我们不仅需要强制保证目的地的线性性，还需要防止目的地逃逸出其作用域，正如第 2.3 节所述。为此， λ_d 会追踪目的地的年龄，也就是当前表达式与目标来源作用域之间打开了多少层嵌套作用域。我们将在第 2.5.2 节看到，作用域由$\mathsf{upd}_{\ltimes} t\ \mathsf{with}\ x \mapsto t'$ 结构引入。例如，若有项$\mathsf{upd}_{\ltimes}\ t_1\ \mathsf{with}\ x_1 \mapsto \mathsf{upd}_{\ltimes} t_2\ \mathsf{with}\ x_2 \mapsto \mathsf{upd}_{\ltimes} t_3\ \mathsf{with}\ x_3 \mapsto x_1$,，那么我们称 x_1 最内层出现的年龄为 $\uparrow^2$，因为两个嵌套的$\mathsf{upd}_{\ltimes}$ 将x_1的定义位置与使用位置隔开。

$$\mathsf{Inl}\,t \triangleq \mathsf{from}'_{\ltimes}(\mathsf{upd}_{\ltimes}\ \text{新建}_{\ltimes}\ \mathsf{with}\ d \mapsto d \triangleleft \mathsf{Inl} \blacktriangleleft t)$$

$$\mathsf{Inr}\,t \triangleq \mathsf{from}'_{\ltimes}(\mathsf{upd}_{\ltimes}\ \text{新建}_{\ltimes}\ \mathsf{with}\ d \mapsto d \triangleleft \mathsf{Inr} \blacktriangleleft t)$$

$$\mathsf{Mod}_{m}\,t \triangleq \mathsf{from}'_{\ltimes}(\mathsf{upd}_{\ltimes}\ \text{新建}_{\ltimes}\ \mathsf{with}\ d \mapsto d \triangleleft \text{模块}_{m} \blacktriangleleft t)$$

$$\boldsymbol{\lambda} x\,{}_{m}\!\mapsto u \triangleq \mathsf{from}'_{\ltimes}(\mathsf{upd}_{\ltimes}\ \text{新建}_{\ltimes}\ \mathsf{with}\ d \mapsto d \triangleleft (\boldsymbol{\lambda} x\,{}_{m}\!\mapsto u))$$

$$\mathsf{from}'_{\ltimes}\,t \triangleq \text{分情况}(\mathsf{from}_{\ltimes}\ (\mathsf{upd}_{\ltimes}\ t\ \mathsf{with}\ un \mapsto un\ ⨟ \mathsf{Mod}_{1\infty}\ ()))\ \mathsf{of}\ (st,\ ex) \mapsto \text{分情况}\ ex\ \text{关于}\ \text{模块}_{1\infty} un \mapsto un\ ⨟ st$$

$$() \triangleq \mathsf{from}'_{\ltimes}(\text{更新}_{\ltimes}\text{新建}_{\ltimes}\ \text{与}\ d \mapsto d \triangleleft ())$$

$$(t_1\,,\,t_2) \triangleq \mathsf{from}'_{\ltimes}(\text{更新}_{\ltimes}\ \text{新建}_{\ltimes}\ \text{与}\ d \mapsto \text{分情况}(d \triangleleft (,\ ))\ \text{的}\ (d_1\,,\,d_2) \mapsto d_1 \blacktriangleleft t_1\ ⨟\ d_2 \blacktriangleleft t_2)$$

代码清单 2.4： λ_d中各项的去糖化规则

对于 λ_d，我们采用与 λ_{Ldm} 相同的模态方法，但扩展了模态半环，使其包含一个年龄 轴。得益于模态方法的代数性质，对于大多数类型规则，我们都可以直接复用 λ_{Ldm} 的规则，无须做任何修改，因为改变的只是半环的元素，而不是半环的性质或结构。

λ_d 项的语法见图 2.1，第 2.2 至 2.4 节中使用的语法糖则见代码清单 2.4。

2.5.1 模态与年龄半环

我们的模态半环（更准确地说，是一个交换加法半群 $+$ 和一个乘法幺半群 $(\cdot,\ 1)$，满足通常的分配律 $\mathsf{n}\cdot(\mathsf{m}_1 + \mathsf{m}_2) = \mathsf{n}\cdot\mathsf{m}_1 + \mathsf{n}\cdot\mathsf{m}_2$,，并配备偏序 $\leq$，使得 $+$ 和 $\cdot$ 保序）正如所承诺的，是重数半环与年龄半环的积：前者用于跟踪线性性，后者用于防止作用域逃逸。由此得到的复合结构也是一个半环。

重数半环包含元素 1 （线性）和 ω （无限制的）；它与 [Atkey，2018] 或 [Bernardy 等人，2018]，所用的半环相同，也与第 1 章中详细描述的半环相同。图 2.1 再次给出了重数半环的完整描述。

年龄更有意思。我们将年龄写作 $\uparrow^k$ （其中 k 是一个自然数），表示“在所定义的作用域之前 k 个作用域”。对于不源自 $\mathsf{upd}_{\ltimes} t\ \mathsf{with}\ x \mapsto t'$ 构造的变量，也有一个年龄 ∞；这类变量不是目标，可以在任何作用域中自由使用并由其返回。因此，年龄 ∞ 的主要作用是保证一个值不包含目标。最后，我们将 $\nu \triangleq \uparrow^0$ （“现在”）记作源自当前作用域的目标的年龄；并且 $\uparrow \triangleq \uparrow^1$。

不过，年龄上的操作或序并非自然数上的常见操作或序。事实上， λ_d 追踪变量的精确年龄至关重要；通常，模式上的序意味着某种形式的子类型关系（如果$\mathsf{n} \leq \mathsf{n}$，那么模式为$\mathsf{n}$ 的变量就可以用于需要模式为m 变量的位置）。在这里，我们不希望将来自两层作用域之前的变量当作来自一层作用域之前的变量使用。年龄序通过将有限年龄排列成平坦序来体现这一点（因此 $\uparrow^1$ 和 $\uparrow^2$ 不可比较），而 ∞ 大于所有有限年龄。年龄的乘法体现了作用域的嵌套，因此，（有限）年龄相乘时，将其数值指数相加即可。在类型规则中，最常见的作用域嵌套形式是打开一个作用域，这通过将现有绑定的年龄乘以 $\uparrow$ 来表示（也就是将年龄视为自然数后加 1）。当同一个变量在

两个子项之间共享时， $+$ 会取上述年龄序中的最小上界；换句话说，该变量必须在两个子项中年龄相同，否则其年龄为 ∞，这样它就可以在每个子项中采用所需的年龄。

新模态半环的单位是重数半环和年龄半环的单位所组成的对，即 1_ν。当模态为该单位时，我们通常会省略模态注释。

最后，正如第 1 章中的 λ_{Lm} 和 λ_{Ldm} 一样，模式上的操作也会提升到类型上下文中，同时仍遵循 ⟦Ghica 和 Smith 的见解， 2014⟧（见图 2.1）。

预先展望下一节，让我们看看 *plus* 如何作用于上下文、模态，尤其是年龄。让我们为表达式 $d \blacktriangleleft x \fatsemi x$ 进行类型推导。这里的大部分类型推导与 λ_{Ldm}–ty 相同，唯一的特殊之处在于，运算符 $\blacktriangleleft$ 会将其右操作数的类型上下文按 $\uparrow$ 进行缩放（这相当于要求它来自前一个作用域，以防止目的地的作用域逃逸）。我们得到如下类型树：

$$\dfrac{\dfrac{\dfrac{}{d :_{1_\nu} \lfloor \mathsf{T} \rfloor \vdash d : \lfloor \mathsf{T} \rfloor}\,\mathrm{ID_V} \quad \dfrac{}{x :_{1_\nu} \mathsf{T} \vdash x : \mathsf{T}}\,\mathrm{ID_V}}{d :_{1_\nu} \lfloor \mathsf{T} \rfloor,\ x :_{1\uparrow} \mathsf{T} \vdash d \blacktriangleleft x : 1}\,\lfloor\rfloor\mathrm{E_L} \quad \dfrac{}{x :_{1_\nu} \mathsf{T} \vdash x : \mathsf{T}}\,\mathrm{ID_V}}{d :_{1_\nu} \lfloor \mathsf{T} \rfloor,\ x :_{\omega\infty} \mathsf{T} \vdash d \blacktriangleleft x \fatsemi x : \mathsf{T}}\,1\mathrm{E}$$

在树的根节点处，运算符为 $\fatsemi$，两个初始分支的类型上下文相加，因此我们有 $(\mathrm{d} :_{1_\nu} \lfloor \mathsf{T} \rfloor, \mathrm{x} :_{1\uparrow} \mathsf{T}) + (\mathrm{x} :_{1_\nu} \mathsf{T}) = \mathrm{d} :_{1_\nu} \lfloor \mathsf{T} \rfloor, \mathrm{x} :_{(1+1)(\nu+\uparrow)} \mathsf{T} = \mathrm{d} :_{1_\nu} \lfloor \mathsf{T} \rfloor, \mathrm{x} :_{\omega\infty} \mathsf{T}$。换句话说，因为我们需要 x 在第一个分支中的年龄为 $\uparrow$，而在第二个分支中的年龄为 ν——我们回想一下， $\nu = \uparrow^0$ 和 $\uparrow = \uparrow^1$ 不可比较，因此不能对模式为 $1\uparrow$ 的变量使用规则 λ_d–ty/id$_\mathrm{V}$ ——所以 x 的年龄必须为 ∞[17]。

或者，我们可以这样：

$$\dfrac{\dfrac{\dfrac{}{d :_{1_\nu} \lfloor \mathsf{T} \rfloor \vdash d : \lfloor \mathsf{T} \rfloor}\,\mathrm{ID_V} \quad \dfrac{}{x :_{\omega\infty} \mathsf{T} \vdash x : \mathsf{T}}\,\mathrm{ID_V}}{d :_{1_\nu} \lfloor \mathsf{T} \rfloor,\ x :_{\omega\infty} \mathsf{T} \vdash d \blacktriangleleft x : 1}\,\lfloor\rfloor\mathrm{E_L} \quad \dfrac{}{x :_{\omega\infty} \mathsf{T} \vdash x : \mathsf{T}}\,\mathrm{ID_V}}{d :_{1_\nu} \lfloor \mathsf{T} \rfloor,\ x :_{\omega\infty} \mathsf{T} \vdash d \blacktriangleleft x \fatsemi x : \mathsf{T}}\,1\mathrm{E}$$

其中 x 从叶节点到根节点被赋予模式 $\omega\infty$。我们可以使用模式为 $\omega\infty$ 的变量，因为我们有 $1_\nu \leq \omega\infty$，这符合规则 λ_d–ty/id$_\mathrm{V}$ 的前提。

2.5.2 类型规则

核心语言 λ_d 的核心类型规则见图 2.2。关于消去 $\multimap$、1、$\oplus$、$\otimes$ 以及 $!_\mathsf{m}$ 的规则与第 1 章中 λ_{Ldm} 的规则相同，因此这里不再赘述。

不过，与 λ_{Ldm} 相比，一个显著的不同之处在于，正如前文所述，数据结构（即类型 1、$\oplus$、$\otimes$ 和 $!_\mathsf{m}$的数据构造器）的引入规则是从目标的消去规则推导而来的，因此不属于核心语言。这些规则单独列于图 2.3。函数 λ_d–TYs/$\multimap$I 的引入形式也可由相应函数目标的消去形式 λ_d–TY/$\lfloor\multimap\rfloor$E 推导而来（因此，除 ampar 和目标本身之外，所有类型的引入形式都可由某个目标消去形式推导得出）。

[17]年龄要求仅适用于范围敏感资源，也就是说，仅适用于目的地及其存储所在的结构。年龄 ∞ 用于表示非范围敏感资源，因此也表示不兼容的年龄要求的结果（因为只有非范围敏感资源才能满足不兼容的年龄要求）。

$\boxed{\Gamma \vdash t : \mathsf{T}}$ （项的类型判断）

$$\frac{1\nu \leq \mathsf{m}}{\omega\nu \cdot \Gamma,\ x :_{\mathsf{m}} \mathsf{T} \vdash x : \mathsf{T}}\ \mathrm{ID_V} \qquad \frac{\Gamma_1 \vdash t : \mathsf{T} \qquad \Gamma_2 \vdash t' : \mathsf{T} \multimap_{\mathsf{m}} \mathsf{U}}{\mathsf{m} \cdot \Gamma_1 + \Gamma_2 \vdash t'\ t : \mathsf{U}}\ \multimap\mathrm{E} \qquad \frac{\Gamma_1 \vdash t : \mathsf{1} \qquad \Gamma_2 \vdash u : \mathsf{U}}{\Gamma_1 + \Gamma_2 \vdash t \fatsemi u : \mathsf{U}}\ \mathsf{1}\mathrm{E}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash t : \mathsf{T}_1 \oplus \mathsf{T}_2 \\ \Gamma_2,\ x_1 :_{\mathsf{m}} \mathsf{T}_1 \vdash u_1 : \mathsf{U} \\ \Gamma_2,\ x_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash u_2 : \mathsf{U}\end{array}}{\mathsf{m} \cdot \Gamma_1 + \Gamma_2 \vdash \mathsf{case}_{\mathsf{m}}\ t\ \mathsf{of}\ \{\mathsf{Inl}\, x_1 \mapsto u_1\ ,\ \mathsf{Inr}\, x_2 \mapsto u_2\} : \mathsf{U}}\ \oplus\mathrm{E}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash t : \mathsf{T}_1 \otimes \mathsf{T}_2 \\ \Gamma_2,\ x_1 :_{\mathsf{m}} \mathsf{T}_1,\ x_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash u : \mathsf{U}\end{array}}{\mathsf{m} \cdot \Gamma_1 + \Gamma_2 \vdash \mathsf{case}_{\mathsf{m}}\ t\ \mathsf{of}\ (x_1\ ,\ x_2) \mapsto u : \mathsf{U}}\ \otimes\mathrm{E} \qquad \frac{\begin{array}{c}\Gamma_1 \vdash t : !_{\mathsf{n}} \mathsf{T} \\ \Gamma_2,\ x :_{\mathsf{m} \cdot \mathsf{n}} \mathsf{T} \vdash u : \mathsf{U}\end{array}}{\mathsf{m} \cdot \Gamma_1 + \Gamma_2 \vdash \mathsf{case}_{\mathsf{m}}\ t\ \mathsf{of}\ \mathsf{Mod}_{\mathsf{n}}\ x \mapsto u : \mathsf{U}}\ !\mathrm{E}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash t : \mathsf{U} \ltimes \mathsf{T} \\ 1\uparrow \cdot \Gamma_2,\ x :_{1\nu} \mathsf{T} \vdash t' : \mathsf{T}'\end{array}}{\Gamma_1 + \Gamma_2 \vdash \mathsf{upd}_{\ltimes}\ t\ \mathsf{with}\ x \mapsto t' : \mathsf{U} \ltimes \mathsf{T}'}\ \ltimes\mathrm{UPD} \qquad \frac{\Gamma \vdash u : \mathsf{U}}{\Gamma \vdash \mathsf{to}_{\ltimes}\ u : \mathsf{U} \ltimes \mathsf{1}}\ \ltimes\mathrm{TO}$$

$$\frac{\Gamma \vdash t : \mathsf{U} \ltimes (!_{1\infty} \mathsf{T})}{\Gamma \vdash \mathsf{from}_{\ltimes}\ t : \mathsf{U} \otimes (!_{1\infty} \mathsf{T})}\ \ltimes\mathrm{FROM} \qquad \frac{}{\omega\nu \cdot \Gamma \vdash \mathsf{new}_{\ltimes} : \mathsf{T} \ltimes \lfloor_{1\nu} \mathsf{T} \rfloor}\ \ltimes\mathrm{NEW} \qquad \frac{\Gamma \vdash t : \lfloor_{\mathsf{n}} \mathsf{1} \rfloor}{\Gamma \vdash t \triangleleft () : \mathsf{1}}\ \lfloor \mathsf{1} \rfloor\mathrm{E}$$

$$\frac{\Gamma \vdash t : \lfloor_{\mathsf{n}} \mathsf{T}_1 \oplus \mathsf{T}_2 \rfloor}{\Gamma \vdash t \triangleleft \mathsf{Inl} : \lfloor_{\mathsf{n}} \mathsf{T}_1 \rfloor}\ \lfloor\oplus\rfloor\mathrm{E}_1 \qquad \frac{\Gamma \vdash t : \lfloor_{\mathsf{n}} \mathsf{T}_1 \oplus \mathsf{T}_2 \rfloor}{\Gamma \vdash t \triangleleft \mathsf{Inr} : \lfloor_{\mathsf{n}} \mathsf{T}_2 \rfloor}\ \lfloor\oplus\rfloor\mathrm{E}_2 \qquad \frac{\Gamma \vdash t : \lfloor_{\mathsf{n}} \mathsf{T}_1 \otimes \mathsf{T}_2 \rfloor}{\Gamma \vdash t \triangleleft (,) : \lfloor_{\mathsf{n}} \mathsf{T}_1 \rfloor \otimes \lfloor_{\mathsf{n}} \mathsf{T}_2 \rfloor}\ \lfloor\otimes\rfloor\mathrm{E}$$

$$\frac{\Gamma \vdash t : \lfloor_{\mathsf{n}} !_{\mathsf{n}'} \mathsf{T} \rfloor}{\Gamma \vdash t \triangleleft \mathsf{Mod}_{\mathsf{n}'} : \lfloor_{\mathsf{n}' \cdot \mathsf{n}} \mathsf{T} \rfloor}\ \lfloor!\rfloor\mathrm{E} \qquad \frac{\begin{array}{c}\Gamma_1 \vdash t : \lfloor_{\mathsf{n}} \mathsf{T} \multimap_{\mathsf{m}} \mathsf{U} \rfloor \\ \Gamma_2,\ x :_{\mathsf{m}} \mathsf{T} \vdash u : \mathsf{U}\end{array}}{\Gamma_1 + (1\uparrow \cdot \mathsf{n}) \cdot \Gamma_2 \vdash t \triangleleft (\lambda x\ {}_{\mathsf{m}}\!\mapsto u) : \mathsf{1}}\ \lfloor\multimap\rfloor\mathrm{E}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash t : \lfloor_{1\nu} \mathsf{U} \rfloor \\ \Gamma_2 \vdash t' : \mathsf{U} \ltimes \mathsf{T}\end{array}}{\Gamma_1 + 1\uparrow \cdot \Gamma_2 \vdash t \triangleleft\!\circ\, t' : \mathsf{T}}\ \lfloor\ \rfloor\mathrm{E_C} \qquad \frac{\begin{array}{c}\Gamma_1 \vdash t : \lfloor_{\mathsf{n}} \mathsf{T} \rfloor \\ \Gamma_2 \vdash t' : \mathsf{T}\end{array}}{\Gamma_1 + (1\uparrow \cdot \mathsf{n}) \cdot \Gamma_2 \vdash t \blacktriangleleft t' : \mathsf{1}}\ \lfloor\ \rfloor\mathrm{E_L}$$

图 2.2： λ_d （λ_d–ty）的类型规则

$\boxed{\Gamma \vdash t : \mathsf{T}}$ *(语法糖形式的导出类型判断)*

$$\frac{\Gamma \vdash t ::: \mathsf{T} \ltimes 1}{\Gamma \vdash \mathsf{from}'_{\ltimes}\, t : \mathsf{T}} \ltimes\text{from'} \qquad \frac{}{\omega\nu \cdot \Gamma \vdash () : 1}\ 1\mathrm{I} \qquad \frac{\Gamma_2,\ x :_{\mathsf{m}} \mathsf{T} \vdash u : \mathsf{U}}{\Gamma_2 \vdash \lambda x._{\mathsf{m}} u ::: \mathsf{T} \multimap_{\mathsf{m}} \mathsf{U}} \multimap\mathrm{I}$$

$$\frac{\Gamma_2 \vdash t : \mathsf{T}_1}{\Gamma_2 \vdash \mathrm{Inl}\, t ::: \mathsf{T}_1 \oplus \mathsf{T}_2} \oplus\mathrm{I}_1 \qquad \frac{\Gamma_2 \vdash t : \mathsf{T}_2}{\Gamma_2 \vdash \mathrm{Inr}\, t ::: \mathsf{T}_1 \oplus \mathsf{T}_2} \oplus\mathrm{I}_2 \qquad \frac{\Gamma_2 \vdash t : \mathsf{T}}{\mathsf{m} \cdot \Gamma_2 \vdash \text{模}_{\mathsf{m}} t ::: !_{\mathsf{m}}\mathsf{T}} !\ \text{我}$$

$$\frac{\Gamma_{21} \vdash t_1 : \mathsf{T}_1 \qquad \Gamma_{22} \vdash t_2 : \mathsf{T}_2}{\Gamma_{21} + \Gamma_{22} \vdash (t_1, t_2) ::: \mathsf{T}_1 \otimes \mathsf{T}_2} \otimes\mathrm{I}$$

图 2.3：语法糖的导出类型规则（λ_d–ty$_s$）

规则 λ_d–ty/id$_V$、λ_d–ty/$\ltimes$new 和 λ_d–ty$_s$/1I 是类型树中可能的叶节点，允许在这些位置对无限制变量进行弱化。因此，它们允许丢弃任何形如 $\omega\nu \cdot \Gamma$ 的类型上下文。这与 λ_{Ldm}–ty/id 或 λ_{Ldm}–ty/1I 中的情况非常相似，只不过现在 Γ 由 $\omega\nu$ 缩放，而不是由 ω 缩放。

规则 λ_d–ty/id$_V$ 除了允许弱化，还允许对所用变量的模式 m 进行强制转换，并受图 2.1 中定义的序约束 $1\nu \leqslant \mathsf{m}$ 限制。与 λ_{Ldm} 中不同，这里并非所有模式 m 都与 1ν 兼容。值得注意的是，模态强制仍不允许将有限年龄改为另一种有限年龄，因为当 $j \neq k$ 时，$\uparrow^j$ 和 $\uparrow^k$ 关于 $\leqslant_{\mathbb{a}}$ 不可比较。因此，例如，我们不能在大多数上下文中使用模式为 $1\uparrow$ 的变量。

对于模式匹配，使用规则 λ_d–ty/$\oplus$E、λ_d–ty/$\otimes$E 和 λ_d–ty/!E，我们沿用 λ_{Ldm} 中的相同深层模式方法：分情况 表达式由模式m 参数化，类型上下文 Γ_1 中的判别式会乘以该模式。随后，绑定判别式各个子分量的变量也会继承此模式。

现在我们来讨论 λ_d 类型规则中的真正新内容，即针对作用于 ampar 和目的地的运算符所制定的规则。

作用域划分规则 由于目的地始终存在于带孔结构的上下文中，并且必须保持在该上下文中，因此我们需要对作用域作出形式化定义。作用域由 λ_d–ty/$\ltimes$upd 创建，因为目的地只能通过$\mathsf{upd}_{\ltimes}$访问。更准确地说，$\mathsf{upd}_{\ltimes}\ t\ \mathsf{with}\ x \mapsto t'$ 会创建一个新作用域，其范围覆盖t'。在该作用域中，x 的年龄为 ν（当前年龄），而 Γ_2 中现有绑定的年龄会乘以 $\uparrow$（也就是说，我们将年龄视为数字，并加 1）。这表示为 t' 在 $1\uparrow \cdot \Gamma_2, x :_{1\nu} \mathsf{T}$ 中的类型推导，而父项 $\mathsf{upd}_{\ltimes}\ t\ \mathsf{with}\ x \mapsto t'$ 则在未缩放的上下文 $\Gamma_1 + \Gamma_2$中定型。$\mathsf{upd}_{\ltimes}$ 引入的x（其中包含目的地）与 Γ_2 之间的年龄差异，使我们能够辨别哪些内容源自更早的作用域。具体而言，在对填充原语进行类型检查时，区分目的地的年龄至关重要，这样才能避免第 2.3 节中所述的陷阱。

图 2.4 展示了 $\mathsf{upd}_{\ltimes}$ 引入的作用域，以及 $\mathsf{upd}_{\ltimes}$ 与 ◀ 的类型规则如何相互作用。这里我们首次看到振幅对值 $/v^2 \wr v_1/$，它们表示为类似对的对象，左侧是带孔结构 v_2，右侧是包含目的地的任意结构 v_1（振幅对值将在第 2.6.1 节正式引入）。使用 $\mathsf{upd}_{\ltimes}$ 时，我们进入一个新作用域，在其中目的地

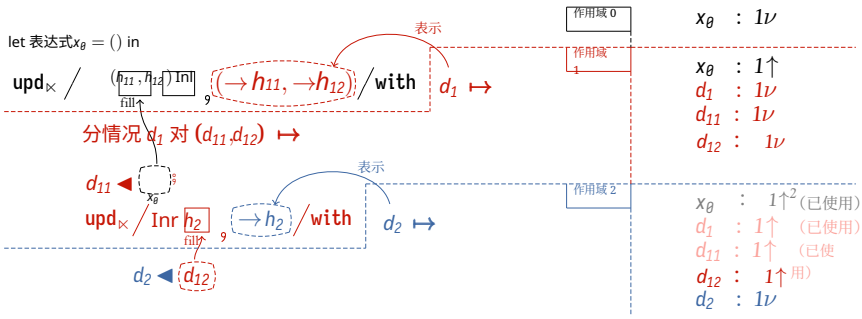


图 2.4：嵌套作用域中年龄的演化

可以访问，但带孔结构仍处于外层作用域。因此，例如，当用 λ_d–TY/$\lfloor\,\rfloor$E$_L$ 填充一个目的地时，例如 $d_{11} \blacktriangleleft x_\theta$（见图 2.4），我们在新作用域中为 d_{11} 定型，而在外层作用域中为 x_θ 定型，因为它正被移到振幅左侧、同样位于外层作用域的带孔结构中。这与 upd⋉ 所做的作用域变化恰恰相反：upd⋉ 会为其主体创建新作用域，而运算符 ◀，以及类似地，$\lhd\!\circ$ 和 $\lhd(\lambda x\,{}_m\!\mapsto u)$，会将其右操作数转移到外层作用域。换言之，◀ 或 $\lhd$ 的右侧是父作用域的一处飞地。

使用 from′⋉（规则 λ_d–ty$_s$/⋉from'）时，ampar 的左侧会被提取到当前作用域中。这正是 ampar 的左侧必须在当前作用域中“发生”的根本原因。我们知道该结构是完整的，并且可以被提取，这是因为右侧的类型是单位（1），因此，右侧没有目标意味着左侧不会残留空洞。from′⋉ 在代码清单 2.4 中是根据 from⋉ 实现的，以使核心演算更加简洁（并限制类型规则、求值上下文等的数量），但在实际实现中可以实现得高效得多。

当使用更一般的 from⋉ in 规则 λ_d–ty/⋉from 消去一个 ampar 时，我们会将 ampar 的两侧都提取到当前作用域中，即使右侧通常处于不同的作用域。这之所以安全，仅仅是因为要求右侧具有 $!_{1\infty}$T 类型，这意味着它是作用域无关的：其中不能包含任何受作用域控制的资源。此外，这还确保右侧不能包含目的地，因此该结构必须是完整结构，并已准备好读取。

另一方面，在 λ_d–ty/⋉to 中，无需处理作用域：运算符to⋉ 将一个已经完成的结构包装在一个平凡 ampar 中，其左侧是该结构（继续在当前作用域中进行类型检查），右侧则是单位。

其余运算符 $\lhd()$ $\lhd$Inl $\lhd$Inr $\lhd$Mod$_m$, $\lhd(,)$（来自 λ_d–ty/$\lfloor\,\rfloor$E 形式的规则）是其他目的地填充原语。它们会将一个空心构造器写入由目的地操作数指向的空洞，并返回为空心构造器中的新空洞所创建的潜在新目的地（如果没有新目的地，则返回单位）。

运算符 $\lhd\!\circ$ from 规则 λ_d–ty/$\lfloor\,\rfloor$E$_C$ 让我们用不完整结构填充一个目标。这是我们将带孔结构组合在一起的主要方式，让人想起 [Mi- namide, 1998] 中的 hcomp 运算符。与 ◀ 不同，运算符 $\lhd\!\circ$ 不会返回 1，而是返回用于填充空洞的不完整结构中剩余的目标。

就类型而言，$\lhd\!\circ$与其他目的地填充原语相比的一个特性是，它只接受类型为$\lfloor{}_{1\nu}$U$\rfloor$的左侧操作数——也就是说，只接受其对应空洞具有单位模态的目标。这一限制是有意为之的。假设我们有$d \lhd\!\circ (ampar ::: \mathrm{U} \rightarrow \lfloor{}_{1\nu}\mathrm{T}\rfloor)$，其中$d :::\lfloor{}_{\omega\nu}U\rfloor$，这意味着$d$指向模态$!_{\omega\nu}$之后的一个空洞。表达式$d \lhd\!\circ ampar$将返回类型为$\lfloor{}_{1\nu}T\rfloor$

（这样的类型签名表明关联的空洞具有 1_ν 模式），该空洞构成了*ampar*的右侧。然而，这个目标实际上指向一个结构中的空洞，而该结构现在已被移动到模态$!_{\omega\nu}$之后，因此只能接受模式为 $\omega\nu$ 的值。

使用 $\triangleleft_\circ$ 操作模态为 $\mathtt{m}$ 、与 1_ν 不同的目的地时，没有简单的方法可以即时更新这类目的地的模态注释。当右侧存储着外部目的地——即引用其他 ampar 中空洞的目的地——时，问题就更加棘手：另一方面，它们的模态绝不能被修改。为避免这些复杂情况并确保可靠性，我们限制 $\triangleleft_\circ$ 只能操作对应空洞的模式为 1_ν 的目的地。

付诸实践 作为练习，既然我们已经具备了所有类型规则，现在可以推导出合成目的地填充运算符 $\triangleleft(::)$ 以及对应的合成构造器 $(::)$，它们从第 2.2.2 节起便已使用。回顾一下，$\mathtt{List\ T} \triangleq 1 \oplus (\mathtt{T} \otimes (\mathtt{List\ T}))$。让我们先给出定义，然后推导 $\triangleleft(::)$的类型树：

$$\triangleleft(::) \ : \ \lfloor_{\mathtt{m}} \mathtt{List\ T} \rfloor \multimap (\lfloor_{\mathtt{m}} \mathtt{T} \rfloor \otimes \lfloor_{\mathtt{m}} \mathtt{List\ T} \rfloor)$$
$$d \triangleleft (::) \triangleq d \triangleleft \mathsf{Inr} \triangleleft (,)$$

A-类型：

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{1_\nu \leq 1_\nu}{d :_{1_\nu} \lfloor_{\mathtt{m}} \mathtt{List\ T} \rfloor \vdash d : \lfloor_{\mathtt{m}} \mathtt{List\ T} \rfloor}\,\mathrm{ID_V}}{d :_{1_\nu} \lfloor_{\mathtt{m}} \mathtt{List\ T} \rfloor \vdash d : \lfloor_{\mathtt{m}} 1 \oplus (\mathtt{T} \otimes (\mathtt{List\ T})) \rfloor}\ \text{unfold List T definition on the right}}{d :_{1_\nu} \lfloor_{\mathtt{m}} \mathtt{List\ T} \rfloor \vdash d \triangleleft \mathsf{Inr} : \lfloor_{\mathtt{m}} \mathtt{T} \otimes (\mathtt{List\ T}) \rfloor}\,\lfloor\oplus\rfloor \mathrm{E}_2}{d :_{1_\nu} \lfloor_{\mathtt{m}} \mathtt{List\ T} \rfloor \vdash d \triangleleft \mathsf{Inr} \triangleleft (,) : \lfloor_{\mathtt{m}} \mathtt{T} \rfloor \otimes \lfloor_{\mathtt{m}} \mathtt{List\ T} \rfloor}\,\lfloor\otimes\rfloor \mathrm{E}}{\cdot \vdash \lambda d \mapsto d \triangleleft \mathsf{Inr} \triangleleft (,) : \lfloor_{\mathtt{m}} \mathtt{List\ T} \rfloor \multimap \lfloor_{\mathtt{m}} \mathtt{T} \rfloor \otimes \lfloor_{\mathtt{m}} \mathtt{List\ T} \rfloor}\,\lambda_d\text{-}\mathrm{TY_S}/\multimap \mathrm{I}$$

现在，让我们继续看 **(::)** 合成构造器：

$$(::) ::: \mathtt{T} \multimap \mathtt{T\ 列表} \multimap \mathtt{T\ 列表}$$
$$x :: xs \triangleq \mathtt{from}'_{\ltimes}(\mathtt{upd}_{\ltimes}\ \mathtt{新建}_{\ltimes}\ \mathtt{with}\ d \mapsto \mathtt{分情况}\ (d \triangleleft (::))\ \mathtt{of}\ (dx, dxs) \mapsto dx \blacktriangleleft x \fatsemi dxs \blacktriangleleft xs)$$

鉴于类型树会变得非常庞大，我们将先为子表达式$B \triangleq \mathtt{分情况}\ (d \triangleleft (::))\ \mathtt{of}\ (dx, dxs) \mapsto dx \blacktriangleleft x \fatsemi dxs \blacktriangleleft xs$进行类型推导，然后为对应于运算符 **(::)** 的整个表达式进行类型推导。

B-TY：

$$\dfrac{\dfrac{1_\nu \leq 1_\nu}{d :_{1_\nu} \lfloor \mathtt{T\ 列表} \rfloor \vdash d ::: \lfloor \mathtt{T\ 列表} \rfloor}\,\mathrm{ID_V}}{d :_{1_\nu} \lfloor \mathtt{T\ 列表} \rfloor \vdash d \triangleleft (::) ::: \lfloor \mathtt{T} \rfloor \otimes \lfloor \mathtt{T\ 列表} \rfloor}\,A\text{-}\mathrm{TY} \quad \dfrac{\dfrac{1_\nu \leq 1_\nu}{dx :_{1_\nu} \lfloor \mathtt{T} \rfloor \vdash dx ::: \lfloor \mathtt{T} \rfloor}\,\mathrm{ID_V} \quad \dfrac{1_\nu \leq 1_\nu}{x :_{1_\nu} \mathtt{T} \vdash x : \mathtt{T}}\,\mathrm{ID_V}}{dx :_{1_\nu} \lfloor \mathtt{T} \rfloor,\ 1^\uparrow \cdot x :_{1_\nu} \mathtt{T} \vdash dx \blacktriangleleft x ::: 1}\,\lfloor\ \mathrm{E_L} \quad \dfrac{\dfrac{1_\nu \leq 1_\nu}{dxs :_{1_\nu} \lfloor \mathtt{T\ 列表} \rfloor \vdash dxs ::: \lfloor \mathtt{T\ 列表} \rfloor}\,\mathrm{ID_V} \quad \dfrac{1_\nu \leq 1_\nu}{xs :_{1_\nu} \mathtt{T\ 列表} \vdash xs ::: \mathtt{T\ 列表}}\,\mathrm{ID_V}}{dxs :_{1_\nu} \lfloor \mathtt{T\ 列表} \rfloor,\ 1^\uparrow \cdot xs :_{1_\nu} \mathtt{T\ 列表} \vdash dxs \blacktriangleleft xs ::: 1}\,\lfloor\ \mathrm{E_L}$$
$$\dfrac{dx :_{1_\nu} \lfloor \mathtt{T} \rfloor,\ dxs :_{1_\nu} \lfloor \mathtt{T\ 列表} \rfloor,\ 1^\uparrow \cdot (x :_{1_\nu} \mathtt{T},\ xs :_{1_\nu} \mathtt{T\ 列表}) \vdash dx \blacktriangleleft x \fatsemi dxs \blacktriangleleft xs ::: 1}{d :_{1_\nu} \lfloor \mathtt{T\ 列表} \rfloor,\ 1^\uparrow \cdot (x :_{1_\nu} \mathtt{T},\ xs :_{1_\nu} \mathtt{T\ 列表}) \vdash \mathtt{分情况}\ (d \triangleleft (::))\ \mathtt{of}\ (dx,\ dxs) \mapsto dx \blacktriangleleft x \fatsemi dxs \blacktriangleleft xs ::: 1}\,\otimes \mathrm{E}$$

C语言，

TY：

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot \vdash \mathtt{新建}_{\ltimes} ::: \mathtt{T\ 列表} \ltimes \lfloor \mathtt{T\ 列表} \rfloor}\,\ltimes \mathrm{NEW} \quad \dfrac{}{d :_{1_\nu} \lfloor \mathtt{T\ 列表} \rfloor,\ 1^\uparrow \cdot (x :_{1_\nu} \mathtt{T},\ xs :_{1_\nu} \mathtt{T\ 列表}) \vdash B ::: 1}\,B\text{-}\mathrm{TY}}{x :_{1_\nu} \mathtt{T},\ xs :_{1_\nu} \mathtt{T\ 列表} \vdash \mathtt{upd}_{\ltimes}\ \mathtt{新建}_{\ltimes}\ \mathtt{with}\ d \mapsto B ::: \mathtt{T\ 列表} \ltimes 1}\,\ltimes \mathrm{UPD}}{x :_{1_\nu} \mathtt{T},\ xs :_{1_\nu} \mathtt{T\ 列表} \vdash \mathtt{from}'_{\ltimes}(\mathtt{upd}_{\ltimes} \mathtt{新建}_{\ltimes}\ \mathtt{with}\ d \mapsto B) ::: \mathtt{T\ 列表}}\,\ltimes \mathrm{from}'}{x :_{1_\nu} \mathtt{T} \vdash \lambda xs \mapsto \mathtt{from}'_{\ltimes}(\mathtt{upd}_{\ltimes} \mathtt{新建}_{\ltimes}\ \mathtt{with}\ d \mapsto B) ::: \mathtt{T\ 列表} \multimap \mathtt{T\ 列表}}\,\lambda_d\text{-}\mathrm{ty_S}/\multimap \mathrm{I}$$
$$\cdot \vdash \lambda x \mapsto \lambda xs \mapsto \mathtt{from}'_{\ltimes}(\mathtt{upd}_{\ltimes}\ \mathtt{新建}_{\ltimes}\ \mathtt{with}\ d \mapsto B) ::: \mathtt{T} \multimap \mathtt{T\ 列表} \multimap \mathtt{T\ 列表}\,\lambda_d\text{-}\mathrm{ty_S}/\multimap \mathrm{I}$$

如果我们从根节点向叶节点观察类型树 C语言-TY，会发现进入 upd$_{\ltimes}$ 时，x 和 xs 的变量绑定都会乘以 1↑（它们的年龄增加一岁）。但这并不是问题；事实上，类型规则 λ_d-TY/⌊⌋EL 要求右操作数的年龄为 ↑（一如既往，这是为了防止作用域逃逸），正如我们在类型树 B-TY中所见。

此外，与常规函数式语言一样，模态，更具体地说，年龄，不会 出现在合成构造器 (::) 的类型签名中；它们只出现在其内部类型推导中，因为从这个角度来看，使用目标来定义 (::) 只是对用户隐藏的实现细节。

2.6 操作语义

我们以 λ_d 第 1 章中使用的归约语义风格给出其操作语义，也就是说，显式地操作求值上下文的语法。不过这一次我们会做得更全面。具体而言，我们将正式引入求值上下文的语法，并定义求值上下文和命令的类型规则，以证明归约过程中类型得以保持。与之前一样，命令$E[t]$用于表示运行中的程序；第 2.6.2 节将对此作详细说明。

不过首先，我们需要一类运行时值（通常简称为值），以及相应的类型规则，因为 λ_d 目前还无法表示目标或空洞，实际上也无法表示任何类型的值（例如，Inl() 到目前为止只是项from′$_{\ltimes}$ (upd$_{\ltimes}$ 新建$_{\ltimes}$ with d ↦ ...) 的语法糖）。λ_d 的一个特点是，值（特别是数据构造器）只在归约过程中使用；而通常在 λ_{Ldm} 这样的函数式语言中，值也允许出现在项语法中。

2.6.1 运行时值与新建类型上下文形式

运行时值的语法如图 2.5 所示。它包含了所有基本类型的构造器，以及一种原始函数形式 $\boldsymbol{\lambda}_{\mathbf{v}}x\ {}_{m}\!\mapsto u$（其限制多于语法糖形式 $\boldsymbol{\lambda}x\ {}_{m}\!\mapsto u$；例如，它不能包含自由变量）。较有趣的值包括空洞 h、目标 $\to h$和 ampar $_{H}/v_2,v_1/$，这些在 λ_{Ldm} 中并未出现，本节余下部分将对此进行说明。为了让操作语义能够使用替换（这要求用值替换变量），我们还将值纳入项的语法，并添加相应的类型规则 λ_d-TY/FROMVAL（见图 2.6 顶部）。

目标和空洞如第 2.2.1 节所示，是同一枚硬币的两面；我们必须确保在整个归约过程中，目标始终指向一个空洞，而每个空洞也始终恰好是一个目标的指向对象。因此，我们系统的新想法是，在类型上下文中引入孔绑定 $h :_{n} T$ 和目标绑定 $\to h :_{m} \lfloor_{n} T \rfloor$，此外还保留常见的变量绑定$x :_{m} T$。在这两种情况下，我们都称$h$ 为孔名。此外，我们还引入新的类型上下文形式：现在有 Γ、Θ 和 Δ。Γ 是为项进行类型判断时使用的类型上下文形式；与以往一样，其中可以包含变量绑定，但现在也允许包含目标绑定。另一方面，新的上下文形式 Θ 可以同时包含目标绑定和孔绑定，但不能同时包含同一孔名对应的目标绑定和孔绑定。Θ 也不能包含变量绑定。最后，类型上下文 Δ 只能包含目标绑定。

我们将之前的上下文操作 + 和 · 扩展为作用于新建绑定形式，如图 2.5 所示。上下文加法仍然是部分定义的；例如，$(h :_{n} T) + (\to h :_{m} \lfloor_{n'} T \rfloor)$ 未定义，因为 h 同时出现在两侧，但采用了不同的绑定形式。

扩展了值的语法：

$$t, u \;::=\; \ldots \;\mid\; v$$

$$\begin{array}{rll} v \;::=\; \boxed{h} & & \textit{(hole)} \\ \mid\; \to h & & \textit{(destination)} \\ \mid\; {}_H\!/v_2\,,v_1/ & & \textit{(ampar value)} \\ \mid\; () \;\mid\; \mathsf{Inl}\, v \;\mid\; \mathsf{Inr}\, v \;\mid\; \mathsf{Mod}_{\mathsf{m}}\, v \;\mid\; (v_1, v_2) \;\mid\; \boldsymbol{\lambda}_{\mathsf{v}} x :_{\mathsf{m}} \mathsf{T} \mapsto u & & \end{array}$$

类型上下文的扩展语法：

$$\begin{array}{rcl} \Delta & ::= & \bullet \mid \to h :_{\mathsf{m}} \lfloor {}_{\mathsf{n}}\mathsf{T} \rfloor \mid \Delta_1, \Delta_2 \\ \Gamma & ::= & \bullet \mid \to h :_{\mathsf{m}} \lfloor {}_{\mathsf{n}}\mathsf{T} \rfloor \mid x :_{\mathsf{m}} \mathsf{T} \mid \Gamma_1, \Gamma_2 \\ \Theta & ::= & \bullet \mid \to h :_{\mathsf{m}} \lfloor {}_{\mathsf{n}}\mathsf{T} \rfloor \mid \boxed{h} :_{\mathsf{n}} \mathsf{T} \mid \Theta_1, \Theta_2 \end{array}$$

对新建类型上下文形式的操作：

$$\begin{array}{rcll} \mathsf{n}' \cdot (\boxed{h} :_{\mathsf{n}} \mathsf{T}, \Theta) & \triangleq & (\boxed{h} :_{\mathsf{n}'\cdot\mathsf{n}} \mathsf{T}), \mathsf{n}' \cdot \Theta & \\ \mathsf{n}' \cdot (\to h :_{\mathsf{m}} \lfloor {}_{\mathsf{n}}\mathsf{T} \rfloor, \Gamma) & \triangleq & (\to h :_{\mathsf{n}'\cdot\mathsf{m}} \lfloor {}_{\mathsf{n}}\mathsf{T} \rfloor), \mathsf{n}' \cdot \Gamma & \dagger \\ & & & \\ (\boxed{h} :_{\mathsf{n}} \mathsf{T}, \Theta_1) + \Theta_2 & \triangleq & \boxed{h} :_{\mathsf{n}} \mathsf{T}, (\Theta_1 + \Theta_2) & \text{if } h \notin \Theta_2 \\ (\boxed{h} :_{\mathsf{n}} \mathsf{T}, \Theta_1) + (\boxed{h} :_{\mathsf{n}'} \mathsf{T}, \Theta_2) & \triangleq & \boxed{h} :_{\mathsf{n}+\mathsf{n}'} \mathsf{T}, (\Theta_1 + \Theta_2) & \\ (\to h :_{\mathsf{m}} \lfloor {}_{\mathsf{n}}\mathsf{T} \rfloor, \Gamma_1) + \Gamma_2 & \triangleq & \to h :_{\mathsf{m}} \lfloor {}_{\mathsf{n}}\mathsf{T} \rfloor, (\Gamma_1 + \Gamma_2) & \text{if } h \notin \Gamma_2 \quad \dagger \\ (\to h :_{\mathsf{m}} \lfloor {}_{\mathsf{n}}\mathsf{T} \rfloor, \Gamma_1) + (\to h :_{\mathsf{m}'} \lfloor {}_{\mathsf{n}}\mathsf{T} \rfloor, \Gamma_2) & \triangleq & \to h :_{\mathsf{m}+\mathsf{m}'} \lfloor {}_{\mathsf{n}}\mathsf{T} \rfloor, (\Gamma_1 + \Gamma_2) & \dagger \end{array}$$

$$\begin{array}{rcl} \to^{-1}(\bullet) & \triangleq & \bullet \\ \to^{-1}(\to h :_{1\mathsf{v}} \lfloor {}_{\mathsf{n}}\mathsf{T} \rfloor, \Delta) & \triangleq & (\boxed{h} :_{\mathsf{n}} \mathsf{T}), \to^{-1}(\Delta) \end{array}$$

$\dagger$: *same rule is also true for* Θ *or* Δ *replacing* Γ

图 2.5：运行时值、新建类型上下文形式和运算符

将值作为项进行类型判断：

$$\frac{\Delta \vdash_{\mathsf{v}} v : \mathsf{T}}{\omega\nu \cdot \Gamma,\ \Delta \vdash v : \mathsf{T}}\ \lambda_d\text{–TY/FROMVAL}$$

$\boxed{\Theta \vdash_{\mathsf{v}} v : \mathsf{T}}$ （值的类型判断）

$$\frac{}{h :_{1\nu} \mathsf{T} \vdash_{\mathsf{v}} h : \mathsf{T}}\ \text{ID}_\text{H} \qquad \frac{1\nu \leq \mathsf{m}}{\to h :_{\mathsf{m}} \lfloor_{\mathsf{n}} \mathsf{T} \rfloor \vdash_{\mathsf{v}} \to h : \lfloor_{\mathsf{n}} \mathsf{T} \rfloor}\ \text{ID}_\text{D} \qquad \frac{}{\cdot \vdash_{\mathsf{v}} () : \mathsf{1}}\ \mathsf{1I}$$

$$\frac{\Delta,\ x :_{\mathsf{m}} \mathsf{T} \vdash u : \mathsf{U}}{\Delta \vdash_{\mathsf{v}} \boldsymbol{\lambda}_{\mathsf{v}} x \,{}_{\mathsf{m}}\!\mapsto u : \mathsf{T} \,{}_{\mathsf{m}}\!\multimap \mathsf{U}}\ \multimap\text{I} \qquad \frac{\Theta \vdash_{\mathsf{v}} v_1 : \mathsf{T}_1}{\Theta \vdash_{\mathsf{v}} \mathsf{Inl}\, v_1 : \mathsf{T}_1 \oplus \mathsf{T}_2}\ \oplus\text{I}_1 \qquad \frac{\Theta \vdash_{\mathsf{v}} v_2 : \mathsf{T}_2}{\Theta \vdash_{\mathsf{v}} \mathsf{Inr}\, v_2 : \mathsf{T}_1 \oplus \mathsf{T}_2}\ \oplus\text{I}_2$$

$$\frac{\Theta_1 \vdash_{\mathsf{v}} v_1 : \mathsf{T}_1 \quad \Theta_2 \vdash_{\mathsf{v}} v_2 : \mathsf{T}_2}{\Theta_1 + \Theta_2 \vdash_{\mathsf{v}} (v_1 , v_2) : \mathsf{T}_1 \otimes \mathsf{T}_2}\ \otimes\text{I} \qquad \frac{\Theta \vdash_{\mathsf{v}} v' : \mathsf{T}}{\mathsf{n} \cdot \Theta \vdash_{\mathsf{v}} \mathsf{Mod}_{\mathsf{n}}\, v' : !_{\mathsf{n}} \mathsf{T}}\ !\text{I} \qquad \frac{1\uparrow \cdot \Delta_1,\ \Delta_3 \vdash_{\mathsf{v}} v_1 : \mathsf{T} \quad \Delta_2,\ \to^{-1}\Delta_3 \vdash_{\mathsf{v}} v_2 : \mathsf{U}}{\Delta_1,\ \Delta_2 \vdash_{\mathsf{v}} {}_{\mathsf{hnames}(\Delta_3)}\!/v_2\ ,v_1/ : \mathsf{U} \ltimes \mathsf{T}}\ \ltimes\text{I}$$

图 2.6：运行时值的类型规则（λ_d–ty$_\text{v}$）

其主要目标之一是确保 λ_d 永远不会读取孔值。在语法层面，我们不区分带有空洞的值和完全定义的值：相反，类型会阻止空洞被读取。类型系统通过在对项进行类型判断时不允许上下文中存在任何孔绑定，来维护这一不变式（事实上，图 2.2 中的类型规则 λ_d–TY 仍然成立，例如，类型上下文必须具有 Γ 的形状）。实际上，引入空洞的唯一位置是在图 2.6 的值的类型规则中，更确切地说，是规则 v_2 ampar ${}_{H}\!/v_2\ ,v_1/$ 中的左侧 λ_d–TYv/$\ltimes$I。

具体而言，空洞来自运算符 $\to^{-1}$，该运算符表示与指定目标绑定集合匹配的孔绑定。正如图 2.5 所定义的，它是类型上下文 Δ 上的一种偏逐点运算。请注意，如果 Δ 中的任何目标绑定具有不同于 1ν 的模式，则 $\to^{-1}\Delta$ 未定义（从而使该运算成为双射）。

此外，在 λ_d–TYv/$\ltimes$I 中，空洞 $\to^{-1}\Delta_3$ 以及对应的目标 Δ_3 会在 ampar 中绑定在一起，因此会从 ampar 的类型上下文中移除：这样我们就能确保每个空洞恰有一个目标，且每个目标恰有一个空洞。我们用下标 H 标注 ampar ${}_{H}\!/v_2\ ,v_1/$ 所绑定的空洞名称集合。这就是为什么在规则 λ_d–TYv/$\ltimes$I 中，结果 ampar 携带空洞名称集合 $\mathsf{hnames}(\Delta_3)$（我们使用运算符 $\mathsf{hnames}(\bullet)$ 表示给定类型上下文中存在的空洞名称）。

话虽如此，ampar 也可以存储并非自身所属、而是来自其他作用域的目标；它们分别由 $1\uparrow \cdot \Delta_1$ 和 Δ_2 表示在 v_1 和 v_2 的 λ_d–TYv/$\ltimes$I 类型上下文中。

规则 λ_d–TYv/IDH表示空洞在类型上下文中的模式必须为 1ν，才能保证类型正确；特别是这里不允许模态强制，也不允许弱化。只有当空洞位于指数模态之后时，该模式才可以变为任意模式 n。空洞的模式会限制可以写入其中的值；例如，在 h:nT ⊢v Modn h ::: !nT中，只有模式为 n(更准确地说，是在形如 n·Θ 的上下文中具有类型的值）才能写入 h。

令人惊讶的是，在 λ_d–tyv/idD 中，我们看到目的地可以具有任意一种能够强制转换为 1ν 的模式 m 。我们这样做是为了模拟规则 λ_d–ty/idV，并使得可以针对 λ_d[18] 表述一般的模态替换引理。然而，我们正式证明了，在良类型闭程序的整个归约过程中，m 永远不会具有重数 ω 或年龄 ∞——目的地始终是线性的，且具有有限年龄——因此实际上从未使用模态强制；我们还利用这一点，大大简化了替换引理的形式化证明。出现在 λ_d–tyv/idD 中的另一种模式 n 并非目的地绑定的模式，而是类型 ⌊nT⌋ 的一部分，并对应于我们可以写入相应 h 的值的模式；因此，对它无法进行强制转换。

其他重要之处 虽然值是在允许同时包含目的地绑定和孔绑定的上下文 Θ 中进行类型判断的，但将值用作 λ_d–ty/fromVal 中的项时，只允许它具有自由目的地，不能有自由洞（因为我们要求该值的类型上下文具有 Δ 的形式）。

还要注意，值（无论是否用作项）都不能有自由变量，因为上下文 Θ 只包含孔绑定和目的地绑定，不包含变量绑定。值是封闭的，这是指称语义或抽象机器语义的标准特征。即使是函数值（λ_d–tyv/⊸I）也是如此，它们同样不能包含自由洞。由于函数体尚未求值，因此很难说函数包含洞究竟意味着什么；至少，这会让我们的系统复杂许多，而且我们不知道允许函数中存在自由洞能带来什么好处。

有人或许会想知道，如何在值层级表示柯里化函数 $\boldsymbol{\lambda} x \mapsto \boldsymbol{\lambda} y \mapsto x \,\mathsf{concat}\, y$，因为内部抽象捕获了自由变量$x$。答案是，这样的函数在值层级被编码为$\boldsymbol{\lambda}_{\mathsf{v}} x \mapsto \mathsf{from}'_{\ltimes}(\mathsf{upd}_{\ltimes}\ \mathsf{new}_{\ltimes}\ \mathsf{with}\ d \mapsto d \triangleleft (\boldsymbol{\lambda} y \mapsto x \,\mathsf{concat}\, y))$，其中内部闭包尚未处于值形式。由于$d \triangleleft (\boldsymbol{\lambda} y \mapsto x \,\mathsf{concat}\, y)$属于项语法，因此允许它含有自由变量$x$。

2.6.2 求值上下文与命令

运行中的程序由一个命令 $E[t]$表示，也就是说，它是一个求值上下文E与一个处于焦点下的（扩展）项t 所组成的对，如第 1.6 节所示。

求值上下文的语法见图 2.7。与 λ_{Ldm}类似，求值上下文E 是任意数量的聚焦组件e的组合。乍看之下，我们的语义中不需要存储或状态的概念来表示填充目标所引发的变异，这似乎令人意外。事实上，目标填充只需要一种非常简单的状态概念——简单到我们只需通过对求值上下文进行替换，就能表示向空洞写入，而不必采用更复杂的存储语义。

然而，需要注意的是，与 λ_{Ldm}不同，命令$E[t]$ 的 λ_d 并不总是有对应的项。原因在于，除了开放 *ampar* 聚焦组件操作H /v_2 ,[ ]/ 之外，所有聚焦组件都直接源自项语法；后者没有对应的项构造。这个聚焦组件表示 ampar 当前正由upd⋉处理，其左侧v_2 （正在构建的结构）附加到开放 ampar 聚焦组件上，

[18]通常，在模态系统中，若 x :mT Γ ⊢u ::: U 且 Δ ⊢v ::: T，则 m ·Δ, Γ ⊢u[x :=v] ::: U（尤其参见 ⟦Abel 和 Bernardy，2020⟧）。在 λ_d中，我们有 x :ω∞ ⌊nT⌋ ⊢() ::: 1 且 →h :1ν ⌊nT⌋ ⊢ →h ::: ⌊nT⌋，因此 ω∞ ·(→h :1ν ⌊nT⌋) ⊢()[x := →h] ::: 1 应当成立。

$$e ::= t'\,[\,] \mid [\,]\,v \mid [\,]\,;u \mid \text{分情况}_m\,[\,]\,\text{of}\,\{\text{Inl}\,x_1 \mapsto u_1 \mid \text{Inr}\,x_2 \mapsto u_2\} \mid \text{分情况}_m\,[\,]\,\text{of}\,(x_1, x_2) \mapsto u \mid \text{分情况}_n\,[\,]\,\text{of}\,\text{Mod}_n\,x \mapsto u \mid \text{upd}_{\ltimes}\,[\,]\,\text{with}\,x \mapsto t' \mid \text{to}_{\ltimes}\,[\,] \mid \text{from}_{\ltimes}\,[\,] \mid [\,] \triangleleft_{\text{o}} t' \mid v \triangleleft_{\text{o}} [\,] \mid [\,] \blacktriangleleft t' \mid v \blacktriangleleft [\,] \mid [\,] \triangleleft () \mid [\,] \triangleleft \text{Inl} \mid [\,] \triangleleft \text{Inr} \mid [\,] \triangleleft (,) \mid [\,] \triangleleft \text{Mod}_m \mid [\,] \triangleleft (\lambda x \mapsto u) \mid \text{操作}\ H\ /v_2\ ,[\,]/\ (\text{开放 ampar 聚焦组件})\ E ::= [\,] \mid E \circ e$$

图 2.7：求值上下文的语法

其右侧（包含目的地）要么位于后续的聚焦组件中，要么位于焦点项中。ampar 在求值 $\text{upd}_{\ltimes}$ 的主体期间保持开放，之后再关闭，这与类型规则中的作用域概念相对应。

求值上下文的类型判定求值上下文在一个只能包含目的地绑定的上下文 Δ 中进行类型判定。求值上下文 E 的类型上下文 Δ，恰好是项 t 为构成有效的 $E[t]$ 所必须使用的类型上下文。换言之，$\Gamma \vdash t ::: \mathsf{T}$ 要求 Γ 的绑定，而判断式 $\Delta \dashv E ::: \mathsf{T} \rightarrowtail \mathsf{U}_0$ 提供 Δ 的绑定。求值上下文的类型规则以及命令的唯一一条类型规则见图 2.8。

求值上下文具有上下文类型 $\mathsf{T} \rightarrowtail \mathsf{U}_0$。$E ::: \mathsf{T} \rightarrowtail \mathsf{U}_0$ 的含义是，给定 $t ::: \mathsf{T}$，$E[t]$ 会返回一个类型为 U_0 的值。将求值上下文 $E ::: \mathsf{T} \rightarrowtail \mathsf{U}_0$ 与一个新建的聚焦组件组合，绝不会影响未来命令的类型 U_0；改变的只有焦点的类型 T。

求值上下文的所有类型规则都可以根据相应项的类型规则系统地推导出来（除了规则 λ_d–$\text{ty}_{\text{E}}/\ltimes\text{op}$，因为不存在与 *openampar* 聚焦组件直接对应的项）。以规则 λ_d–$\text{ty}_{\text{E}}/\otimes\text{E}$ 为例：

$$\frac{\mathsf{m} \cdot \Delta_1,\ \Delta_2 \dashv E : \mathsf{U} \rightarrowtail \mathsf{U}_0 \qquad \Delta_2,\ x_1 :_{\mathsf{m}} \mathsf{T}_1,\ x_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash u : \mathsf{U}}{\Delta_1 \dashv E \circ \mathsf{case}_{\mathsf{m}}\,[\,]\,\mathsf{of}\,(x_1, x_2) \mapsto u : (\mathsf{T}_1 \otimes \mathsf{T}_2) \rightarrowtail \mathsf{U}_0}\ \lambda_d\text{–ty}_{\text{E}}/\otimes\text{E} \qquad \frac{\Gamma_1 \vdash t : \mathsf{T}_1 \otimes \mathsf{T}_2 \qquad \Gamma_2,\ x_1 :_{\mathsf{m}} \mathsf{T}_1,\ x_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash u : \mathsf{U}}{\mathsf{m} \cdot \Gamma_1 + \Gamma_2 \vdash \mathsf{case}_{\mathsf{m}}\,t\,\mathsf{of}\,(x_1, x_2) \mapsto u : \mathsf{U}}\ \lambda_d\text{–ty}/\otimes\text{E}$$

- 在 λ_d–$\text{ty}_{\text{E}}/\otimes\text{E}$ 中，E 前提里的类型上下文 $\mathsf{m} \cdot \Delta_1,\ \Delta_2$，对应于 λ_d–$\text{ty}/\otimes\text{E}$ 结论中的 $\mathsf{m} \cdot \Gamma_1 + \Gamma_2$；
- 在 λ_d–$\text{ty}_{\text{E}}/\otimes\text{E}$ 中，项 u 前提里的类型上下文 $\Delta_2, x_1 :_{\mathsf{m}} \mathsf{T}_1, x_2 :_{\mathsf{m}} \mathsf{T}_2$，对应于 λ_d–$\text{ty}/\otimes\text{E}$ 中同一项的类型上下文 $\Gamma_2, x_1 :_{\mathsf{m}} \mathsf{T}_1, x_2 :_{\mathsf{m}} \mathsf{T}_2$；
- 在 λ_d–$\text{ty}_{\text{E}}/\otimes\text{E}$ 的结论中，$E \circ \mathsf{case}_{\mathsf{m}}\,[\,]\,\mathsf{of}\,(x_1, x_2) \mapsto u$ 对应的类型上下文 Δ_1，与 λ_d–$\text{ty}/\otimes\text{E}$ 中项 t 前提里的类型上下文相对应（项 t 位于 λ_d–$\text{ty}_{\text{E}}/\otimes\text{E}$ 中焦点 $[\,]$ 所在的位置）。

我们将求值上下文的类型规则视为相关项的类型规则的一种旋转：交换一个子项与结论的类型上下文，而其他可能子项的类型上下文保持不变（不同之处在于，求值上下文的类型上下文形式为 Δ，而不是 Γ）。

$\boxed{\Delta \dashv E : \mathsf{T} \rightarrowtail \mathsf{U}_0}$ *(Typing judgment for evaluation contexts)*

$$\frac{}{\bullet \dashv [\,] : \mathsf{U}_0 \rightarrowtail \mathsf{U}_0}\ \mathrm{ID} \qquad \frac{\mathsf{m}\cdot\Delta_1,\ \Delta_2 \dashv E : \mathsf{U} \rightarrowtail \mathsf{U}_0 \quad \Delta_2 \vdash t' : \mathsf{T}_{\mathsf{m}} \multimap \mathsf{U}}{\Delta_1 \dashv E \circ t'\,[\,] : \mathsf{T} \rightarrowtail \mathsf{U}_0}\ \multimap\mathrm{E}_1 \qquad \frac{\mathsf{m}\cdot\Delta_1,\ \Delta_2 \dashv E : \mathsf{U} \rightarrowtail \mathsf{U}_0 \quad \Delta_1 \vdash v : \mathsf{T}}{\Delta_2 \dashv E \circ [\,]\,v : (\mathsf{T}_{\mathsf{m}} \multimap \mathsf{U}) \rightarrowtail \mathsf{U}_0}\ \multimap\mathrm{E}_2$$

$$\frac{\Delta_1,\ \Delta_2 \dashv E : \mathsf{U} \rightarrowtail \mathsf{U}_0 \quad \Delta_2 \vdash u : \mathsf{U}}{\Delta_1 \dashv E \circ [\,] \fatsemi u : \mathsf{1} \rightarrowtail \mathsf{U}_0}\ 1\mathrm{E} \qquad \frac{\mathsf{m}\cdot\Delta_1,\ \Delta_2 \dashv E : \mathsf{U} \rightarrowtail \mathsf{U}_0 \quad \Delta_2,\ x_1 :_{\mathsf{m}} \mathsf{T}_1 \vdash u_1 : \mathsf{U} \quad \Delta_2,\ x_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash u_2 : \mathsf{U}}{\Delta_1 \dashv E \circ \mathsf{case}_{\mathsf{m}}\,[\,]\ \mathsf{of}\ \{\mathsf{Inl}\,x_1 \mapsto u_1\,,\ \mathsf{Inr}\,x_2 \mapsto u_2\} : (\mathsf{T}_1 \oplus \mathsf{T}_2) \rightarrowtail \mathsf{U}_0}\ \oplus\mathrm{E}$$

$$\frac{\mathsf{m}\cdot\Delta_1,\ \Delta_2 \dashv E : \mathsf{U} \rightarrowtail \mathsf{U}_0 \quad \Delta_2,\ x_1 :_{\mathsf{m}} \mathsf{T}_1,\ x_2 :_{\mathsf{m}} \mathsf{T}_2 \vdash u : \mathsf{U}}{\Delta_1 \dashv E \circ \mathsf{case}_{\mathsf{m}}\,[\,]\ \mathsf{of}\ (x_1\,,\,x_2) \mapsto u : (\mathsf{T}_1 \otimes \mathsf{T}_2) \rightarrowtail \mathsf{U}_0}\ \otimes\mathrm{E} \qquad \frac{\mathsf{m}\cdot\Delta_1,\ \Delta_2 \dashv E : \mathsf{U} \rightarrowtail \mathsf{U}_0 \quad \Delta_2,\ x :_{\mathsf{m}\cdot\mathsf{m}'} \mathsf{T} \vdash u : \mathsf{U}}{\Delta_1 \dashv E \circ \mathsf{case}_{\mathsf{m}}\,[\,]\ \mathsf{of}\ \mathsf{Mod}_{\mathsf{m}'}\,x \mapsto u : !_{\mathsf{m}'}\mathsf{T} \rightarrowtail \mathsf{U}_0}\ !\mathrm{E}$$

$$\frac{\Delta_1,\ \Delta_2 \dashv E : \mathsf{U} \ltimes \mathsf{T}' \rightarrowtail \mathsf{U}_0 \quad 1\!\uparrow\cdot\Delta_2,\ x :_{1_\nu} \mathsf{T} \vdash t' : \mathsf{T}'}{\Delta_1 \dashv E \circ \mathsf{upd}_{\ltimes}\,[\,]\ \mathsf{with}\ x \mapsto t' : (\mathsf{U} \ltimes \mathsf{T}) \rightarrowtail \mathsf{U}_0}\ \ltimes\mathrm{UPD} \qquad \frac{\Delta \dashv E : (\mathsf{U} \ltimes \mathsf{1}) \rightarrowtail \mathsf{U}_0}{\Delta \dashv E \circ \mathsf{to}_{\ltimes}\,[\,] : \mathsf{U} \rightarrowtail \mathsf{U}_0}\ \ltimes\mathrm{TO}$$

$$\frac{\Delta \dashv E : (\mathsf{U} \otimes (!_{1_\infty}\mathsf{T})) \rightarrowtail \mathsf{U}_0}{\Delta \dashv E \circ \mathsf{from}_{\ltimes}\,[\,] : (\mathsf{U} \ltimes (!_{1_\infty}\mathsf{T})) \rightarrowtail \mathsf{U}_0}\ \ltimes\mathrm{FROM} \qquad \frac{\Delta \dashv E : \mathsf{1} \rightarrowtail \mathsf{U}_0}{\Delta \dashv E \circ [\,] \lhd () : \lfloor_{\mathsf{n}}\mathsf{1}\rfloor \rightarrowtail \mathsf{U}_0}\ \lfloor 1\rfloor\mathrm{E}$$

$$\frac{\Delta \dashv E : \lfloor_{\mathsf{n}}\mathsf{T}_1\rfloor \rightarrowtail \mathsf{U}_0}{\Delta \dashv E \circ [\,] \lhd \mathsf{Inl} : \lfloor_{\mathsf{n}}\mathsf{T}_1 \oplus \mathsf{T}_2\rfloor \rightarrowtail \mathsf{U}_0}\ \lfloor\oplus\rfloor\mathrm{E}_1 \qquad \frac{\Delta \dashv E : \lfloor_{\mathsf{n}}\mathsf{T}_2\rfloor \rightarrowtail \mathsf{U}_0}{\Delta \dashv E \circ [\,] \lhd \mathsf{Inr} : \lfloor_{\mathsf{n}}\mathsf{T}_1 \oplus \mathsf{T}_2\rfloor \rightarrowtail \mathsf{U}_0}\ \lfloor\oplus\rfloor\mathrm{E}_2$$

$$\frac{\Delta \dashv E : (\lfloor_{\mathsf{n}}\mathsf{T}_1\rfloor \otimes \lfloor_{\mathsf{n}}\mathsf{T}_2\rfloor) \rightarrowtail \mathsf{U}_0}{\Delta \dashv E \circ [\,] \lhd (,) : \lfloor_{\mathsf{n}}\mathsf{T}_1 \otimes \mathsf{T}_2\rfloor \rightarrowtail \mathsf{U}_0}\ \lfloor\otimes\rfloor\mathrm{E} \qquad \frac{\Delta \dashv E : \lfloor_{\mathsf{m}\cdot\mathsf{n}}\mathsf{T}\rfloor \rightarrowtail \mathsf{U}_0}{\Delta \dashv E \circ [\,] \lhd \mathsf{Mod}_{\mathsf{m}} : \lfloor_{\mathsf{n}} !_{\mathsf{m}}\mathsf{T}\rfloor \rightarrowtail \mathsf{U}_0}\ \lfloor!\rfloor\mathrm{E}$$

$$\frac{\Delta_1,\ (1\!\uparrow\cdot\mathsf{n})\cdot\Delta_2 \dashv E : \mathsf{1} \rightarrowtail \mathsf{U}_0 \quad \Delta_2,\ x :_{\mathsf{m}} \mathsf{T} \vdash u : \mathsf{U}}{\Delta_1 \dashv E \circ [\,] \lhd (\lambda x_{\,\mathsf{m}} \mapsto u) : \lfloor_{\mathsf{n}}\mathsf{T}_{\mathsf{m}} \multimap \mathsf{U}\rfloor \rightarrowtail \mathsf{U}_0}\ \lfloor\multimap\rfloor\mathrm{E} \qquad \frac{\Delta_1,\ 1\!\uparrow\cdot\Delta_2 \dashv E : \mathsf{T} \rightarrowtail \mathsf{U}_0 \quad \Delta_2 \vdash t' : \mathsf{U} \ltimes \mathsf{T}}{\Delta_1 \dashv E \circ [\,] \lhd\!\circ\, t' : \lfloor_{1_\nu}\mathsf{U}\rfloor \rightarrowtail \mathsf{U}_0}\ \lfloor\ \rfloor\mathrm{E}_{\mathrm{C1}}$$

$$\frac{\Delta_1,\ 1\!\uparrow\cdot\Delta_2 \dashv E : \mathsf{T} \rightarrowtail \mathsf{U}_0 \quad \Delta_1 \vdash v : \lfloor_{1_\nu}\mathsf{U}\rfloor}{\Delta_2 \dashv E \circ v \lhd\!\circ\, [\,] : \mathsf{U} \ltimes \mathsf{T} \rightarrowtail \mathsf{U}_0}\ \lfloor\ \rfloor\mathrm{E}_{\mathrm{C2}} \qquad \frac{\Delta_1,\ (1\!\uparrow\cdot\mathsf{n})\cdot\Delta_2 \dashv E : \mathsf{1} \rightarrowtail \mathsf{U}_0 \quad \Delta_2 \vdash t' : \mathsf{T}}{\Delta_1 \dashv E \circ [\,] \blacktriangleleft t' : \lfloor_{\mathsf{n}}\mathsf{T}\rfloor \rightarrowtail \mathsf{U}_0}\ \lfloor\ \rfloor\mathrm{E}_{\mathrm{L1}}$$

$$\frac{\Delta_1,\ (1\!\uparrow\cdot\mathsf{n})\cdot\Delta_2 \dashv E : \mathsf{1} \rightarrowtail \mathsf{U}_0 \quad \Delta_1 \vdash v : \lfloor_{\mathsf{n}}\mathsf{T}\rfloor}{\Delta_2 \dashv E \circ v \blacktriangleleft [\,] : \mathsf{T} \rightarrowtail \mathsf{U}_0}\ \lfloor\ \rfloor\mathrm{E}_{\mathrm{L2}} \qquad \frac{\mathsf{hnames}(E)\ \#\#\ \mathsf{hnames}(\Delta_3) \quad \Delta_1,\ \Delta_2 \dashv E : (\mathsf{U} \ltimes \mathsf{T}') \rightarrowtail \mathsf{U}_0 \quad \Delta_2,\ \rightarrow^{i}\Delta_3 \vdash_{\mathbf{v}} v_2 : \mathsf{U}}{1\!\uparrow\cdot\Delta_1,\ \Delta_3 \dashv E \circ {}^{\mathrm{op}}_{\mathsf{hnames}(\Delta_3)}/v_2\,{}_{9}[\,]/ : \mathsf{T}' \rightarrowtail \mathsf{U}_0}\ \ltimes\mathrm{OP}$$

$\boxed{\vdash E[\,t\,] : \mathsf{T}}$ （命令的类型判断）

$$\frac{\Delta \dashv E : \mathsf{T} \rightarrowtail \mathsf{U}_0 \quad \Delta \vdash t : \mathsf{T}}{\vdash E[\,t\,] : \mathsf{U}_0}\ \lambda_d\text{-}\mathrm{TY}_{\mathrm{CMD}}$$

图 2.8：求值上下文（λ_d–ty$_{\mathrm{E}}$）和命令（λ_d–ty$_{\mathrm{CMD}}$）的类型规则

2.6.3 归约语义

现在所有部分都已就绪，让我们关注 λ_d 的归约规则，这些规则展示于图 2.9 和图 2.10 中。与 λ_{Ldm} 的第 1.6 节一样，λ_d 的聚焦（F 后缀）、取消聚焦（U 后缀）和收缩（C 后缀）规则都是以完全确定的方式触发的。一旦子项成为值，就不能再次对其进行聚焦。

图 2.9 展示了所有不会对求值上下文执行任何替换的规则

函数应用和模式匹配的归约规则与 λ_{Ldm} 中的相同。$\mathtt{to}_\times$ 的归约规则相当直接：一旦我们得到一个值，就将其嵌入一个平凡 ampar 中，并在右侧放置单位。$\mathtt{from}_\times$ 的规则相当对称：一旦我们得到一个右侧具有 $\mathtt{Mod}_{1\otimes v_1}$ 形状的值的 ampar，就可以将左侧和右侧都从 ampar 外壳中提取出来，组成一个普通对。

最后，$\mathtt{新建}_\times$ 只有一条规则，会将其转换为“恒等” ampar 对象：左侧只有一个空洞，右侧则是相应的目标。

图 2.10 中的规则都与 $\mathrm{upd}_\times$ 和目的地填充算子有关；它们的收缩规则会深入修改求值上下文，而不只是压入或弹出一个聚焦组件。为此，我们需要引入特殊替换 $E\,(\!|h :=_H v|\!)$，用于更新栈中附着于开放 ampar 聚焦组件的构造中的结构。当焦点项中的目标 $\to h$ 被填充时，就会触发这种替换；通常，这发生在目的地填充原语的归约过程中，并会导致值 v 被写入空洞 h。值 v 本身也可能包含空洞（例如，当空心构造器 Inl h′+1 被写入 λ_d–sem/$\lfloor\oplus\rfloor$E₁C 中的空洞 h 时），因此集合 H 会跟踪值 v 引入的潜在孔名称，并用于更新相应（开放）ampar 的孔名称集合。$E\,(\!|h :=_H v|\!)$ 的正式定义见图 2.9 底部。

λ_d–sem/$\lfloor 1\rfloor$EC 和 λ_d–sem/$\lfloor\multimap\rfloor$EC（图 2.10）不会创建任何新空洞；它们只会向现有空洞写入一个值。另一方面，规则 λ_d–sem/$\lfloor\oplus\rfloor$E₁C、λ_d–sem/$\lfloor\oplus\rfloor$E₂C、λ_d–sem/$\lfloor!\rfloor$EC 和 λ_d–sem/$\lfloor\otimes\rfloor$EC 都会向空洞 h 写入一个包含新空洞的空心构造器。因此，我们需要为这些新空洞生成新鲜名称，并为每个新空洞返回一个具有匹配名称的目标。

只有当 $E(\!|h :=_H v|\!)$ 中的替换应当仅在 h 是全局唯一名称时执行；否则，我们就会破坏一次写入内存模型的承诺。为此，当 ampar 处于值形式时，我们允许名称遮蔽（这就是为什么每次都允许 $\mathtt{new}_\times$ 归约为相同的 $\{1\}/1,\to 1/$），但一旦 ampar 被开放（即成为开放 ampar 聚焦组件），它就应当具有全局唯一的孔名称。在规则 λ_d–ty$_E$/$\times$op 中，通过前提 $\mathtt{hnames}(E)$ ## $\mathtt{hnames}(\Delta_3)$强制执行这一限制：当在 $\mathtt{upd}_\times$ 的归约过程中创建开放 ampar 聚焦组件时，要求来自 E 和 Δ_3 的孔名称集合互不相交。同样，写入空洞的任何空心构造器都应具有全局唯一的孔名称。为简单起见，我们假设孔名称是自然数。

为获得全局新名称，在相应规则的前提中，我们首先设定 $h' = \max(\mathrm{hnames}(E) \cup \{h\})+1$，或对 h″ 和 h‴ 作类似定义（见图 2.10 底部），以查找一个新的未使用名称[19]。然后，我们使用移位后的集合 $H_{\pm h'}$，或使用图 2.9 底部定义的条件移位算子 $h[H_{\pm h'}]$，将所有名称或仅特定名称替换为新鲜的未使用名称。我们以显然的方式将条件移位 $\bullet[H_{\pm h'}]$ 扩展到任意值、项和类型上下文（注意，$_H/v_2,v_1/$ 会绑定 H′ 中的名称）。

[19]有时我们不会选取可能的最小未使用名称；这通常是为了让形式化证明更容易而采用了方便的做法。例如，在使用 Coq 进行形式化证明时，重命名是通过更通用的置换实现的，这样更容易与表示为函数的类型上下文交互，但要使引理成立，也因此增加了一些额外要求。

$E\ [\ t\ ]\longrightarrow E'\ [\ t'\ ]$ (命令的小步求值

$E\ [\ t'\ t\ ]\longrightarrow (E\ \circ t'\ [\,]\ )\ [\ t\ ]$ ⋆ $\multimap EF_1($

$E\ \circ t'\ [\,]\ )\ [\ v\ ]\longrightarrow E\ [\ t'\ v\ ]$ $\multimap EU_1$

$E\ [\ t'\ v\ ]\longrightarrow (E\ \circ\ [\,]\ v)\ [\ t'\ ]$ ⋆ $\multimap$

$EF_2(E\ \circ\ [\,]\ v)\ [\ v'\ ]\longrightarrow E\ [\ v'\ v\ ]$ $\multimap$

$EU_2 E\ [\ (\lambda_{\_} x\ m\mapsto u)\ v\ ]\longrightarrow E\ [\ u[x:=v$

$]\ ]$ $\multimap EC\ E\ [\ t\ ;u\ ]\longrightarrow (E\ \circ\ [\,]\ ;u)\ [\ t$

$]$ ⋆ $1EF(E\ \circ\ [\,]\ ;u)\ [\ v\ ]\longrightarrow E\ [\ v\ ;u$

$]$ $1EU\ E\ [\ ()\ ;u\ ]\longrightarrow E\ [\ u$

$]$ $1EC\ E\ [\ 分情况_m\ t\ of\ \{Inl\ x$

$_1\mapsto u_1\ ,Inr\ x_2\mapsto u_2\}\ ]\longrightarrow (E\ \circ 分情况_m\ [\,]\ of\ \{Inl\ x_1\mapsto u_1\ ,Inr\ x_2\mapsto u_2\})\ [\ t\ ]$ ⋆ $\oplus EF(E\ \circ 分情况$

$_m\ [\,]\ of\ \{Inl\ x_1\mapsto u_1\ ,Inr\ x_2\mapsto u_2\})\ [\ v\ ]\longrightarrow E\ [\ 分情况_m\ v\ of\ \{Inl\ x_1\mapsto u_1\ ,Inr\ x_2\mapsto u_2\}\ ]$ $\oplus EU\ E\ [\ 分$

$情况_m\ (Inl\ v_1)\ of\ \{Inl\ x_1\mapsto u_1\ ,Inr\ x_2\mapsto u_2\}\ ]\longrightarrow E\ [\ u_1[x_1:=v_1]\ ]$ $\oplus EC_1 E\ [$

$分情况_m\ (Inr\ v_2)\ of\ \{Inl\ x_1\mapsto u_1\ ,Inr\ x_2\mapsto u_2\}\ ]\longrightarrow E\ [\ u_2[x_2:=v_2]\ ]$ $\oplus$

$EC_2 E\ [\ 分情况_m\ t\ of\ (x_1\ ,x_2)\ \mapsto u\ ]\longrightarrow (E\ \circ 分情况_m\ [\,]\ of\ (x_1\ ,x_2)\ \mapsto u)\ [\ t$

$]$ ⋆ $\otimes EF(E\ \circ 分情况_m\ [\,]\ of\ (x_1\ ,x_2)\ \mapsto u)\ [\ v\ ]\longrightarrow E\ [\ 分情况_m\ v\ of\ (x_1\ ,x_2)$

$\mapsto u\ ]$ $\otimes EU\ E\ [\ 分情况_m\ (v_1\ ,v_2)\ of\ (x_1\ ,x_2)\ \mapsto u\ ]\longrightarrow E\ [\ u[x_1:=v_1][x_2:=v$

$_2]\ ]$ $\otimes EC\ E\ [\ 分情况_m\ t\ of\ Mod_n\ x\ \mapsto u\ ]\longrightarrow (E\ \circ 分情况_m\ [\,]\ of$

$Mod_n\ x\ \mapsto u)\ [\ t\ ]$ ⋆ $!EF(E\ \circ 分情况_m\ [\,]\ of\ Mod_n\ x\ \mapsto u)\ [\ v\ ]\longrightarrow E\ [\ 分情$

$况_m\ v\ of\ Mod_n\ x\ \mapsto u\ ]$ $!EU\ E\ [\ 分情况_m\ Mod_n\ v'\ of\ Mod_n\ x\ \mapsto u\ ]\longrightarrow E\ [\ u[x:=v']\ ]$ $!EC\ E\ [\ to_\ltimes\ u\ ]\longrightarrow ($

$E\ \circ to_\ltimes\ [\,])\ [\ u\ ]$ ⋆ $\ltimes toF(E\ \circ to_\ltimes\ [\,])\ [\ v$

$_2\ ]\longrightarrow E\ [\ to_\ltimes\ v_2\ ]$ $\ltimes toU\ E\ [\ to_\ltimes\ v_2\ ]\longrightarrow E$

$[\ \{\}/v^2\ 9()/\ ]$ $\ltimes toC\ E\ [\ from_\ltimes\ t\ ]\longrightarrow ($

$E\ \circ from_\ltimes\ [\,])\ [\ t\ ]$ ⋆ $\ltimes fromF(E\ \circ from_\ltimes\ [\,])\ [$

$v\ ]\longrightarrow E\ [\ from_\ltimes\ v\ ]$ $\ltimes fromU\ E\ [\ from^\times\ \{\}/$

$v^2\ 9Mod_{1\infty}\ v_1/\ ]\longrightarrow E\ [\ (v_2\ ,Mod_{1\infty}\ v_1)\ ]$ $\ltimes fromC\ E\ [\ 新建$

$_\ltimes\ ]\longrightarrow E\ [\ \{1\}/\ 1\ ,\to 1/\ ]$ $\ltimes newC$

□

⋆：仅当将成为新焦点的项尚不是一个值时才允许

名称集移位和条件名称移位：

$$H\pm h' \triangleq \{h+h' \mid h \in H\}$$

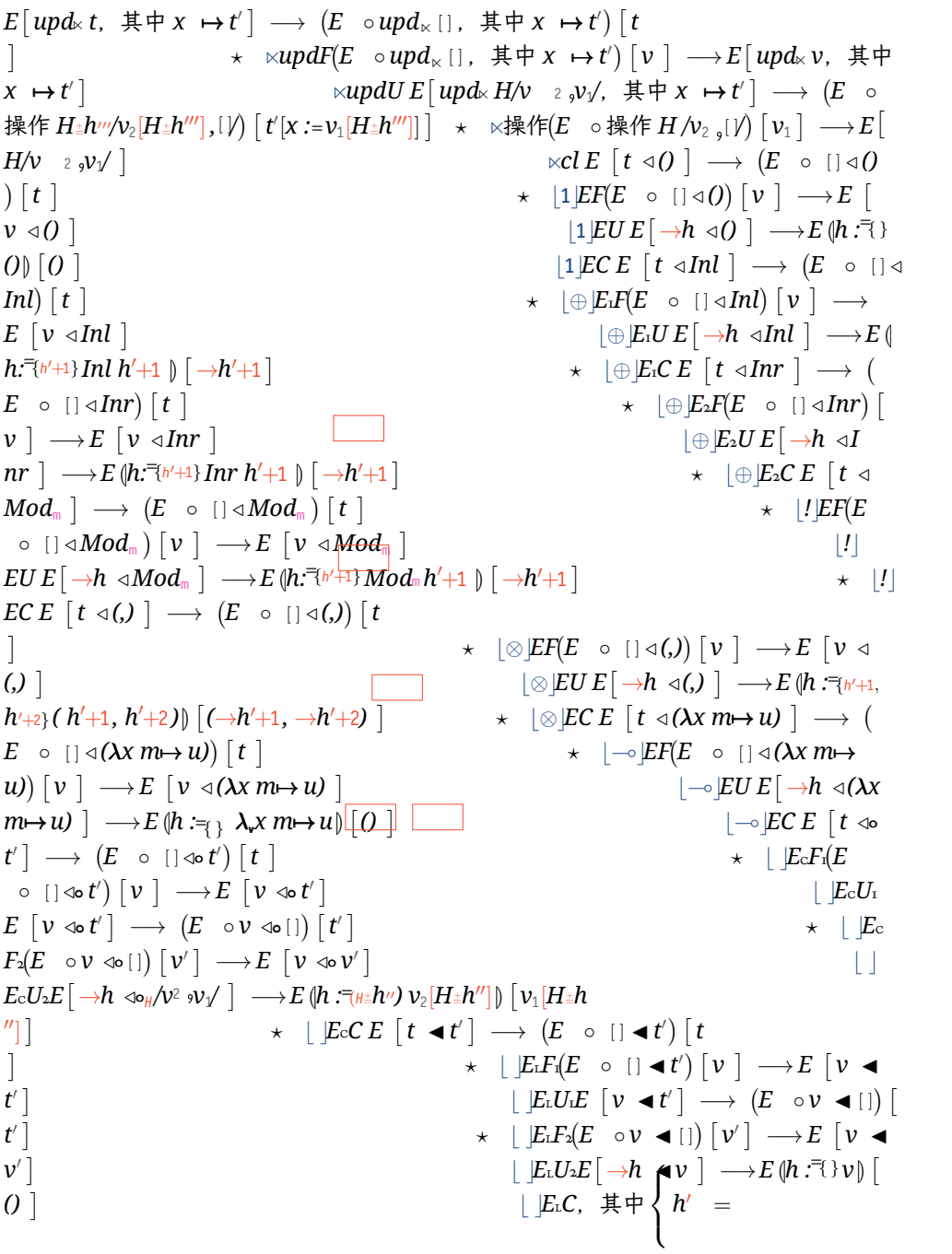
$$h[H\pm h'] \triangleq \begin{cases} h+h' & \text{如果} h \in H \\ h & \text{否则} \end{cases}$$

开放振幅的特殊替换：

$$(E\ \circ\ {}^{\mathrm{op}}_{\{h\}\sqcup H}/v_2\ {}_9[\,]/)\ (h:=_{H'}\ v')\ = E\ \circ\ {}^{\mathrm{op}}_{H\sqcup H'}/v_2\ (h:=_{H'}\ v')\ {}_9[\,]/$$

$$(E\ \circ\ e)\ (h:=_{H'}\ v')\ = E\ (h:=_{H'}\ v')\ \circ\ e \qquad \text{if } h \notin e$$

图 2.9： λ_d （λ_d–sem）的 小步语义，第 1 部分



^{y} 仅当将成为新焦点的项尚未　　y a value

图 2.10： λ_d (λ_d–sem) 的小步语义，第 2 部分

规则 λ_d–sem/⋉op 和 λ_d–sem/⋉cl 规定了 ampar（一个值）如何以及何时转换为开放 ampar（一个聚焦组件），以及如何反向转换；这些规则使用了我们刚刚介绍的移位策略。根据 λ_d–sem/⋉op，ampar 绑定的孔名称会被重命名为新的名称，左侧会附加到聚焦组件 op H±h′/v$_2$[H±h″]，[]/ 上，而右侧（包含目的地）会代入 upd⋉ 语句的主体中（该主体会成为新的焦点项）。当 upd⋉ 语句的主体规约为一个值时，规则 λ_d–sem/⋉cl 便会触发。在这种情况下，我们可以通过从栈 E 中弹出聚焦组件，并将其与 v$_2$ 合并，从而关闭 ampar，再次形成一个 ampar 值。

在规则 λ_d–sem/⌊⌋EC 中，我们将 ampar 值 H/v$_2$,v$_1$/ 的左侧 v$_2$ 写入某个结构 E 内部某处的空洞 h 中。这会将两个带孔结构组合起来。由于我们将先前由 ampar 联结词绑定在一起的 v$_2$ 和 v$_1$ 分离开来（v$_2$ 与另一个结构合并，而 v$_1$ 成为新的焦点），它们的孔名称不再受到绑定，因此我们需要确保这些名称全局唯一，就像使用 upd⋉ 打开 ampar 时所做的那样。此重命名由条件移位 v$_2$[H±h″] 和 v$_1$[H±h″] 执行。

实践应用 让我们重复使用第 2.5.2 节中带有合成 (::) *cons* 构造器的示例，并观察表达式 () :: Inl() 的归约。这里，Inl () 是对应于 *nil* 的值形式（根据我们对列表的编码），并且我们还假设表达式中的 () 是图 2.5 的值形式，而不是代码清单 2.4 中定义的语法构造器。最后，为简洁起见，我们将 ′⋉ from 视为原语，其收缩规则为 λ_d–SEM/⋉FROM′C : $E\left[\,\mathbf{from}'_{\ltimes}(\{\}/v^2\ ,()/)\,\right] \longrightarrow E\left[\,v_2\,\right]$，并采用平凡的聚焦规则和取消聚焦规则。

[][0 :: Inl()]

⟶ [][from′⋉(upd⋉ 新建⋉ with d ↦ 分情况 (d ◁Inr ◁(,)) of (dx, dxs) ↦ dx ◀() ⨟dxs ◀Inl())]

⟶ ([] ∘from′⋉ []) [upd⋉ 新建⋉ with d ↦ 分情况 (d ◁Inr ◁(,)) of (dx , dxs) ↦ dx ◀() ⨟dxs ◀Inl()]

⟶ ([] ∘from′⋉ [] ∘upd⋉ [] with d ↦ 分情况 (d ◁Inr ◁(,)) of (dx, dxs) ↦ dx ◀() ⨟dxs ◀Inl ()) [新建⋉]

⟶ ([] ∘from′⋉ [] ∘更新⋉ [] 使用 d ↦ 分情况 (d ◁Inr ◁(,)) 取值 (dx, dxs) ↦ dx ◀() ⨟dxs ◀Inl ()) [{1}/ 1 ,→1/]

⟶ ([] ∘from′⋉ []) [更新⋉ {1}/ 1 ,→1/ 带有 d ↦ 分情况 (d ◁Inr ◁(,)) 关于 (dx, dxs) ↦ dx ◀() ⨟dxs ◀Inl ()]

⟶ ([] ∘from′⋉ [] ∘操作 {2}/ 2 ,[]/) [分情况 (→2 ◁Inr ◁(,)) of (dx , dxs) ↦ dx ◀() ⨟dxs ◀Inl()]

⟶ ([] ∘from′⋉ [] ∘操作 {2}/ 2 ,[]/ ∘分情况1′ [] of (dx , dxs) ↦ dx ◀() ⨟dxs ◀Inl()) [→2 ◁Inr ◁(,)]

⟶ ([] ∘from′⋉ [] ∘操作 {2}/ 2 ,[]/ ∘分情况1′ [] of (dx , dxs) ↦ dx ◀() ⨟dxs ◀Inl() ∘ [] ◁(,)) [→2 ◁Inr]

⟶ ([] ∘from′⋉ [] ∘操作 {2}/Inr 4 ,[]/ ∘分情况1′ [] of (dx, dxs) ↦ dx ◀() ⨟dxs ◀Inl () ∘ [] ◁(,)) [→4]

⟶ ([] ∘from′⋉ [] ∘操作 {4}/Inr 4 ,[]/ ∘分情况1′ [] 于 (dx, dxs) ↦ dx ◀() ⨟dxs ◀Inl ()) [→4 ◁(,)]

⟶ ([] ∘from′⋉ [] ∘操作 {6,7}/Inr (6 , 7) , []/ ∘分情况1′ [] 于 (dx, dxs) ↦ dx ◀() ⨟dxs ◀Inl ()) [(→6 , →7)]

⟶ ([] ∘from′⋉ [] ∘操作 {6,7}/Inr (6 , 7) , []/) [分情况 (→6 , →7) of (dx, dxs) ↦ dx ◀() ⨟dxs ◀Inl ()]

⟶ ([] ∘from′⋉ [] ∘操作 {6,7}/Inr(6 , 7) , []/) [→6 ◀() ⨟ →7 ◀Inl ()]

⟶ ([] ∘from′⋉ [] ∘操作 {6,7}/Inr(6 , 7) , []/ ∘ [] ⨟ →7 ◀Inl ()) [→6 ◀()]

⟶ ([] ∘from′⋉ [] ∘操作 {7}/Inr (() , 7) , []/ ∘ [] ⨟ →7 ◀Inl ()) [()]

⟶ ([] ∘from′⋉ [] ∘操作 {7}/Inr (() , 7) , []/) [() ⨟ →7 ◀Inl ()]

⟶ ([] ∘from′⋉ [] ∘操作 {7}/Inr (() , 7) , []/) [→7 ◀Inl()]

⟶ ([] ∘from′⋉ [] ∘操作 {}/Inr (() , Inl ()) , []/) [()]

⟶ ([] ∘from′⋉ []) [{}/Inr (() , Inl ()) , ()/]

⟶ [][from′⋉({}/Inr (() , Inl ()) ,()/)]

⟶ [][Inr (() , Inl ())]

展开 (::) 和 ◁(::) 定义

⋉from'F

⋉updF

⋉新建C语言

⋉updU

⋉OP

⊗EF

⌊⊗⌋EF

⌊⊕⌋E₂C语言 和子式。 E(2: ∀{4} Inr 4)

⌊⊗⌋欧盟

⌊⊗⌋EC 与子式。 E(4: ∀{6,7} (6 , 7))

⊗EU

⊗EC

1EF

⌊ ⌋E₁C语言 与子类型。 E(6: ∀{} ())

1EU

1EC

⌊ ⌋E₁C，带有子类型。 E(7:∀{} Inl ())

⋉CL

⋉from'U

⋉来自'C

归约结束时，我们得到空求值上下文，其中焦点上的值为`Inr`((), `Inl`())，这正是仅包含单位的单元素列表的值形式，符合预期。

我们还可以看到，在整个归约过程中，焦点下目标填充操作的收缩会触发对求值上下文 $E\langle h :=_H v \rangle$ 的全局替换 E。如前所述，这些替换负责改变正在构造的结构；该结构始终附着在求值上下文 开放 ***ampar*** 聚焦组件上，位于求值上下文中的某处E。

2.6.4 类型安全

现在语义已经定义，我们可以陈述通常的类型安全定理：

定理 1 （类型保持）。 若 $\vdash E[t] ::: \mathtt{T}$ 且 $E\ [t] \longrightarrow E'\ [t']$，则 $\vdash E'[t'] ::: \mathtt{T}$。

定理 2 （进展）。 若 $\vdash E[t] ::: \mathtt{T}$ 且 $\forall v,\ E[t] \neq [][v]$ 则 $\exists E',\ t'.E\ [t] \longrightarrow E'\ [t']$。

如上例所示，形式为 $[][v]$ 的命令无法继续归约，因为它只包含一个完全确定的值，没有待处理的计算。这就是归约的终点，任何类型正确的命令最终都会达到这种形式。

鉴于我们在第 2.6.1 节中对空洞和目标的类型规则，我们可以保证，闭合程序不包含任何自由洞或自由目标，并且归约结束时 $[][v]$ 中剩余的任何空洞和目标都必须正确配平，并包裹在一个 ampar 中。

2.7 类型安全性的形式化证明

我们对 λ_d 进行了形式化，并使用 Coq证明助手在 ⟦Bagrel 和 Spiwack，2025b⟧ 中证明了类型保持和进展定理。

转而使用证明助理是一个务实的选择：处理 λ_d 中的类型上下文可能相当繁琐，而且如果没有计算机辅助，很难确保我们的证明没有错误。我们所证明的 λ_d 版本以 Ott ⟦Sewell 等人，2007⟧，编写，并且同一个 Ott 文件还被用作本文档中所展示类型规则的来源，从而确保我们证明的系统与所介绍的系统相同；不过，为了增强视觉清晰度，编译期间脚本会进行一些简化。

大部分证明都是我独立完成的，而我此前几乎没有使用 Coq 的经验。不过，在工业界导师的帮助下，开发进度大幅加快。他凭借丰富的 Coq 使用经验，引入了一些合适的核心引理，我们据此便能轻松地构建许多其他引理。

整个证明总计约 7000 行，包含近 500 条引理。类型保持和进展引理中的许多分情况都很相似。为了处理这些重复的分情况，基于大型语言模型的自动补全系统被证明相当有效，尽管这牺牲了证明的优雅性。例如，我们没有对半环进行任何抽象形式化：手动逐一强制验证所需性质更为省事。

尽管如此，我们的 Coq 开发中仍有几点值得关注，接下来几段将对此进行介绍。

类型上下文的表示我们将上下文表示为有限定义域函数，而不是语法列表，因为在定义上下文的和类型时，这种表示方式效果好得多。生态系统中虽然有少数几个有限函数库，但我们需要的是有限依赖函数（因为绑定符的类型取决于我们绑定的是变量名还是孔名）。当时并没有这样的库，但就我们的有限需求而言，自己实现一个是可行的（证明约 1000 行）。底层数据类型实际上是函数：这种做法更容易实现，但正如我们很快会看到的，它会使相等的处理比使用专门设计的数据类型更复杂。

有限函数和上下文类型的定义如下：

```
定义 支持 {A B} (l : 列表 A)(f: 对于所有 x:A, 选项类型 (B x))          : 命题
 2 :=对于所有 (x:A) (y:B x)，f x  = Some y -> 列表.In x l。
3 4 记录类型 T A B := { 5 underlying :> 对于所有 x:A，选项
类型 (B x); 6 supported : 存在 l : 列表 A，支持 l
underlying; 7 }。8 归纳类型 名称 : 类型 :=9       |
name_Var (x:变量)10       | name_DH (h:hname)。11 12
定义 绑定类型 (n : 名称) : 类型 :=匹配 nwith14       |
name_Var _  ⇒ binding_var15       | name_DH _  ⇒
binding_dh 16 end。17 18 归纳类型 binding_var : 类型 :=
19       | binding_Var (m:模式) (T:类型)。20 21 归纳类型
binding_dh : 类型 :=22       | binding_Dest (m:模式) (T:类
型) (n:模式)23       | binding_Hole (T:类型) (n:模式)。24
25 定义 上下文 : 类型 := T name binding_type_of。
```

支持谓词表示某个列表是我们的偏函数有效定义域的过近似。随后，在需要时，我们使用经典的构造性不定描述公理，根据给定的上下文获取合适的支持列表：

```
1 (* 公理 *）2 ClassicalEpsilon.constructive_indefinite_description 3：对所
有（A：类型）和（P：A ->命题），（存在 x：A，P x）-> {x：A  | P x} 4 5 定
义 a_support {A B}（f：T A B）：列表 A 6：=
constructive_indefinite_description _ f.(supported)。
```

我们还引入ctx 的定义域作为一个用于筛选a_support ctx 的筛选器，以仅保留我们的偏函数有效定义域中的元素ctx。

在证明中，我们还需要对类型上下文的相等进行推理。我们的偏函数类型定义天然适用于集合oid相等，但遗憾的是，通过 Ott 生成的推理规则并不支持这种相等。

作为变通方法，我们假设经典的函数外延性和证明无关性公理，从而可以断言，只要两个上下文对所有变量和孔名称赋予相同的绑定，它们就相等：

```
1 (* 公理 *) 2 ProofIrrelevance.proof_irrelevance : forall (P : 命题) (p1 p2 : P), p1 =
p2 3 FunctionalExtensionality.functional_extensionality_dep 4 : forall (A : 类型) (B : A ->
类型) (f g : forall x : A, B x), 5 (forall x : A, f x = g x) ->f = g 6 7 引理 ext_eq' : forall {A B} (f
g : T AB), f.(underlying) = g.(underlying) -> f = g. 8 Proof. intros A B [f f_supp] [g
g_supp]h_ext. cbn in *. subst g. f_equal. 9 apply proof_irrelevance. 10 Qed. 11 12 引理
ext_eq : forall {A B} (f g : T A B), (forall x, f x = g x) -> f = g. 13 Proof. intros * h_ext.
apply ext_eq'. 14 apply functional_extensionality_dep. 15 assumption. 16 Qed.
```

实际上，我们恢复了一种外延相等，其行为从本质上看类似于集合oid相等，但能与我们通过 Ott 生成的基础设施无缝集成。

上下文的部分加法 如第 2.5.2 节和第 2.6.1 节所示，上下文的加法具有很强的部分性。例如，只有当两个绑定的名称和类型都相同时，才能将它们合并；对于同一个孔名，我们也不能将孔绑定与目的地绑定合并。

我们不将加法编码为一个返回可选上下文的二元函数——这种做法与 Ott 生成的规则难以集成——而是将其建模为一个从上下文到上下文的全函数。相反，为了表示失败情况，我们在模态半环中引入一种特殊的错误模式。携带此错误模式的绑定表示整个上下文被视为无效：

```
1 定义 mode : 类型 := 选项类型 (mul * 年龄)。 2 记号 "' '" := None。 3 记号 "
'1∨'" :=(Some (pair Lin (Fin 0)))。 4 记号 "'1 ↑'" :=(Some (pair Lin (Fin 1)))。 5 记号
"'1 ∞'" :=(Some (pair Lin Inf))。 6 ... 7 8 定义 mode_plus (m1 m2: mode) :
mode :=9 匹配 m1、m2，若为10 | None, _ ⇒None11 | _, None ⇒
None12 | Some (p1, a1), Some (p2, a2) ⇒ Some (mul_plus p1 p2,
age_plus a1 a2) 13 结束。 14 15 归纳类型 IsValid : mode -> 命题 16 := 归纳类型
IsValidProof : forall (pa : mul * 年龄), IsValid (Some pa)。
```

```
17 18 定义 ValidOnly (G: 上下文) : 命题 19 := forall (n : 名称)(binding: 绑定类型), G n  = Some binding -> 有
效 (mode_of binding)。 20 ... 21 22 引理 ValidOnly_union_forward 23 : forall (G1 G2 : 上下文), ValidOnly
G1 ->ValidOnly G2 -> G1 # G2 -> ValidOnly (G1 + G2)。 24 引理 ValidOnly_union_backward 25 : forall (
G1 G2 : 上下文), ValidOnly (G1 + G2) -> ValidOnly G1 /\ ValidOnly G2。 26 ...
```

这种方法的代价是，现在大多数类型规则都必须包含显式的良构条件——例如 有效 和 仅有效——以确保输入上下文不含任何错误绑定。还需要额外的引理，通过上下文操作传播这些有效性条件，并利用不相交性等性质来保证结果上下文中不存在错误。

2.8 将先前工作翻译为 λ_d 项和类型

我们在第 2.1 节中指出， λ_d 涵盖了现有系统。下面来看看具体情况。

带孔数据结构的函数式表示 Minamide 系统 [Minamide, 1998]^^^引入了两个基本操作，happ 和 hcomp，以及孔抽象形式 $\lambda x \mapsto t$，用于表示带有一个空洞的结构，其中x在t中恰好出现一次。

论文指出，孔抽象中不能有 λ 抽象、应用或分情况表达式作用于该抽象的变量；但除此之外，如果这些结构作用于其他变量或值，则可以出现在孔抽象的主体中。

^^^ 在类型正确的闭合程序中，我们可以尽可能地求值孔抽象（在 λ下）的主体，直到它除了单次出现的 x 之外几乎成为一个值（我们知道 x 不会导致归约卡住，因为它不会出现在应用或 case 表达式的求值路径中）。^^^ 完全求值的孔抽象随后可直接转换为我们的 ampar 值形式：我们将 λx ↦ t 解释为 {h}/t[x :=h],→h/。换句话说，我们把孔抽象的主体作为 ampar 的左侧，将其中单次出现的变量替换为空洞 h，并直接将对应的目标 →h 用作右侧。孔抽象的类型 (T, S)hfun 转换为 S ∝ ⌊T⌋。

在一般情况下，孔抽象可能尚未完全求值，我们通过以下一系列操作来解释它：

- 新建一个 ampar，使用 新建∝；
- 使用upd∝打开它；
- 通过填充表达式链 $d \triangleleft \ldots$或 $d \blacktriangleleft \ldots$构造出包含空洞变量的数据构造器（按照我们从目的地填充算子中还原普通数据构造器或复合构造器时所用的相同方法；参见第 2.2.2 节开头的 (::)）；

- 将我们希望放置空洞的数据构造器字段所对应的目标，作为upd∝的主体的返回结果；

- 其他构造，例如 case 表达式，或与其他变量或值相关的应用，会原封不动地转置到upd∝的主体中。

对这种转换给出形式化描述有些繁琐，但这一过程始终是系统性的；通过一个示例可以看得相当清楚：左侧是 Minamide 的项，右侧是 λ_d 项：

ha ::: (T, T 列表)hfunha
≜ λ^^^x ↦ 0 :: x ::(f 2)

$$
\begin{aligned}
&\mathtt{ha}_{\lambda_d} \;:\; (\mathtt{List}\ \mathtt{T}) \ltimes \lfloor \mathtt{T} \rfloor \\
&\mathtt{ha}_{\lambda_d} \triangleq \mathtt{case}\ \mathtt{upd}_{\ltimes}\ \mathtt{new}_{\ltimes}\ \mathtt{with}\ d \mapsto \\
&\qquad (d \lhd (::))\ \mathtt{of}\ (d_0\,, dt) \mapsto \\
&\qquad\quad \mathtt{case}\ (dt \lhd (::))\ \mathtt{of}\ (d_1\,, dt') \mapsto \\
&\qquad\qquad d_0 \blacktriangleleft 0 \fatsemi dt' \blacktriangleleft (f\ 2) \fatsemi d_1
\end{aligned}
$$

一旦 (f 2) 化简为值 v， λ_d 的版本就会求值为承诺的 ampar 形式 $H/0 :: h ::v$， $\to h/$。
□

由于 Minamide 系统中不包含目标，⟦Minamide， 1998⟧, Minamide 系统中的程序里的任何变量都可以赋予年龄 ∞ （因此可以采用模式 1∞ 或 $\omega\infty$，具体取决于它是否为线性变量），当然，孔抽象绑定的变量除外。

happ 和 hcomp 的翻译更加直接：

happ ::: (T, S)hfun —∘ T —∘ Shappλ_d ::: S ⋉ ⌊T ⌋ —∘ T —∘ Shappλ_d *ampar x* ≜ from′⋉(upd⋉*ampar* 与 *d* ↦ *d* ◀ *x*)

hcomp ::: (T, S)hfun —∘ (U,T)hfun —∘ (U,S)hfun hcompλ_d ::: S ⋉ ⌊T⌋ —∘ T ⋉ ⌊U⌋ —∘ S ⋉ ⌊U⌋hcompλ_d ampar$_1$ ampar$_2$ ≜ upd⋉ ampar$_1$ with d ↦ d ◁∘ampar$_2$

通过这种转换，我们可以在 λ_d 中完整表达 Minamide 系统，且不会损失灵活性。

命令式二叉搜索树的函数式本质 Leijen 和 Lorenzen ⟦2025⟧ 以及 Lorenzen、Leijen、Swierstra 和 Lindley ⟦2024⟧ 关于一等构造器上下文的研究，在表达能力和呈现方式上都与 Minamide 的孔抽象颇为相似。孔抽象 $\lambda x \mapsto t$ 写作上下文 $t[x{:=}\_]$，其主体与 t 相同，只是将命名的空洞变量 x 替换为下划线 _（在他们的一些版本中则是方括号 □）。类似地，happ 会替换为 $++.$ 运算符（注意句点），而hcomp 会替换为 $++$ 运算符。因此，从 ⟦Leijen 和 Lorenzen，2025年；Lorenzen、Leijen、Swierstra 和 Lindley 的 2024⟧ 系统到我们系统的项转换，与 Minamide 系统的转换相同。

对于类型，⟦Leijen and Lorenzen, 2025; Lorenzen, Leijen, Swierstra, and Lindley, 2024⟧ 仅定义上下文<T>，其中空洞与所得结构（填充空洞后）具有相同的类型 T。这对应于 T ⋉ ⌊T⌋ 于 λ_d 中。

2.9 使用原地内存修改实现 λ_d

第 2.5 节和第 2.6 节中介绍的形式语言并非旨在按原样实现。第 3 章和第 4 章将重点讨论大多数 λ_d 的理念的实际实现。

首先， λ_d 没有递归，否则会掩盖对该系统的形式化表述。不过，加入标准形式的递归并不会带来任何复杂性。

其次，在 λ_d 中，ampar 并非以线性方式管理；只有目的地才是。也就是说，ampar 可以被包裹在指数模态中，例如 $\mathtt{Mod}_{\omega\nu}\ \{h\}/0::h\ ,{\to}h/$（表示一个可被非线性使用的差分列表 0::），然后以不同方式使用两次：

分情况 $\mathtt{Mod}_{\omega\nu}\ \{h\}/0::h\ ,{\to}h/$ of $\mathtt{Mod}_{\omega\nu}\ x$
$\mapsto$let $x_1 := x$ 追加 1 in let $x_2 := x$ 追加 2 in
toList $(x_1$ concat$x_2)$ $\longrightarrow^*$ $0::1::0::2::[]$

乍看之下，这似乎有悖直觉，但这个程序在 λ_d中是有效且安全的。得益于我们在第 2.6.3 节详细说明的重命名规则，每当对某个 ampar 执行$\mathtt{upd}_{\times}$时，它的孔名称都会被重命名为全新的名称。实践中实现这一点的一种方法是，每次调用$\mathtt{upd}_{\times}$ 时，都为它分配一个新的x 副本（回想一下，append 是通过$\mathtt{upd}_{\times}$实现的），采用写时复制 的方式。这样，填充目的地仍然是通过变异实现的。

然而，这与第 2.2 节承诺的高效实现相去甚远。借助完全原地函数式编程，可以优化写时复制 [Lorenzen、Leijen 和 Swierstra， 2023],其中，得益于引用计数，当差分列表没有别名时，我们无需执行复制。对于带孔结构而言，这一思想在他们较新的著作第 7 章中得到了进一步探讨 [Leijen 和 Lorenzen， 2025]。不过，在接下来的论述中，我们不会沿着这个方向展开，因为我们不想显式处理引用计数。

另一种做法是进一步细化线性类型系统，以保证 ampar 具有唯一性，并彻底避免写时复制。我们在 λ_d 的形式化中没有采用这种做法，因为这同样会使系统的阐述变得晦涩，却没有带来多少好处。

为了使 ampar 具有线性，我们遵循在第 1.8 节中制定的方案，该方案深受 [Spi- wack, 2023b; Spiwack 等人，2022], 的启发，并引入一种新的内置类型 令牌，以及原语 dup、丢弃 和 withToken。我们还将 新建$_{\times}$ 切换为 新建$_{\times\mathtt{IP}}$：

dup ::: 令牌 $\multimap$令牌$\otimes$令牌丢弃 ::: 令牌
$\multimap$ 1withToken ::: (令牌$_{1\infty}\multimap !_{\omega\infty}\mathsf{T}) \multimap !$
$\omega\infty$T新建$_{\times\mathtt{IP}}$ ::: 令牌 $\multimap \mathsf{T} \ltimes \lfloor \mathsf{T} \rfloor$

由 新建$_{\times\mathtt{IP}}$ 生成的 ampar 对 令牌存在线性依赖。如果在块 t 中两次使用同一个最初由 新建$_{\times\mathtt{IP}}$ tok创建的 ampar，那么 t 就需要一个类型上下文 $\{tok :_{\omega\infty}$令牌$\}$，因此该块会被 withToken 拒绝。另一方面，先将 tok 复制为 tok_1 和 tok_2，然后分别使用每个新令牌来创建不同的 ampar，则在线性意义上是有效的。

No由于 ampar 采用线性管理，我们可以更改分配和重命名机制 ：

· 新建 ampar 的孔名从一开始就会选取一个全新的名称（这对应于一次新的堆分配）；

· 添加新的空心构造器仍要求其孔名称是全新的（这同样对应于一次新的堆分配）；

· 对 ampar 使用 $\mathtt{upd}_{\times}$ 并填充目的地，或使用 $\lhd_{\mathrm{o}}$ 组合两个 ampar，都不再需要任何重命名：我们可以保证所有涉及的名称都是全局唯一的，且只能使用一次，因此这实际上对应于一次原地内存更新。

在专门讨论如何在函数式环境中实现 DPS 的第 3 章和第 4 章中，我们将把 ampar 视为线性资源，采用与 λ_d 扩展了 令牌 和 新建$_{\times\mathtt{IP}}$ 的方式相同的风格。

从二元积和和类型到更高效的内存形式 在 λ_d 中，我们只有 二元 积类型和和类型。不过，将语言扩展为支持 n 元积的 n 元和类型，并为每个变体实现基于目的地的构造，使其构造器直接包含多个字段，而不是像二元积的和类型那样让每个字段都需要额外的间接寻址，这非常简单 `1⊕(S⊗(T⊗U))`。事实上，我们将在下一章中这样做。不过，我们仍然要求字段的值用指针表示。这对于 ◁₀ 运算符的语义至关重要。

2.10 结论

λ_d 是一种纯函数演算，它将目的地视为一等值，可以传递和灵活使用。它支持带有多个空洞的数据结构，并允许组合带孔的数据结构，也允许将目的地存储在自身带有空洞的数据结构中。

借助 λ_d，我们现在可以在纯函数式环境中运用传统上仅限于命令式语言的某些技术和算法，同时保留安全保证。事实上，得益于由年龄系统增强的线性类型系统，使用目的地所引入的变异是不透明且受控的。

不过，我们并不认为像 λ_d 这样的年龄系统会真正用于编程语言：目的地不太可能在编程语言的设计中如此核心，以至于值得将它们如此深入地嵌入类型系统。或许，某个在优化器中大量使用目的地的编译器可以将 λ_d 用作带类型的中间表示。但更现实地说，我们预期 λ_d 可用作分析目的地传递系统的理论框架：如果一个目的地传递系统能够用 λ_d 基本操作来表达，那么它就是健全的。

在接下来的章节中，我们旨在确定需要对 λ_d的灵活性施加哪些（希望程度适中的）限制，才能将其大多数基本概念安全地移植到一种实际的函数式编程语言中，即 Haskell——它的类型系统不如 λ_d的类型系统丰富，而后者是专为目的地控制而设计的。

第3章

Haskell 中目标传递的首次实现

在上一章中，我们了解了如何从头开始构建一个 λ演算，使其对目标传递提供一等支持，同时仍能实现内存安全。现在，是时候在一种广泛使用的编程语言中付诸实践了。

本章重新审视并完善了 ⟦Bagrel，2024a⟧, 中提出的结果。该成果发表于 JFLA 2024，当时第2章所介绍的理论工作尚未完成。

3.1 实现目标：线性 Haskell

设计一种工业级编程语言仅为了添加对目的地的支持，将是一项艰巨的任务，失败的风险也很高。因此，我们将工作建立在一种现有的函数式编程语言之上，例如 Haskell[20], ，并力求在该语言中实现尽可能多的 λ_d 思想（见第2章）。Haskell 自 9.0 版本起便通过其主编译器 *GHC*获得了对线性类型的支持。1 ⟦Bernardy 等人， 2018⟧。关于线性 Haskell 的入门知识，以及如何利用它控制资源使用，敬请读者参阅第1.7节。此外，Haskell 是一种纯函数式语言，这意味着我们的目标传递 API 必须是纯的，并将（自然地具有副作用的）内存写入安全地封装其中，正如引言中所解释的那样，从而使用户无法观察到任何变异或不完整结构。

主要的挑战在于，尽管 Haskell 引入了线性类型，却没有作用域或年龄控制的概念。因此，如果不限制系统的灵活性，我们就无法避免第 2.3 节中提出的根本问题。

[20]其他工业级函数式编程语言并不多，而且其中大多数都不支持线性类型，而线性类型是构建类似 λ_d safe 这样的系统所必需的第一块基石。得益于 Lorenzen、White 等人近期的工作，⟦2024⟧, OCaml 很快就会支持仿射性，但不会支持线性性 （尽管前述工作使用了“线性性”这一术语）。仿射性确保资源至多使用一次，但不能保证资源恰好使用一次；资源可能会在未使用的情况下被丢弃。这对我们来说会是个问题，因为我们依赖这样一项保证：目标不会被丢弃。该保证确保当一个`ampar` 的右侧不再包含目标时，相应结构中的所有空洞都已填充。

3.2 仅用线性类型实现安全目的地传递

正如我们在第 2.3 节中所看到的，单靠线性类型不足以确保 λ_d 安全。我们确实需要年龄和作用域控制。在本章中，我们将采取一个大胆的决定：将目的地限制为只能用于构建非线性（也称为不受限制的）数据结构，从而彻底消除作用域逃逸问题。这也排除了第 2.3 节中描述的兄弟目标之间可能发生的交互。作为交换，我们将无法把目的地存储在使用 ampar 构建的数据结构中。我们还将用 *DPS Haskell* 来称呼我们在 Haskell 中开发的这个目的地传递 API 原型。

在实践中，这意味着目的地只能用不受限制的值填充（这就禁止用另一个目的地填充某个目的地，因为目的地是线性资源）。我们将使用 Haskell 类型 `data UDest t` 表示不受限制值的目的地，并使用类型 `data UAmpar s t` 表示其中`s` 一侧不受限制的 ampar。

由于创建线性或无限制数据结构的脊柱[21] 时并无区别，因此，我们用于向现有结构添加新空心构造器的填充函数，与 λ_d中的完全相同。与形式化定义相比，实现中唯一会改变的是作用于数据结构叶节点的填充运算符的类型签名。 λ_d 与 DPS Haskell 中运算符和类型的映射关系见图 3.1

在 λ_d中，目的地的年龄总是有限的。因此，我们使用年龄 ∞ 作为标记，表示某物不包含目的地。这样，即使不考虑线性性，我们仍然可以实施作用域控制。这在规则 λ_d–ty/$\times$（见图 2.2）中尤为明显：只要线性资源的年龄为 ∞，就可以从 ampar 的右侧提取该资源。

不过，在 DPS Haskell 中，我们没有年龄。因此，我们利用目的地也总是线性的这一事实，并使用 ω 重数作为标记，表示某物不包含目的地。当然，这并非没有代价：这样做会将目的地的缺失与非线性混为一谈。以这种方式控制作用域意味着我们必须采取更保守的做法以确保安全性，这会限制少数运算符的使用可能性。

现在，让我们看看 DPS Haskell 中的程序是什么样的。我们将从重新实现上一章中的示例开始，看看它们在实际环境中如何呈现。

3.3 DPSHaskell 编程概览

以下小节介绍三种典型情况，在这些情况下，相较于更传统的函数式实现，DPS 编程能够带来更强的表达能力或性能优势。我们将从重新审视第 2.2 节中的一些示例开始。

3.3.1 高效的差分列表

链表是函数式编程中的常见结构，但正如我们在第 2.2.3 节所见，链表不适合进行串接，尤其是在串接调用向左嵌套时。为了更高效地进行串接，我们可以定义基于目标传递的差分列表，它本质上是在末尾带有一个空洞的列表。

[21]脊柱指的是由相互连接的数据构造器组成的主链，它构成数据结构的整体形状，并连接其“叶节点”。

目的地填充算子：

λ_d	DPS Haskell		
$d \triangleleft ()$	填充 `@'()d`	or	`d&填充 @'()`
$d \triangleleft$ Inl	填充 `@'Inld`	or	`d&填充 @'Inl`
$d \triangleleft$ Inr	填充 `@'Inrd`	or	`d&填充 @'Inr`
$d \triangleleft (,)$	填充`@'(,)d`	or	`d&填充@'(,)`
$d \triangleleft$ 模块 $_{\omega\circ\circ}$	填充`@'Urd`	or	`d&填充@'Ur`
$d \triangleleft (\lambda x \ldots \mapsto$ 项$)$	`fillLeaf (\x → 项) d`	or	`d &fillLeaf (\x → 项)`
$d \triangleleft\!\circ$项	`fillComp 项 d`	or	`d &fillComp 项`
$d \blacktriangleleft$项	`fillLeaf 项 d`	or	`d&fillLeaf 项`

（可以将`&填充 @'` 视为 $\triangleleft$ 中 λ_d的等价形式，将`&fillLeaf` 视为 $\blacktriangleleft$，并将`&fillComp` 视为 $\triangleleft\!\circ$）

类型：

λ_d	DPS Haskell
$\lfloor_{\omega\circ\circ} T \rfloor$	`UDest t`
$!_{\omega\circ\circ} S \rtimes T$	`UAmpar s t`

图 3.1： λ_d 与 DPS Haskell 之间的映射

这个例子将展示 DPS Haskell 如何操作内存。到目前为止，借助 λ_d，我们使用的是一种理论上的内存模型，其中包含全局替换（作用于整个求值上下文），但没有对内存分配进行表示。在这里，使用 DPS Haskell，我们假定拥有一个适当的堆。目前我们暂不考虑垃圾回收和内存释放；这一主题将在第 3.5.1 节中讨论。

根据第 3.2 节的映射表，我们知道差分列表在 DPS Haskell 中的表示为 `type DList t = UAmpar [t] (UDest [t])`。如前所述，由于我们的差分列表是通过 `UAmpar` 构建的，因此无法容纳线性资源。这是这种针对 Haskell 的 DPS 编程初步方法的主要局限：我们无法使用高效的基于目的地的数据结构来存储线性数据。例如，第2章中给出的广度优先树遍历示例使用了这类基于目的地的差分列表来存储目的地，因此，如果不加修改，它无法在 DPS Haskell 中通过类型检查。

与 λ_d 中的 ampar 一样，`UAmpar` 的左侧承载正在构建的结构，右侧则承载该结构的目的地：`UDest [t]` 必须用一个无限制的 `[t]`填充，才能得到可读的（无限制的） `[t]`。

目标位置支持的差分列表的实现见代码清单 3.1（回顾一下， λ_d 版本的实现见代码清单 2.2）。请注意，Haskell 使用 `::`表示类型，使用 `(:)` 作为列表的 *cons* 构造器；这与这些符号在 λ_d 中的含义恰好相反。

- `newDList` 只是 `newUAmpar @[t]`, 的别名，用于通过线性方式交换一个令牌，以获得新的 `UAmpar [t] (UDest [t])`——也就是说，恰好是 `DList t`。回想一下，现在 UAmpar 被视为线性资源，并且遵循第 1.8 节和第 2.9 节所述的原则，通过交换一个线性令牌来创建。目前这个新的 UAmpar 中还没有数据，它只是一个空洞，以及一个指向该空洞的目标，如图 3.2 所示；

```
1 -- 回顾 Haskell 中列表类型的定义 2 data [t] = [] -- 空列表构造子
3             | (:) t [t] -- cons 构造器 4 5 -- 我们重用第 1.8 节中相同的线性令
牌 API 6 withToken :: (Token ⊸ Ur t) ⊸ Ur t 7 -- 回顾 Haskell 中 `
updWith` 的类型签名 8 updWith :: ∀ s t u. UAmpar s t ⊸ (t ⊸ u) ⊸
UAmpar s u 9 10 type DList t = UAmpar [t](UDest [t]) 11 12 newDList ::
Token ⊸ DList t 13 newDList = newUAmpar @[t]14 15 append :: DList t
 ⊸ t → DList t 16 dlist `追加` x =17 dlist `updWith` \d → 分情况 (d &fill
@'(:)) of 18 (dh, dt) → (dh &fillLeaf x) ° dt 19 20 concat :: DList t ⊸ DList t
 ⊸ DList t 21 dlist1 `concat` dlist2 = dlist1 `updWith` \dt1 → (dt1 &
fillComp dlist2) 22 23 toList :: DList t ⊸ Ur [t]24 toList dlist =
fromUAmpar' (dlist `updWith` \dt → dt &fill @'[])
```

代码清单 3.1：DPS Haskell 中差分列表的实现

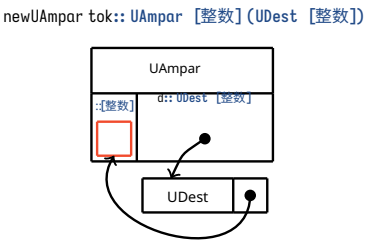


图 3.2：`newUAmpar tok` 在 DPS Haskell 中的内存表示

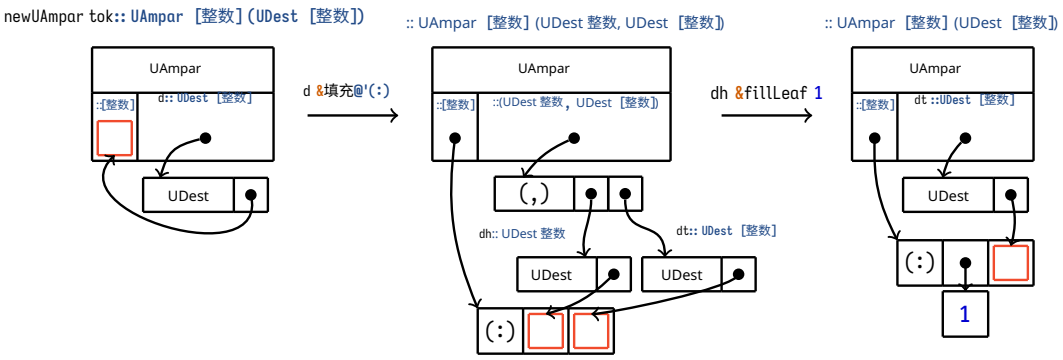


图3.3：DPS Haskell 中 追加 newUAmpar 1 的内存行为

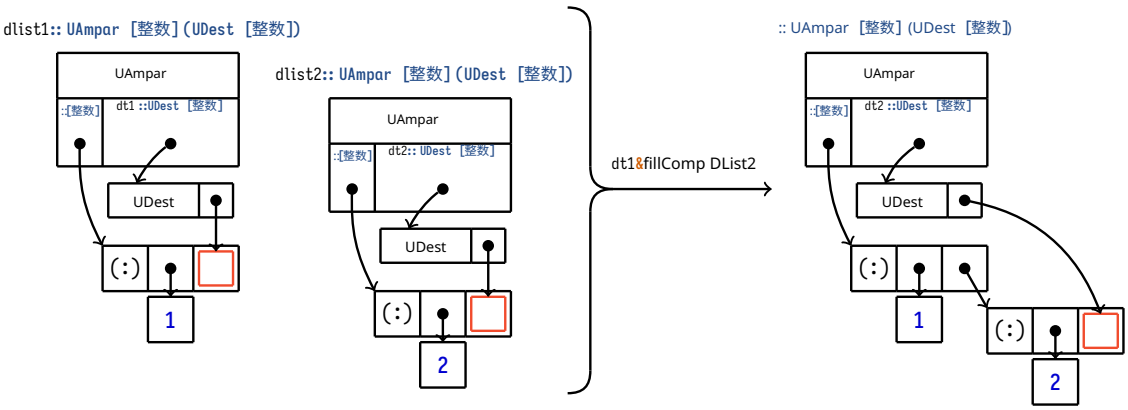


图 3.4：DPS Haskell 中concat dlist1 dlist2 的内存行为（基于fillComp）

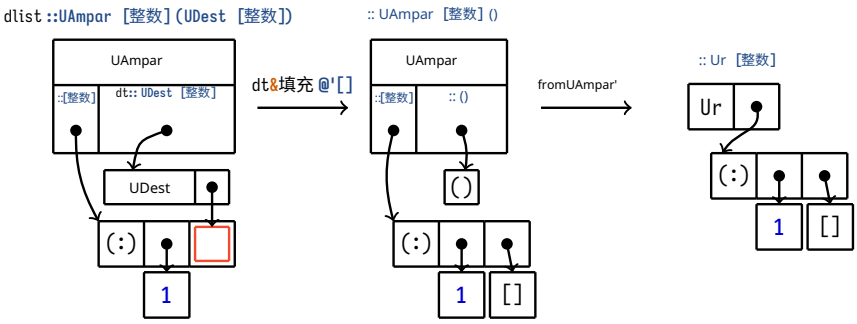


图 3.5：DPS Haskell 中 toList dlist 的内存行为

- 追加 （图3.3）会在差分列表的尾部添加一个元素。为此，它首先使用填充 `@'(:)`，用一个带有两个新空洞的空洞构造器填充由`d :: UDest [t]` 表示的列表末尾的空洞，这两个新空洞分别由`dh :: UDest t` 和`dt:: UDest [t]`指向。然后，`fillLeaf` 会用要追加的`t` 类型的值填充由`dh` 表示的空洞。生成的差分列表末尾的空洞由`dt ::UDest [t]` 指向；该空洞尚未填充，并保留在生成的 UAmpar 右侧。
- `concat` （图 3.4）将两个差分列表`dlist1` 和`dlist2`连接起来。它使用`fillComp` ，用第二个差分列表`dlist2`的根节点填充第一个差分列表的目标`dt1` 。因此，生成的`UAmpar` 对象与第一个列表具有相同的根节点，包含两个列表的元素，并继承第二个列表的空洞。从内存操作来看，`concat` 只需将第二个列表的地址写入第一个列表的空洞中。
- `toList` （图 3.5）通过使用 `fill @'[]`, 将一个新的 nil `[]` 构造器填入其空洞，从而完成 UAmpar 结构；随后，由于该结构现已完成，使用 `fromUAmpar'`移除 `UAmpar` 包装器。完成的列表被封装在 `Ur` 中，因为它可以被非线性地使用——这对 UAmpar 而言总是如此——并且如果需要，它还可以脱离由 `withToken` 创建的线性作用域。

通过线性令牌技术强制实现的 UAmpar 线性性至关重要[22]，因为当发生目标填充操作时，它能够安全地就地进行内存更新，如图 3.2 至图 3.5 所示。否则，我们可以先使用 `let Ur l = toList dlist`完成一个差分列表，然后使用 `append dlist x` 向 `dlist` 追加一个新的 cons 单元（实际上是第二次复用 `dlist` 内部的目标）。这会在 `l`内部创建一个空洞，尽管它的类型为 `[t]`，因此我们可以对其进行模式匹配，并且可能导致段错误！

本节将讨论这一实现（与差分列表的函数式编码相比）的性能，详见第 3.6.1 节。

3.3.2 广度优先树遍历

现在来看看广度优先树遍历。我们希望遍历一棵二叉树，并更新其节点值，使其按照广度优先顺序编号。

在第 2.4 节中，我们充分利用了 λ_d 的灵活性，使用队列来控制处理顺序，实现了广度优先遍历，这与我们在命令式编程语言中采用的做法相同。该队列按照广度优先顺序，包含输入子树和用于输出子树的目标。此外，我们使用队列和基于 ampar 的差分列表组成的对，高效地实现了该队列。

然而，在 DPS Haskell 中，目标不能用线性元素填充；因此，基于 ampar 的差分列表无法存储我们需要存储的目标（即表示输出子树的目标），相应地，基于 ampar 的高效队列也无法做到这一点。

因此，我们别无选择，只能回归 Haskell 中常规的、非基于 ampar 的数据结构，以构建合适的 Hood-Melville 队列[Hood 和 Melville， 1981]。由此可见，Haskell 中的目的地传递无法像在 λ_d 中那样完全取代构造器；数据构造器和目的地填充原语都必须共存[23]。Hood-Melville 队列的实现见清单 3.2。

[22]在 λ_d 中，我们可以选择对孔名称采用智能重命名技术（在本质上等同于写时复制），这样就不必以线性方式管理 ampar。不过，由于我们的目标是实现高效的实现，复制 ampar 并随时更新目的地的目标这一做法似乎并不可取。[23]只要我们能够用线性资源填充目标（如 λ_d所示），基于目标的数据结构构建就比基于构造器的结构构建更通用。当我们加入目标只能由非受限元素填充这一限制（正如 DPS Haskell 中的做法）时，基于目标的构建就不再比基于构造器的结构构建更通用。为了保持相同的表达能力，我们需要在语言中保留这两种构建方式。

```
1 data HMQueue t = HMQueue [t] [t]2 3 newHMQueue ::
HMQueue t 4 newHMQueue = HMQueue [] []5 6 singleton :: t ⊸
HMQueue t 7 singleton x = HMQueue [x] []8 9 toList :: HMQueue t
 ⊸ [t]10 toList (HMQueue front backRev) = front ++ reverse
backRev 11 12 enqueue :: HMQueue t ⊸ t ⊸ HMQueue t 13
enqueue(HMQueue front backRev) x = HMQueue front (x : backRev)
14 15 dequeue :: HMQueue t ⊸ Maybe (t, HMQueue t) 16
dequeue(HMQueue front backRev) = 分情况 front of17 [] → 分情
况 reverse backRev of18 [] → Nothing 19 (x : front') → Just (x,
HMQueue front' []) 20 (x : front') → Just (x, HMQueue front'
backRev)
```

清单 3.2：Haskell 中 Hood-Melville 队列的实现

与之前一样，对于广度优先遍历，我们会维护一个队列，其中存放由待重新标记的树和预期接收重新标记结果的目标组成的对，并在轮到每一项时对其进行处理。唯一的变化是队列的实现。广度优先树遍历的 DPS Haskell 实现见代码清单3.3。

除了选择不同的队列实现，以及必须从令牌创建 ampar 之外，该实现与第 2.4 节中的实现基本相同。在go 函数的类型签名中，线性箭头确保每个曾放入队列的目标最终都会在某个时刻被消耗，从而保证函数运行完毕后输出树是完整的。

正如我们将在第 3.6 节看到的，这种在 DPS Haskell 中以命令式风格实现的广度优先遍历，尽管没有使用差分列表等高效队列表示，与 Gibbons 等人的函数式实现相比，仍展现出显著的性能提升。 ⟦2023⟧, 我们必须承认，后者虽然相当优雅，但其设计并未将性能作为首要考量。

3.4 DPS Haskell API 与设计考量

清单 3.4 展示了 DPS Haskell 的纯 API。该 API 足以实现第 3.3 节中的所有示例。本节将详细说明它的各个部分，以及它与第 2 章演算的异同。

```
1 data Tree t = Nil | Node t (Tree t) (Tree t) 2 3 relabelDps:: Token ⊸ Tree t → Tree Int 4
relabelDps tree = fst (mapAccumBfs (\st _ → (st + 1, st)) 1 tree) 5 6 mapAccumBfs :: ∀ s t u. Token
⊸ (s → t → (s, u)) → s → Tree t → Ur (Tree u, s) 7 mapAccumBfs tok f s0 tree =8
fromUAmpar (newUAmpar @(Tree u) tok `updWith` \dtree → go s0 (singleton (Ur tree, dtree))) 9
where 10 go :: s → Queue (Ur (Tree t), UDest (Tree u)) ⊸ Ur s 11 go st q = 分情况 dequeue q of
12 Nothing → Ur st 13 Just ((utree, dtree), q') → 分情况 utree of 14 Ur Nil → dtree &填充 @'Nil ⨟
go st q' 15 Ur (Node x tl tr) → 分情况 (dtree &填充 @'Node) of 16 (dy, dtl, dtr) →17 let 表达式 q'' =
q' `入队` (Ur tl, dtl) `入队` (Ur tr, dtr) 18 (st', y) = f st x 19 in dy &填充Leaf y ⨟ go st' q''
```

代码清单3.3：DPS Haskell 中广度优先树遍历的实现

```
1 data Token
2 dup :: Token ⊸ (Token, Token) 3 丢弃:: Token ⊸ () 4 withToken :: ∀ t. (Token
⊸ Ur t) ⊸ Ur t 5 6 data UAmpar s t 7 newUAmpar :: ∀ s. Token ⊸ UAmpar s
(UDest s) 8 tokenBesides :: ∀ s t. UAmpar s t ⊸ (UAmpar s t, Token) 9 toUAmpar :: ∀
s. Token ⊸ s → UAmpar s () 10 fromUAmpar :: ∀ s t. UAmpar s (Ur t) ⊸ Ur (s, t) 11
fromUAmpar' :: ∀ s. UAmpar s () ⊸ Ur s 12 updWith :: ∀ s t u. UAmpar s t ⊸ (t ⊸
u) ⊸ UAmpar s u 13 14 data UDest t 15 type family UDestsOf lCtor t -- 返回与构造器字
段关联的目的地 16 fill :: ∀ lCtor t. UDest t ⊸ UDestsOf lCtor t 17 fillComp :: ∀ s t.
UAmpar s t ⊸ UDest s ⊸ t 18 fillLeaf :: ∀ t. t → UDest t ⊸ ()
```

清单 3.4：DPS Haskell API，初始版本

3.4.1 UAmpar 类型

与上一章中的 Ampar s () 一样，UAmpar 结构作为带孔结构的包装器，可以自由传递和存储，但在对其进行任何读取（即模式匹配）之前，必须先完成填充。因此，`UAmpar s t` 被定义为一种不透明数据类型，不提供公共构造器。

在 `UAmpar s t` 中，`s` 表示正在构建的结构的类型，而 `t` 则是读取该结构之前需要线性消耗的内容的类型。最终，在结构构建完成后，它将被封装在 `Ur` 中；这说明它完全由非受限元素构成。

`newUAmpar` 运算符是 API 用户新建 UAmpar 的主要方式。它的类型签名等同于第 2.9 节中的 `newWxIP`。

此外，有时采用 DPS 风格编程时，我们手头没有 `令牌`，只有一个现有的 `UAmpar`。在这种情况下，我们可以借助该现有 UAmpar 的线性性（因为它直接或间接线性依赖于一个令牌），获取一个新的线性 `令牌`，从而生成新的 UAmpar，或者实际上任何其他线性资源。这就是 `tokenBesides` 函数的目的。

在 DPS Haskell 中实现高效队列时（如第 2.2.3 节所示），`tokenBesides` 运算符尤其有用：

```
1 类型 DList t = UAmpar [t] (UDest [t]) 2 数据 EffQueue t =
EffQueue [t] (DList t) 3 4 newEffQueue:: 令牌 ⊸ EffQueue t 5
newEffQueue tok = EffQueue [] (newUAmpar @[t] tok) 6 7 出队 ::
EffQueue t ⊸ Maybe (t, EffQueue t) 8 出队 (EffQueue 前端 后端) =
分情况 front of9 [] → 分情况 (toList 后端) of 10 Ur [] →
Nothing 11 Ur (x : xs) → Just (x, (EffQueue xs
(newUAmpar @[t] tok))) 12 (x : xs) → Just (x, (EffQueue xs 后端))
```

在最内层 `分情况`的第二个分支中，当我们已将后端差分列表（此前我们从中写入数据）转换为新的前端列表（现在我们将从中读取数据）时，就必须新建一个后端差分列表，也就是一个新的 ampar。然而，我们手头没有令牌。对于 `出队` 这样的函数来说，要求传入一个线性令牌相当不切实际；更好的做法是，只在生成新的数据结构时请求令牌，而不是在对现有结构执行操作时请求。这正是`tokenBesides`派上用场的地方：我们可以改用现有 UAmpar 的线性要求：

```
1 tokenBesides :: ∀ s t. UAmpar s t ⊸ (UAmpar s t, 令牌) 2 3 dequeue ::
EffQueue t ⊸ Maybe (t, EffQueue t) 4 dequeue (EffQueue 前端 后端) = 分
情况前端 of5 [] → 分情况 tokenBesides 后端 of (后端, tok) → 分情况
(toList 后端) of 6 Ur [] → 丢弃 tok ; Nothing 7 Ur (x : xs) → Just (x,
(EffQueue xs (newDList tok))) 8 (x : xs) → Just (x, (EffQueue xs 后端))
```

不过，这个新版本也并不完美。我们甚至还不知道是否需要新令牌，就必须先创建一个新令牌。事实上，只有读取旧 UAmpar 的内容后，我们才能知道是否真的需要新令牌；而只有当它不再是 UAmpar 时，我们才能读取它……所以到那时，我们就无法再生成新令牌了。因此，解决办法是在使用旧 UAmpar 之前总是先生成一个新令牌，并承担这样的风险：如果队列确实为空，就必须立即丢弃这个新令牌（使用 `丢弃`），然后返回 `Nothing`。正如线性约束 [Spiwack, 2023a; Spiwack 等人 2022], 所提出的，隐式令牌传递似乎确实是彻底解决这类问题的正确方向。

从 UAmpar 中取出结构 与 λ_d 中相同，封装在 UAmpar 中的结构可以通过两种方式释放：使用 fromUAmpar' 时，t 侧的值必须是单位（()），并且只返回完整的 s，外面包裹着 Ur。使用 fromUAmpar 时，t 侧的类型必须是 Ur t' 的形式，此时返回一对 Ur (s, t')。回想一下，在 λ_d 中，我们有 $\mathrm{from}'_{\ltimes} : S \ltimes 1 \multimap S$ 和 $\mathrm{from}_{\ltimes} : S \ltimes (!_{1_\infty} T) \multimap (S \otimes !_{1_\infty} T)$，其中从 ampar 中取出的左侧部分没有包裹任何模态。

在 DPS Haskell 中，用 `Ur` 包裹已构建的结构实际上是安全的，因为正如前面所说，它的叶节点要么来自非线性来源（例如 `fillLeaf ::t → UDest t ⊸ ()` 会以非线性方式消耗它的第一个参数），要么由使用 `fill` 添加的零元构造器构成；这两种情况都可以安全地以无限制的方式使用；而它的脊柱由空心构造器组成，可以随意复制。

3.4.2 目的地填充函数

API 的最后一部分负责以自顶向下的方式实际构建结构。要填充由 `UDest t` 表示的空洞，可以使用以下三个函数：

`fillLeaf :: ∀ t. t → UDest t ⊸ ()` 使用类型为 `t` 的值来填充目标所表示的空洞，如 ◀ from λ_d所示。然而，如果目标是以线性方式消耗的，如 λ_d所示，则用于填充空洞的值并非如此（第一条非线性箭头已表明这一点）。这是 UAmpar 只能容纳不受限数据的关键所在。从内存角度看，使用 `fillLeaf` 时，类型为 `t` 的对象地址会被写入由类型为 `UDest t` 的目标指向的内存单元中（见图 3.8）。

`fillComp :: ∀ s t. UAmpar s t ⊸ UDest s ⊸ t` 用于将两个 `UAmpar` 对象连接起来。与类似的运算符 ◁∘ from λ_d 一样，函数的类型签名中并未表示父级 `UAmpar`。相反，类型签名只通过 `UDest s` 表示父级 UAmpar 中用于接收子级 UAmpar 地址的空洞；而类型签名中的 `UAmpar s t` 指的是子级 UAmpar。调用 `fillComp` 时，始终处于对父级执行 `` `updWith` `` 的作用域内（因为这是获取构成 `fillComp`左侧操作数的目标的唯一方式）：

```
1 父级:: UAmpar BigStruct (UDest SmallStruct, UDest OtherStruct)2 子
级:: UAmpar SmallStruct (UDest Int)3 组合 = 父级 `updWith` \(ds,
do) → (ds &fillComp 子级, do)4 :: UAmpar BigStruct (UDest Int,
UDest OtherStruct)
```

最终得到的结构 `组合` 从本质上说是一个类似于 `BigStruct` 的 `父级`，它从子结构继承了空洞（`UDest Int`），并且仍有另一个待填充的空洞（`UDest OtherStruct`）。图 3.4 展示了 `fillComp` 的内存行为示例。

最后，`fill :: ∀ lCtor t. UDest t ⊸ UDestsOf lCtor t` 是一个用于用空心构造器填充目标的通用函数。它将一个数据构造器作为类型参数（`lCtor`），并分配一个对应的空心构造器——也就是说，一个具有与指定构造器相同头部、但字段未指定的堆对象

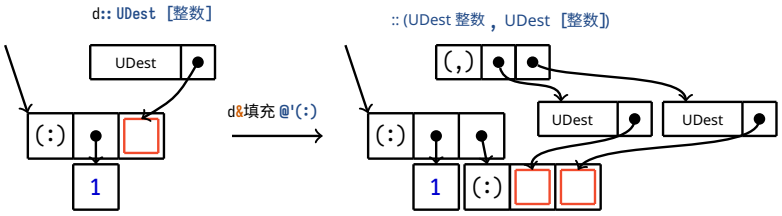


图 3.6：内存行为 填充 @'(:) :: UDest [t] ⊸ (UDest t, UDest [t])

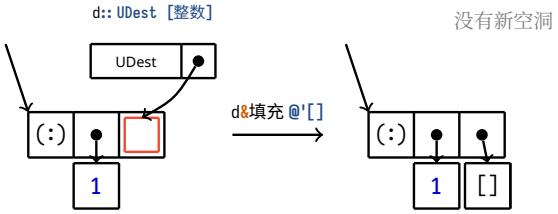


图 3.7： 填充 @'[] :: UDest [t] ⊸ ()

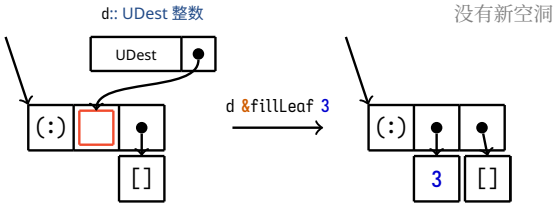


图 3.8：fillLeaf 的内存行为 fillLeaf :: t → UDest [t] ⊸ ()

——并为该构造器的每个字段返回一个新目标。已分配的空心构造器的地址会写入提供给fill的目标中。`fill @'(:) :: UDest [t] ⊸ (UDest t, UDest [t])` 的内存行为示例见图 3.6，而`fill @'[]:: UDest [t] ⊸ ()` 的内存行为示例见图 3.7。

`fill` 函数的输出由一个个目标组成，每个目标的类型都与指定构造器的一个字段相匹配；在 Haskell 中，这由类型 `UDestsOf lCtor t` 表示。`UDestsOf` 是一个类型族（即从类型到类型的函数），其作用是将一个构造器（提升为类型）映射为其字段对应的目标类型。例如，`UDestsOf '[] [t] = ()` 和 `UDestsOf '(:) [t] = (UDest t, UDest [t])`。更一般地说，`UDestsOf` 类型类体现了构造器字段类型与空心构造器目标类型之间的对偶性，这一点在第 2.2.1 节中已首次提及。

3.5 紧凑区域：实现 DPS Haskell 的安全空间

内存中存在不完整结构，天然会与垃圾回收器和编译器产生诸多冲突。事实上，GHC 的垃圾回收器假设它遍历的每个堆对象都是格式良好的，而 UAmpar 结构却完全不符合这一要求：它们包含未初始化指针，而垃圾回收器绝对不应跟随这些指针。此外，编译器和垃圾回收器可以进行某些优化，因为它们假设每个对象都是不可变的；然而，DPS 编程会在构造器分配后对其进行变异，从而破坏这一保证（尽管每个构造器最多只能进行一次更新）。

因此，我们寻找了一种替代性的内存管理方案，该方案基本独立于垃圾回收器，使我们能够实现 DPS Haskell API，而无需修改垃圾回收器的内部机制或 GHC 使用的内存表示。

3.5.1 紧凑区域

Yang 等人提出的紧凑区域 [2015] 是 Haskell 的特殊内存*arena*。紧凑区域表示 Haskell 堆中的一块内存区域，它几乎完全独立于垃圾回收器和垃圾回收堆的其余部分。对于垃圾回收器而言，每个紧凑区域都被视为一个具有单一生命周期的堆对象。垃圾回收器可以高效地检查垃圾回收堆中是否至少有一个指针指向该区域；只要存在这样的指针，该区域就会保持存活。一旦不再满足这一条件，整个区域就会被丢弃。这样，垃圾回收器便不会遍历该区域中的任何节点：该区域会被视为一个不透明的块（尽管实际上它是由一串大小相同的块实现的，但这并不影响原理）。此外，紧凑区域在内存中不可移动；垃圾回收器不会移动它们，因此指向紧凑区域中空洞的目标可以直接实现为原始指针（在 Haskell 中，类型为`Addr#`）：`data UDest r t = UDest Addr#`，因为我们可以保证包含该空洞的 UAmpar 不会移动。

通过使用紧凑区域实现 DPS 编程，我们完全避开了刚才提到的垃圾回收器与 UAmpar 结构之间的张力问题。反过来，我们的目标传递 API 也为使用紧凑区域提供了一种全新而强大的方式。

作为这项实现决策的代价，我们多了两项限制。首先，区域中的每个结构都必须处于完全求值形式。这与 Haskell 通常采用的求值方案形成对比：在该方案中，默认情况下所有内容都是惰性求值的。因此，由于区域是严格求值的，复制到区域中的任何堆对象都会先被强制求值至范式（也就是说，它会被完全求值）。这并不总是有利的；有时惰性求值能带来更好的性能。

其次，区域中的数据不能包含指向垃圾回收堆或其他区域的指针：它必须是自包含的。这迫使我们对 API 稍作修改，添加一个幻影类型参数 `r`，用它为每个对象标记其所属区域的标识符，从而使我们的 API 无需运行时检查也能保证内存安全。由此带来两个相关后果：首先，将垃圾回收堆中的值用作 `fillLeaf` 的参数时，必须先将其完全求值并复制到区域中，而不能只进行指针更新；其次，`fillComp` 只能将来自同一区域的两个 `UAmpar` 拼接在一起。

还需要一个类型类 `Region r`，用于携带实现区域所需的详细信息。该类型类只有一个用户无法使用的方法 `reflect`，用于返回与标识符 `r`关联的 `RegionInfo` 结构。

`inRegion` 函数是列表 3.5 所示修改后 API 中新增的内容。它接收一个任意类型的表达式，其中 `r` 必须是自由类型变量。在内部，使用该函数时，它会生成一个新的紧凑区域和一个刚性类型变量 `r`（其行为类似于一个新鲜的、具体的、不透明的类型），并使用reflect 库即时提供一个 `Region r` 实例，将 `r` 与新区域的 `RegionInfo` 关联起来（通过 `reflect` 函数）；随后，在计算 `inRegion` 的主体时，将 `r` 替换为 `r`，这样对 `reflect @r` 的任何调用都会变成 `reflect @r`，并返回正确的 `RegionInfo`。实际上，`inRegion` 是一种作用域函数（尽管这一次它不是线性的）。

```
data Token
dup :: 令牌 ⊸ (令牌, 令牌)
丢弃 :: 令牌 ⊸ ()
withToken :: ∀ t. (令牌 ⊸ Ur t) ⊸ Ur t

type Region r :: Constraint
inRegion :: ∀ t. (∀ r. 区域 r ⇒ t) ⊸ t

data UAmpar r s t
newUAmpar :: ∀ r s. 区域 r ⇒ 令牌 ⊸ UAmpar s (UDest r s)
tokenBesides :: ∀ r s t. 区域 r ⇒ UAmpar r s t ⊸ (UAmpar r s t, 令牌)
toUAmpar :: ∀ r s. 区域 r ⇒ 令牌 ⊸ s → UAmpar r s ()
fromUAmpar :: ∀ r s t. 区域 r ⇒ UAmpar r s (Ur t) ⊸ Ur (s, t)
fromUAmpar' :: ∀ r s. 区域 r ⇒ UAmpar r s () ⊸ Ur s
updWith :: ∀ r s t u. 区域 r ⇒ UAmpar r s t ⊸ (t ⊸ u) ⊸ UAmpar r s u

data UDest r t
类型族 UDestsOf lCtor r t -- 返回与构造器字段相关联的目标位置
fill :: ∀ lCtor r t. 区域 r ⇒ UDest r t ⊸ UDestsOf lCtor r t
fillComp :: ∀ r s t. 区域 r ⇒ UAmpar r s t ⊸ UDest r s ⊸ t
fillLeaf :: ∀ r t。区域 r ⇒ t → UDest r t ⊸ ()
```

列表 3.5：紧凑区域的 DPS Haskell API

3.5.2 紧凑区域中的实用 DPS 编程

现在，我们将研究一个与实现问题相关的示例，在这个示例中，DPS 编程真正大显身手（而紧凑区域也非常适用）。

在客户端-服务器应用中，以下模式非常常见：服务器接收来自客户端的请求及序列化负载，然后反序列化该负载、运行一些代码，并响应请求。大多数情况下，反序列化后的负载会在整个请求处理期间保持存活。在垃圾回收语言中，这会带来实际开销：垃圾回收器（GC）会一次又一次地遍历反序列化后的负载，尽管我们知道其所有内部指针在请求处理期间都保持存活。

相反，我们更希望将反序列化后的负载视为单个堆对象，无需对其进行遍历，并且可以作为一个块释放。事实上，紧凑区域非常适合这一任务，因为 GC 永远不会跟随指针进入紧凑区域，并且会将每个区域视为其中所有内容具有相同的生命周期。

如果我们直接使用紧凑区域和 GHC 提供的 API，就会先在垃圾回收堆中正常反序列化有效载荷，然后将其复制到紧凑区域中，并且只保留对该副本的引用。这样，垃圾回收器就不会跟踪区域副本中的内部指针；之后，该副本会作为一个整体被回收，而垃圾回收堆中的原始对象则会立即被回收。

然而，我们仍然为反序列化后的有效载荷分配了两份副本（一份在垃圾回收堆中，另一份在复制过程中分配到区域中）。这很浪费；直接在区域中分配会好得多。幸运的是，借助我们对采用紧凑区域的 DPS Haskell 的实现，现在我们有了一种更好的方法，可以直接在这些紧凑区域中分配并构建结构！

```
parseSList:: 字节字符串 → 整数 → [SExpr] → 错误或结果类型
SExpr parseSList bs i acc = 分情况 bs !? iof Nothing → Left (S列表意
外文件结束 i) Just x → if | x == ')' → Right (SList i (reverse
acc)) | isSpace x → parseSList bs (i + 1) acc |
otherwise → 分情况 parseSExpr bs i of Left err → Left err Right
child → parseSList bs (endPos child + 1) (child : acc)
```

清单 3.6：Haskell 中 S 表达式解析器的实现（不含目的地）

```
parseSListDPS ::字节字符串 → 整数 → UDest [SExpr] ⊸ 错误或结果
类型 Int parseSListDPS bs i d = 分情况 bs !? i of Nothing → d &fill @'[]
Left (S列表意外文件结束 i) Just x → if | x == ')' → d &fill @'[]
Right i | isSpace x → parseSListDPS bs (i + 1) d | otherwise →
分情况 d &fill @'(:) of → 分情况 parseSExprDps bs i dh of Left
err → dt &fill @'[] Left err Right endPos → parseSListDPS bs (endPos
+ 1) dt
```

清单 3.7：DPS Haskell 中 S 表达式解析器的实现

来看看，如何为 S 表达式（用于表示我们的请求负载）解析器使用目标和紧凑区域，从而提升性能。S 表达式是由空格分隔元素的带括号列表。这些元素可以是多种类型：整数、字符串、符号（不带引号的文本标记），或其他 S 表达式组成的列表。

解析 S 表达式可以通过相互递归函数以一种朴素的方式完成：

- `parseSExpr` 扫描下一个字符；如果遇到左括号，就分派到`parseSList`；如果遇到左引号，就分派到`parseSString`；否则，最终会将字符串解析为数字或符号；

- `parseSList` 调用`parseSExpr` 解析下一个令牌，然后再次调用自身，直到遇到右括号，并在此过程中累积解析出的元素。

这里只展示`parseSList` 的实现，因为这已足以满足我们的目的；不过，整个解析器的朴素版本和基于目的地的版本的完整实现，可在 [Bagrel, 2023b]的`src/Compact/Pure/SExpr.hs` 中找到。

清单 3.6 中给出的实现相当标准：累加器 `acc` 以逆序收集由 `parseSExpr` 返回的节点（因为在没有目的地的情况下，这是构建链表的自然顺序）。到达列表末尾时（第 5 行），累加器会被反转，包装在 `SList` 构造器中，然后返回。我们还会在解析结果中存储索引 `i`，它对应于输入字符串中该结构的结束位置。

我们将看到，只需对代码进行非常少的风格调整，目的地就能带来非常显著的性能提升。只需将尾递归函数的累加器改为目的地即可。与之前一样，将元素写入一个最终需要反转的列表不同，采用目的地风格的程序会直接按正确顺序将元素写入其最终位置！

清单 3.7 展示了 parseSListDPS 的代码。让我们看看它与朴素实现相比有哪些变化：

- 即使在错误情况下，我们也必须使用作为参数接收的目标（以保持线性），因此会向其中写入一些合理的默认数据（见第 3 行）；
- parseSExprDps 产生的 SExpr 值不会被parseSListDPS 收集，而是由 parseSExprDps 通过传递并填充目标 dh，直接写入其最终位置（见第 9 行）；
- 将一个 SExpr 类型的元素添加到累加器 [SExpr] 的操作，改为在UDest [SExpr], 表示的空洞中，使用 fill @'(:) 写入一个新建的 cons 单元；将一个元素写入新建的*head*目标，然后以新建的 *tail* 目标作为参数进行递归调用（其类型同样为 UDest [SExpr]）；
- 由于列表是以自顶向下的方式构建的，因此无需在处理结束时反转列表并返回累加器；只需将一个 Nil 元素写入尾部目标来完成列表即可（使用 填充 @'[], ，见第 5 行）；
- 归根结底，目的地传递风格函数只返回下一个待读取字符的偏移量，而不是返回解析后的值（因为解析后的值会直接写入作为参数接收的目标中）。

得益于这一新实现，其代码行数仅比朴素实现略多，程序的运行速度几乎提高了一倍，这主要是因为垃圾回收时间几乎降为零。详细的基准测试见第 3.6 节。

由此可见，紧凑区域与目标传递风格相辅相成：紧凑区域让 DPSHaskell API 易于实现，而目标传递风格则让紧凑区域的使用更加高效、灵活。

3.5.3 紧凑区域中 UAmpar 的内存表示

如我们在第 2.6.1 节和第 3.3.1 节中所述，我们希望 UAmpar r s t 包含一个类型为 s 的值和一个类型为 t的值，并在类型为 t 的值被完全消耗（或线性转换为 Ur t'）后，释放类型为 s 的值。

我们还知道，newUAmpar 返回一个 UAmparr s (UDests)：这里实际上只有一个空的内存单元，它将承载未来的类型为 s的结构的根节点；与之关联的类型为 UDest s 的目标指向该根节点，如图 3.2 所示。正如我们之前所说，放入目标中的内容，正是随后将在 s 一侧取回的内容。

最直观、最直接的想法，是将 UAmpar r s t 在内存中表示为一个类对构造器，其中一个字段的类型为 s，另一个字段的类型为 t；第一个字段用于存放正在构建的结构的根节点（如图 3.2 所示）。然而，根洞稍后将用于存放类型为 s的结构，它可能包含垃圾数据，因此必须避免垃圾回收器读取它并追踪其中未初始化的指针（否则可能导致段错误）。在我们的紧凑区域实现中，可以让根洞位于紧凑区域内，从而避免这个问题（这样垃圾回收器就不会查看其中的内容）。但我们还希望 UAmpar构造器位于垃圾回收堆中，以便编译器有时可以将其优化掉，并且能尽早将其释放。这两个要求显然互不相容，因为在这种表示中，根洞对应于 UAmpar 构造器的第一个字段。

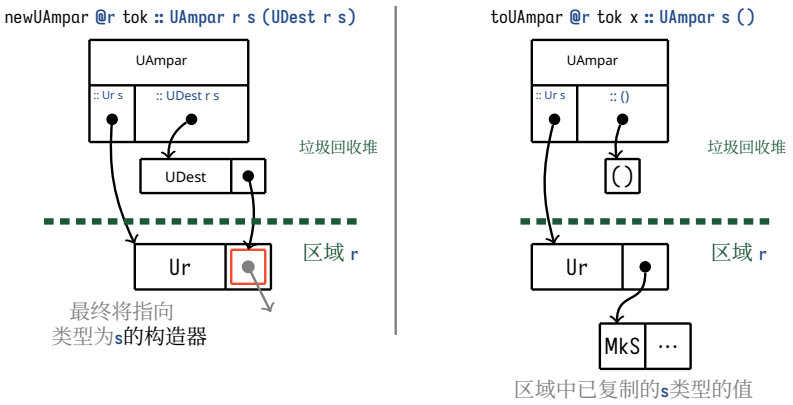


图 3.9：使用UAmpar r Ur 的表示（未采用）

鉴于 UAmpar 中保存的是不受限数据，一种可能的变通方案是将 UAmpar r s t在内存中表示为一个构造器，其中一个字段的类型为 Ur s ，另一个字段的类型为 t （而不是此前的 s 和 t ）。这意味着我们需要在 UAmpar 包装器与正在构建的类型为 s 的结构之间加入一个中间的 Ur 构造器。我们可以选择在区域内分配这个 Ur 构造器，并让其类型为 s的字段也位于区域内，作为根洞。如图 3.9 所示，采用这种方法，根洞不会被垃圾回收器读取，而 UAmpar 包装器则可以毫无问题地位于垃圾回收堆中，并像我们希望的那样尽早丢弃！此外，fromUAmpar'的实现也会非常高效，因为完整 UAmpar 的第一个字段就是类型为 Ur s 的值，也正是 fromUAmpar' 应当返回的值（不过，这对 fromUAmpar 没有帮助）。其缺点是，每个 UAmpar 都会在区域中分配几个字（用于容纳 Ur 构造器[24]），即使其父级 UAmpar 已被回收，这些字也会在很长时间内无法被垃圾回收器回收。具体而言，这会使toUAmpar 在内存使用方面相当低效，因为它必须在区域中分配一个 Ur 包装器，而对于已经完整的结构来说，这个包装器毫无用处（因为完整结构根本没有空洞，因此不需要在紧凑区域中设置根洞）。

我们在实际实现中决定采用的另一种选项，是仅为使用newUAmpar创建的真正不完整结构，在紧凑区域内分配一个对象来容纳该结构的根洞。对于已经完整、通过toUAmpar转换为UAmpar 的结构，我们跳过此分配；但在这两种情况下（newUAmpar 和toUAmpar），我们都保留 UAmpar 字段的相同底层类型。这是通过在newUAmpar返回的 UAmpar 中使用特殊的间接对象实现的（在 GHC 代码库中为stg_IND ）。stg_IND 对象是一种单字段构造器，虽然其字段的类型也是s，但它被视为具有s 类型。

我们在图 3.10 中对此进行了说明：

- 在newUAmpar返回的类对结构中，s 一侧指向一个间接对象，该对象分配在区域中，并充当不完整结构的根洞（由于它位于紧凑区域中，垃圾回收器此刻不会跟踪其中可能包含的垃圾指针）；

- 在 toUAmpar 返回的类对结构中，s 一侧直接指向已复制到区域中的 s 类型对象，无需间接寻址。

[24]有人可能会问，为什么Ur 不能是一个零成本的newtype 包装器。首先，如果Ur 是newtype，它就无法实现我们在此处的目标，因为我们依赖于它在内存中创建间接寻址这一特性。其次，也是更重要的是，这背后还有语义方面的动机，详见 [Spiwack, 2024]。

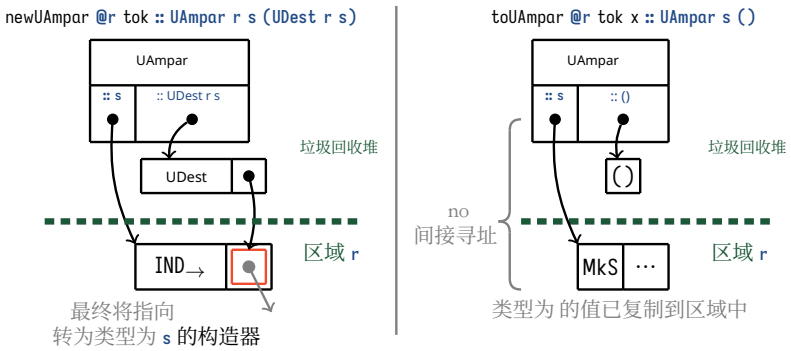


图 3.10：使用间接引用的UAmpar r 表示

与之前的方案相比，使用 toUAmpar 时，这种方案效率更高（因为不会产生长期存在的垃圾），但使用 fromUAmpar' 时不再带来效率提升（因为我们不再生成 Ur 对象）。这就是我们在原始工件 ⟦Bagrel, 2023b⟧ 中采用的方案。

在 ⟦Bagrel, 2023b⟧，中，我们因此为 API 类型定义了以下私有定义[25]：

```
data Token = Token -- 与 () 表示相同
data UAmpar r s t  =UAmpar s t -- 与 (s, t) 表示相同，
                                -- 有时 s 字段会带有间接寻址
data UDest r t  = UDestAddr# -- 装箱的 Addr#，与 Ptr t 表示相同
```

3.5.4 推导填充所有构造器的 Generics

在 λ_d 中，为每个对应的数据构造器提供专用的目标填充函数相当容易。事实上，我们只需处理有限个构造器（单位类型对应一个，和类型对应两个，对应一个，对指数模态对应一个；而函数对应的填充函数则完全可以省略）。

然而，在 Haskell 中，我们希望能够对任意数据类型（包括用户自定义类型）进行目的地传递，因此，为每个可能的数据构造器手动实现一个填充函数并不可行。因此，我们必须将所有填充函数泛化为一个多态的 fill 函数，使其能够处理任意数据构造器。

为此，我们创建一个类型类 Fill lCtor t，并将 fill 定义为该类唯一的方法。这样的类型类使我们能够将 fill 的行为与类型参数 lCtor 关联起来。其思路是，lCtor 表示我们希望与 fill 配合使用的数据构造器的类型级名称。调用处的语法形式为 fill @'Ctor：我们先用 ' 运算符将构造器 Ctor 提升为类型级名称，然后通过 @进行类型应用。需要回顾的是，fill 函数应将指定变体的新建空心构造器填入结构中的空洞，而该结构对应的目标作为参数传入。

现在来看看，如何利用构造器 lCtor 的类型级名称及其基类型 t，获取编写相应填充函数的具体实现所需的全部信息。首先，我们需要了解构造器的形态，更确切地说，是它的字段数量和类型，因为链式数据类型的每个数据构造器都可以视为一个 n 元积构造器。因此，我们将借助 GHC。Generics 来查找所需信息。

[25]也就是说，对于 API 的使用者而言，它们是不透明的。

GHC。Generics 是 Haskell 内置的库，可通过Generic 类型类在编译时检查类型定义，例如构造器列表、字段、内存表示等。而且，几乎任何类型都可以自动派生该类型类！

例如，下面是Generic 表示 Maybe t：

```
repl> :k! Rep(Maybe a) () -- 显示 Maybe a 的 Generic 表示
M1 D(MetaData "Maybe" "GHC.Maybe" "base" False) (
  M1 C(MetaCons "Nothing" PrefixI False) U1
  :+: M1 C(MetaCons "Just" PrefixI False) (M1 S [...] (K1 R t)))
```

我们可以看到，这里有两个不同的构造器（由 M1 C ... 行表示）：Nothing 没有字段（由 U1表示），而Just有一个类型为t 的字段（由 K1 R t表示）。

如上例所示，类型的 Generic 表示包含了我们想要了解的有关其构造器的大部分信息。因此，借助类型级编程技术[26]，，特别是类型族，我们可以提取该 Generic 表示中与我们相关的部分（作为类型级表达式），并在类型类实例的头部引用它们，从而为fill提供多个专门实现。例如，在 ⟦Bagrel, 2023b⟧,中，对于lCtor中从 0 到 7^{27} 的每一种可能字段数，我们都为fill 提供了一个专门实现。主要原因是，fill 应返回一个元组，其中包含的目标数量应与所选构造器的字段数量相同；而目前，Haskell 尚无法对元组长度进行抽象，因此我们需要对每种可能的情况分别硬编码。

将 UDestsOf lCtor t 类型族解析为具体的目的地元组类型，方法也类似：根据基类型 t 的 Generic 表示，通过类型级表达式完成；lCtor 属于该基类型。

不过，此时我们仍缺少一个重要环节：用于在紧凑区域中分配适当空心构造器对象的 Haskell 内部机制。

3.5.5 GHC 内部机制及其运行时系统的变更

下面我们将介绍如何分配一个空心构造器，也就是为给定构造器分配一个空心堆对象。不过，首先让我们稍作绕行，进一步了解编译器内部机制的背景。

Haskell 的运行时系统（RTS）由 C语言和 C-- 混合编写 ⟦Ramsey、Jones 和 Lindig, 2005⟧。RTS 有许多职责，包括管理线程、组织垃圾回收以及管理紧凑区域。它还定义了各种原语操作，称为 外部原语操作，这些操作将 RTS 的能力作为普通函数提供出来，以便在 Haskell 代码中使用。尽管 RTS 承担了所有这些职责，但令人意外的是，它并不负责分配普通构造器（在垃圾回收堆中构建）。原因之一是，它并不具备构建构造器堆对象所需的全部信息，也就是与构造器关联的信息表。

在 Haskell 中，堆对象是内存中的一段数据，对应于数据构造器的某个给定实例，或某个（部分应用的）函数。信息表定义了堆对象的布局和行为。表示同一数据构造器（例如 Just）的所有堆对象都具有相同的信息表，该信息表充当该构造器的恒等卡片，即使相关类型不同（例如 Just x :: Maybe Int 和 Just y:: Maybe Bool 共享同一个信息表）。在编译过程中，

[26]参见 src/Compact/Pure/Internal.hs: 418 ⟦ 2023 ⟧在 Bagrel 中，我们可以设得更高；

[27] 7 只是字段数量的任意上限，在标准Haskell 库中相当常见。

在代码生成期间，`Just` 构造器的所有实例都会持有标签 `Just_con_info`，该标签随后由链接器解析为指向共享信息表的实际指针，而该构造器对应的所有堆对象都会携带这一指针。

另一方面，运行时系统（RTS）是一段静态代码，在构建 GHC 时编译一次，之后会链接到 GHC 构建的每个可执行文件中。只有在程序启动时，运行时系统才会运行（以监督程序的执行），因此它无法直接访问程序编译期间生成的信息。具体来说，它无法检查信息表：标签早已被实际指针取代，而这些指针在内存中与其他任何数据都无法区分。但另一方面，运行时系统知道如何在紧凑区域内为我们的新空心构造器分配空间。

因此，我们需要一种方式将信息，更准确地说，将信息表指针从编译时传递到运行时。为此，我们通过两个新的基本操作来管理在紧凑区域中创建新的空心构造器：

- 一个内部原语操作，它会在编译时将数据构造器解析为表示其信息表指针的（静态）值。

- 一个在运行时系统（RTS）中的外部原语操作，它会在紧凑区域内为一个空心构造器分配空间，并将该空心构造器的信息表指针设为开发者可通过内部原语操作提供的参数值。

此处将展示的所有 GHC 修改，其完整版本均见 ⟦Bagrel, 2023a⟧。

内部原语操作：获取构造器的信息表指针作为静态值 内部原语操作是一些宏，会在编译时转换为 C- - 代码。因此，对于我们这个用于获取构造器信息表标签（之后会解析为指针）的新内部原语操作，必须将所选构造器指定为类型级参数，以便在编译时对其进行检查（因为显然，项级值无法在编译时轻易检查）。为此，我们在构造器名称前使用*tick* 运算符 `'`，它会将一个对象从项级命名空间（数据构造器位于其中）提升到类型级，使其可用于通常需要类型的地方[28]。

现在，我们需要借助内部原语操作来检查表示所选构造器的类型参数。通常，在 Haskell 中，我们会使用类型为 `Proxy t`（带有幻影类型参数 `t` 的单位类型）的参数，作为我们想要传递给函数的类型级输入的载体。遗憾的是，原语操作是特殊的函数，而且由于编译器的一个特性，它们在解析阶段已无法访问其参数的类型，因此，如果我们为原语操作设计一个带有 `Proxy lCtor` 参数的形式，它就无法读取传入的 `lCtor`。然而，原语操作却能出人意料地访问其返回类型。因此，我们为原语操作设计一个自定义返回类型 `InfoPtrPlaceholder#`，用来携带原语操作的输出值（也就是信息表指针）；同时，我们还为 `InfoPtrPlaceholder#` 添加一个幻影类型参数 `lCtor`。该参数与底层值无关，只是作为载体，承载表示所选构造器的类型级输入。

具体实现总结于清单 3.8 中（我们省略了散布在 GHC 代码库其他部分的大约一百行样板代码，这些代码负责处理原语操作注册、接口声明等）。这一实现比我们刚才描述的内容更进一步，因为`reifyInfoPtr#` 不仅能够获取数据构造器的信息表指针（正如前文所述），还能够获取特殊编译器对象（例如我们在第 3.5.3 节中使用的间接对象）的信息表指针。该原语操作会根据其返回值的类型 `resTy` 进行模式匹配（该类型应当采用 `InfoPtrPlaceholder# lCtorOrSym` 的形式）。

[28]关于如何在 Haskell 中将项级对象提升为类型级对象，更多信息请参阅 ⟦Yorgey 等人， 2012⟧。

```
1 分情况 primop of2       [...]3 ReifyInfoPtrOp  → \_ → -- 我们不关心函数参数 (# #) 4
opIntoRegsTy $ \[res] resTy → emitAssign (CmmLocal res) $ 分情况 resTy of 5 -- 当幻影类型参数是
提升的数据构造器时，将其提取为数据构造器 6 TyConApp _infoPtrPlaceholderTyCon [
_typeParamKind,TyConApp tyCon _]7              | Just dataCon ← isPromotedDataCon_maybe
tyCon →8 CmmLit (CmmLabel ( 9 mkConInfoTableLabel (dataConName dataCon) DefinitionSite))
10 -- 当幻影类型参数是 Symbol 时，提取符号值并存入 'sym' 11 TyConApp _infoPtrPlaceholderTyCon [
_typeParamKind, LitTy (StrTyLit sym)] →12 CmmLit (CmmLabel ( 13 mkCmmInfoLabel rtsUnitId
(fsLit "stg_" `追加FS` sym))) 14 _ → [...] -- 没有模式匹配时出错
```

代码清单 3.8： reifyInfoPtr# 在 compiler/GHC/StgToCmm/Prim.hs 中的实现

如果它读取的是一个提升的数据构造器（这是最常见的情况），就会将原语操作调用解析为标签 `<构造器>_con_info`，该标签对应于该构造器的信息表指针。如果`lCtorOrSym`是一个符号（即类型级字符串字面量），就会将原语操作调用解析为标签 `stg_<符号>`（例如，如前所述，用于分配一个 `stg_IND` 对象）。返回的 `InfoPtrPlaceholder# lCtorOrSym`之后可以使用 `unsafeCoerceAddr` 函数转换回 `Addr#`。尽管名称如此，这种用法并不涉及不安全的强制转换。

外部原语操作：在区域中分配一个空心构造器 作为运行时系统的一部分，我们的外部原语操作直接用 C语言编写。其实现见代码清单 3.9，名称为 `stg_compactAddHollowzh`；而在 Haskell 代码中，我们通过名称 `compactAddHollow#`访问它。该原语操作主要是对 `ALLOCATE` 宏的一次增强调用。此宏定义在 `Compact.cmm` 文件中；如果空间足够，它会尝试在紧凑区域的当前块中进行指针递增分配，否则会向该区域添加一个新块。接着，它会获取待分配构造器的信息表指针（作为第二个参数 `W_ info` 传入，因为它无法自行获取此信息），并使用 `SET_HDR`将其写入堆对象的第一个字。然后，将指向新分配的空心构造器的指针设为返回值。

组合两个原语操作以获取一个新的空心构造器 现在是展示这两个原语操作如何发挥作用的时候了。下面以在紧凑区域中分配一个空心 `Just` 为例：

```
1 hollowJust:: Maybe a  = compactAddHollow#2      区域3 (unsafeCoerceAddr
(reifyInfoPtr# (# #):: InfoPtrPlaceholder# 'Just ))
```

需要注意的是，表达式 `reifyInfoPtr# (# #)` 会在编译时被替换为一个静态值，该值表示 `Just` 的信息表指针；而对 `compactAddHollow#` 的调用则与普通函数调用一样，只有在程序运行时才会执行。

```
1 // compactAddHollow# 2 // ::Compact# → Addr# → State# Real
World → (# State# RealWorld, a #) 3 stg_compactAddHollowzh(P_ compact,
W_ info) { 4 W_ pp, ptrs, nptrs, size, tag, hp; 5 P_ to, p; p = NULL; // p 实际上不
会被 ALLOCATE 宏使用 6 again: MAYBE_GC(again); STK_CHK_GEN(); 7 8 pp =
compact + SIZEOF_StgHeader + OFFSET_StgCompactNFData_result; 9 ptrs =
TO_W_(%INFO_PTRS(%STD_INFO(info))); 10 nptrs =TO_W_(%INFO_NPTRS(%
STD_INFO(info))); 11 size = BYTES_TO_WDS(SIZEOF_StgHeader) + ptrs +
nptrs; 12 13 ALLOCATE(compact, size, p, to, tag); 14 P_[pp] =to; 15
SET_HDR(to, info, CCS_SYSTEM); 16 #if defined(DEBUG) 17 调用
verifyCompact(compact); 18 #endif 19 return (P_[pp]); 20 }
```

代码清单 3.9： compactAddHollow# 在 rts/Compact.cmm 中的实现

用于从提升构造器转换为关联符号的内置类型族 我们刚引入的内部原语操作 `reifyInfoPtr#` 以类型层级提升的构造器作为输入。因此，底层使用 `reifyInfoPtr#` 和 `compactAddHollow#` 的 `fill`函数，也应使用类型层级提升的构造器来选择要添加到带孔结构中的空心构造器。

然而，`fill` 依赖 `GHC.Generics` 来获取所选构造器的各种信息（见第 3.5.4 节）；而在类型的 `Generic` 表示中，构造器通过其名称（类型级字符串）来标识。这些名称是与我们在 `reifyInfoPtr#` 中使用的类型层级提升构造器不同的类型级实体。

因此，我们需要一种方法，将类型层级提升的构造器转换为表示该构造器名称的类型级字符串[29]。这正是我们也添加到 GHC 中的类型族 `LCtorToSymbol` 的作用。它只需检查表示构造器的（类型级）参数，获取其关联的数据构造器 结构，并返回一个携带构造器名称的类型级字符串（种类为 符号），如代码清单 3.10 所示。

整合所有内容及进一步演进 所有这些元素结合起来，为实现我们的 `填充` 函数提供了所需的底层骨干。它们总结于图 3.11 中。上述实现旨在尽量减少对 GHC 本身的改动，其依据是改动越少，

[29]我们沿此方向进行转换，是因为单独的类型级字符串无法唯一标识一个构造器：代码库中可能存在名称相同但形状不同的多个构造器。相反，通过使用提升到类型层的构造器，我们可以利用 GHC 的导入和名称解析逻辑来避免任何歧义。

```
1 matchFamLCtorToSymbol:: [Type] → Maybe (CoAxiomRule, [Type], Type) 2
matchFamLCtorToSymbol [kind, ty]3       | 类型构造器应用 tyCon _ ← ty, Just
dataCon ← isPromotedDataCon_maybe tyCon  =4 let symbolLit  = (mkStrLitTy .
occNameFS . occName . getName $ dataCon) 5 in Just (axLCtorToSymbolDef, [kind,
ty],symbolLit) 6 matchFamLCtorToSymbol tys  = Nothing 7 8 axLCtorToSymbolDef =9
mkBinAxiom "LCtorToSymbolDef" typeLCtorToSymbolTyCon Just 10 (\case {
TyConApp tyCon _  → isPromotedDataCon_maybe tyCon ; _ → Nothing }) 11 (\_
dataCon → Just (mkStrLitTy . occNameFS . occName . getName $ dataCon))
```

代码清单 3.10：LCtorToSymbol 的实现位于 compiler/GHC/Builtin/Types/Literal.hs

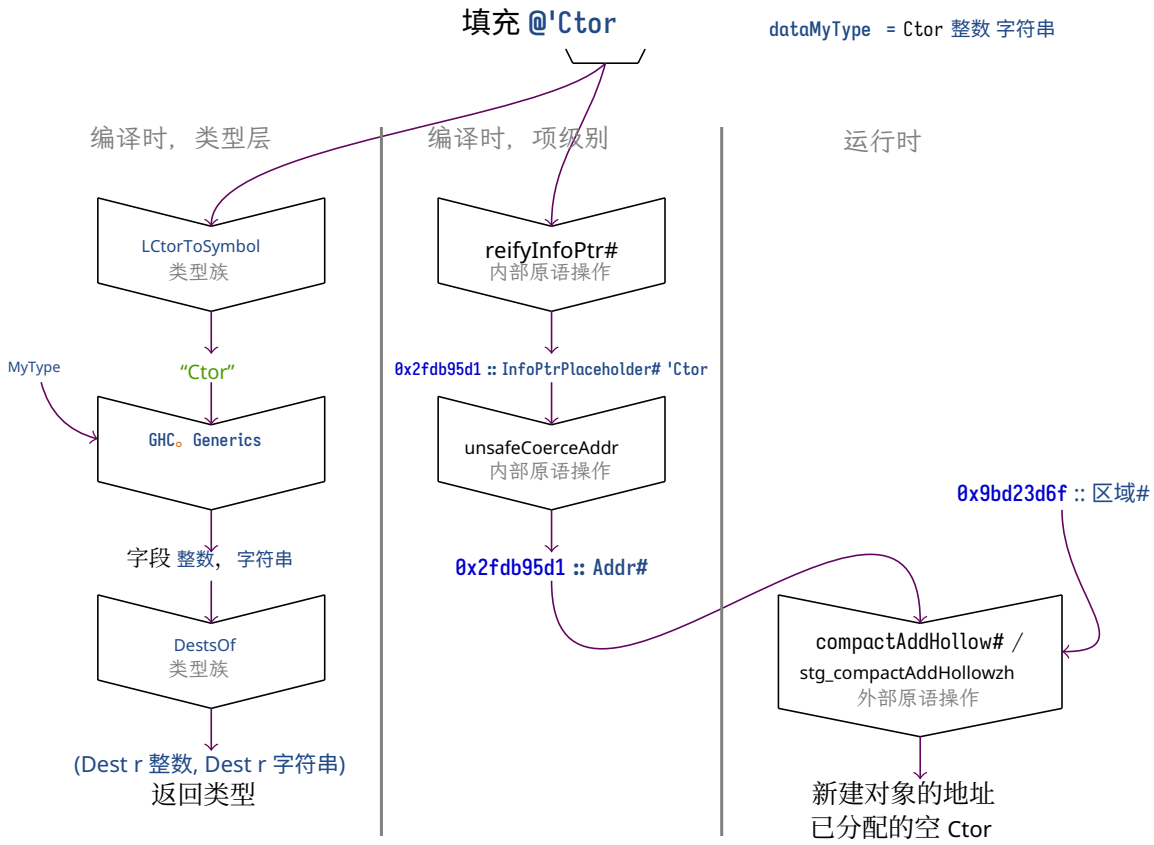


图 3.11：用于填充的实现架构示意图 填充

所需的修改越少，这项功能就越有可能顺利运行并被纳入 GHC 主版本中，尤其是考虑到目标传递对于这款工业级编译器而言仍是一个相对小众的关注点。

不过，自这次初步试验以来 [Bagrel, 2023a]，有人已提交拉取请求 [Bagrel, 2024b]，以征求社区对此议题的意见；如今，我们似乎正朝着比先前提到的三个基本操作稍完整一些的方向发展。具体而言，GHC 本身很可能会提供对 `填充` 的原语支持（而不是像目前这样仅由库代码提供）。截至撰写本文时，接口设计仍在讨论中，许多实现工作仍有待完成（或根据早期试验进行调整）。

3.6 评估 DPSHaskell 的性能

现在该看看，我们关于性能的主张是否能通过目标传递实现原型得到验证了 [Bagrel, 2023b]。

基准测试方法 我们将比较采用常规函数式风格和 DPS 风格的程序，并观察它们在速度和内存效率方面是否存在差异。不过，首先必须谨慎衡量时间和空间用量，尤其是因为 Haskell 默认采用惰性求值策略。具体来说，我们需要一种方法，强制 Haskell 对基准测试中的实现结果进行完全求值。

在所有 DPS 实现中，程序输出都存储在紧凑区域中（这是要求）。如第 3.5.1 节所述，紧凑区域会强制严格性，因此其结构会自动完成完全求值。虽然这在某些实际用例中可能并非最优，但至少简化了对这些实现进行基准测试的过程。

对于朴素实现，我们必须选择一种强制完全求值的方法。一种选项类型是将结果保留在垃圾回收堆中，并使用 `控制。DeepSeq。强制求值` 递归地对每个惰性求值项进行求值。或者，也可以使用 `Data。Compact。compact`将结果复制到紧凑区域中，从而强制严格性（通过运行时系统定义的块强制求值过程实现，这与 `控制。DeepSeq。强制求值`不同）。

只有少数程序真正适合复制到紧凑区域中：这些程序的结果会在内存中保留很长时间，并且会被完整使用（因为若结果位于紧凑区域中，未使用的部分无法被回收）。不过，`compact` 有时比`force`更快。因此，我们主要使用`force`对朴素实现进行基准测试，但在`compact` 看起来有利时（以`.copyCR` 后缀标记），也会使用它——有时是因为将结果存储在紧凑区域中更合适（例如解析器示例），有时则是因为这样能显著提升性能。

所有图表的 Y 轴（时间和内存）均采用相对刻度。基线是引用方法所耗费的时间和使用的峰值内存，并以海军蓝绘制。因此，每种输入大小对应的绝对刻度各不相同，其中 1.0 表示该输入大小下引用方法的值。对于内存图表，所有系列（`峰值`、`已分配`和`已复制`）均以引用方法的峰值内存值为基线，因此在给定大小下，各系列的值可以直接比较。

关于内存报告 在 Haskell 中准确报告内存颇具挑战。通过 `Tasty.Bench`运行基准测试时，每项测试都会重复执行，直到时间结果的偏差落在目标范围内。`Tasty.Bench`还会报告三项内存指标：已分配内存、垃圾回收期间复制的数据量，以及峰值内存。遗憾的是，已分配内存不包括紧凑区域中的分配，因此不适合作为基线。于是我们采用峰值内存作为基线，以此近似衡量某项实现的随机存取存储器占用量。

然而，如果我们使用 Tasty.Bench运行整套基准测试，那么整个运行过程中报告的峰值内存值只会不断增加，而且在对套件的另一项实现进行基准测试时不会重置。我们尝试通过分别运行每项实现的基准测试+size（而不是启动整套测试）来解决这个问题，这样就能为每项实现正确报告峰值内存。但即便如此，Tasty.Bench 报告的峰值内存使用量仍至少为 69 MB，即使给定的分情况实际使用量远低于此值（从相关的已分配值和已复制值即可推断）。

因此，除了 Tasty 基准测试之外，我们还对每种实现+的每个规模分别运行了 10 次，并使用 RTS选项 +T +s，以获得更准确的峰值内存使用量读数。在这种基准测试模式下，我们观察到的峰值内存使用量最低为 6 MB——其中包括 Haskell 运行时和其他样板开销——这比 Tasty 报告的 69 MB Bench 要好得多。

不过，许多程序报告的峰值内存使用量恰好为 6 MB，而实际分配量在 0 到 3 MB 之间（平均为 0.5 MB）。这表明，报告的峰值内存并不能准确反映实现本身实际使用的内存——它始终会高估 3 到 6 MB，通常为 5.5 MB。基于此，我们决定从所有大于或等于 10.5 MB 的峰值内存读数中减去 5.5 MB，以获得调整后的峰值内存值。对于峰值读数低于 10.5 MB 的数据点，我们不予展示，因为误差范围最高可达 2.5 MB，可能超过调整后数值的一半。

当我们没有可信的数据点来确定引用方法的峰值内存值时，就为该输入大小采用任意原点（该原点的计算旨在保持已复制和已分配曲线的形状）。这样，我们仍然可以轻松地相互比较其他值。

执行环境 我们针对给定大小，单独运行每种实现的基准测试，可以手动运行，也可以通过 Tasty.Bench 运行，并使用 RTS 选项 +T +s。我们不设置任何其他运行时选项；特别是，我们不 使用多线程，也不配置任何特定的垃圾回收选项。通过 Tasty.Bench 运行时，测试会重复进行，直到时间结果的偏差达到 5% 的目标范围内。手动运行时，每项测试重复 10 次，并取结果的平均值。

所有程序均使用优化选项 -O2 编译，所用的 GHC 9.12 版本经过修改，纳入了 ⟦Bagrel, 2023a⟧ 中最初提出的更改（这些更改最初是为 GHC 9.8 设计的）。

基准测试在一台运行 Kubuntu 22.04 的笔记本电脑上进行，处理器为 Intel i7-1165G7 @ 2.80 GHz，配备 32 GB 随机存取存储器，并连接电源。

3.6.1 串接列表和差分列表

我们比较了三种列表拼接实现。

第一种实现 concatListRight。force 使用常规链表，并且对 (++) 的调用向右嵌套，为列表拼接提供了最优的上下文：其运行时间应为 $\mathcal{O}(n)$。我们仅将其作为参考，因为这并不是实践中最常见的串接用例。在实际程序中，列表拼接通常向左嵌套，例如在最终将日志刷新到磁盘之前，持续向日志追加行。我们没有纳入包含向左嵌套的 (++) 调用的版本，因为其时间和内存消耗增长过快，会使图表的其余部分难以阅读。

另外两种实现中，对 concat 的调用向左嵌套。实现 concatDListFunLeft。force 使用函数支持的差异列表，而 concatDListDpsLeft 使用目标支持的差异列表（来自第 3.3.1 节）。不过，得益于差分列表的魔力，它们的运行时间仍应为 $\mathcal{O}(n)$。

我们从图 3.12 中可以看到，与另外两种实现相比，目标位置支持的差异列表仅在处理非常大的数据集时，内存消耗略低，运行时间也更短（约 10%）；但在处理所有较小的数据集时，它的速度要慢得多，内存效率也更低。

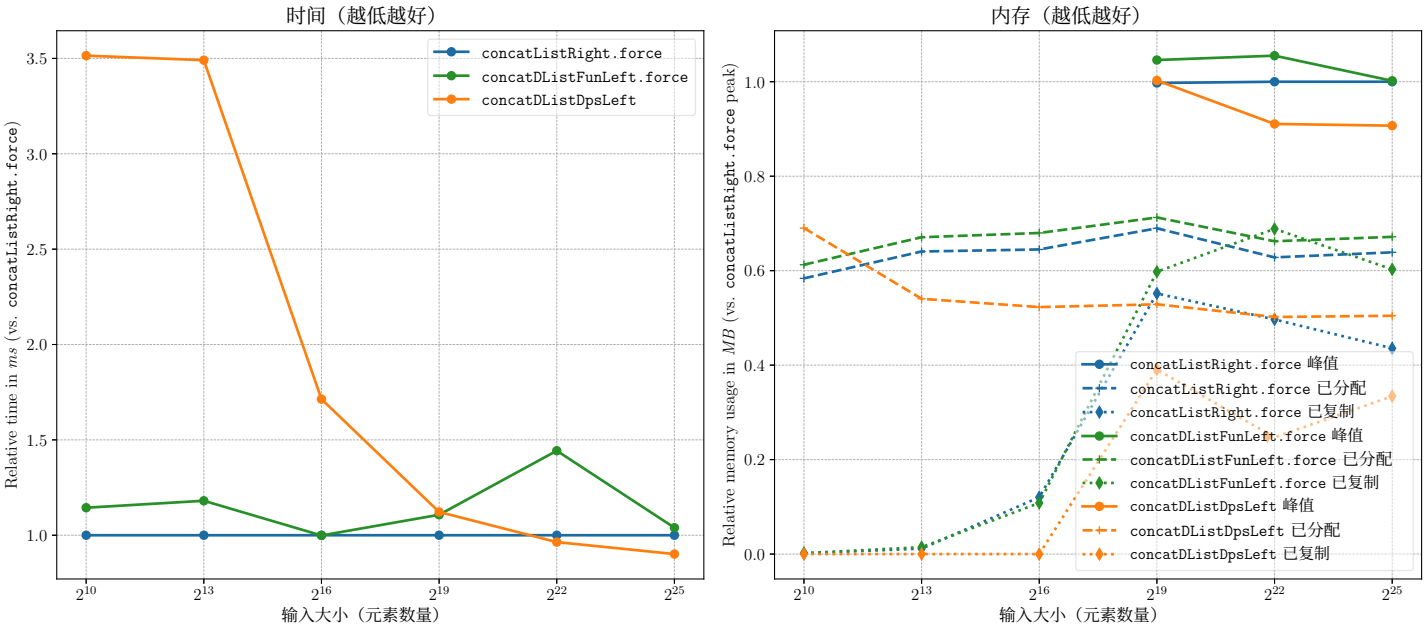


图 3.12：列表和差分列表上的迭代连接基准测试

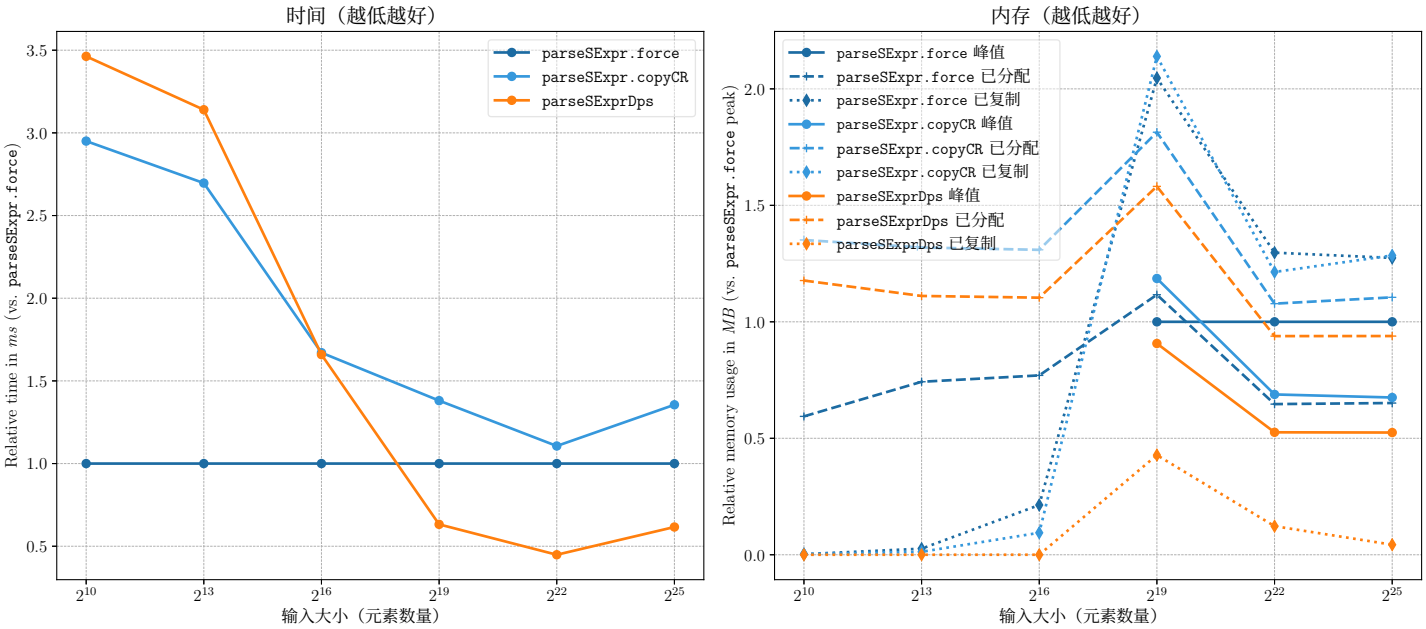


图 3.13：S 表达式解析器基准测试

这是我们在其余基准测试中会看到的趋势：对于非常大的数据集，垃圾回收的代价会变得非常高；目标传递实现的优势并不在于创建或处理数据更快，而在于避免对仍处于存活状态的大型数据树进行垃圾回收。回收垃圾（例如垃圾回收堆中的`Ampar`或`Dest`包装器）要比保留非垃圾便宜得多。

与最初呈现的基准测试相比 [Bagrel, 2024a], ，基于目标传递的差分列表的性能似乎有所提升，这是因为编译器对线性程序的处理更加高效；不过，与朴素实现相比，其差距仍远未达到理论预期，尤其是还存在其他更简单的调整方式（特别是在程序启动时设置自定义垃圾回收参数）。

3.6.2 广度优先树重标记

对于广度优先树遍历，我们将第3.3.2节的目标传递实现与一种基于*Phases*应用函子的实现进行基准测试。该实现由 [Gibbons 等人 2023]提出，是一种同样简洁的函数式环境下广度优先树遍历方法：

```
1 导入 Control.Monad.State.Lazy（runState、state） 2 3 数据类型 Phases m a，其中 4
Pure :: a → Phases m a 5 Link :: (a → b → c) → m a → Phases m b → Phases m c 6 7 8 实
例 Functor (Phases m)，其中 ... 9 实例 (Applicative m) ⇒ Applicative (Phases m) ... 10 11 now ::
(Applicative m) ⇒ m a → Phases m a 12 now xs = Link (库里 fst) xs (Pure ()) 13 14 later ::
(Applicative m) ⇒ Phases m a → Phases m a 15 later xs =Link (库里 snd) (pure ()) xs 16 17
runPhases :: (Applicative m) ⇒ Phases m a → m a 18 runPhases (Pure x) = pure x 19
runPhases (Link f xs ys) = pure f <*> xs <*> runPhases ys 20 21 traverseBFS :: (Applicative
m) ⇒ (a → m b) → 二叉树 a → Phases m (二叉树 b) 22 traverseBFS _ Nil = pure Nil 23
traverseBFS f (Node xtl tr) =24 pure Node <*> now (f x) <*> later ((traverseBFS f) tl) <*> later
((traverseBFS f) tr) 25 26 relabelPh :: 二叉树 () → (二叉树 整数, 整数) 27 relabelPh tree =
runState (runPhases $ traverseBFS (\_ → state (\s → (s, s + 1))) tree)
                                                                                0
```

从图 3.14 的部分结果中可以看出，与 Gibbons 等人 [2023] 的实现相比，基于目标的树遍历在时间和内存方面的效率都高出约五倍。不过我们必须承认，我们并没有像优化整体 DPS 框架那样，特别针对 *Phases* 实现的性能进行优化。

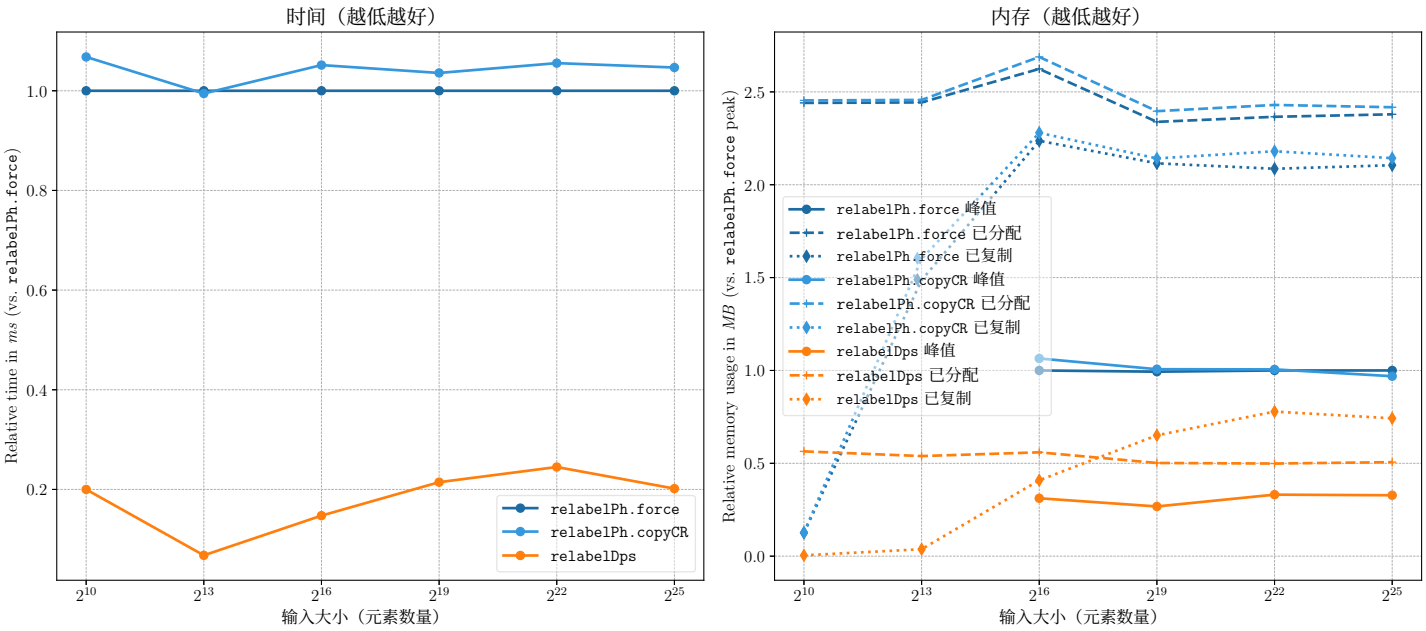


图3.14：广度优先树重标记的基准测试

这是一个很好的例子：对于这类程序，目标传递版本写起来几乎和常见的函数式风格版本一样自然，但即使在较小规模下，性能也显著更好。

3.6.3 解析 S 表达式

在图 3.13 中，我们比较了第 3.5.2 节所介绍的 S 表达式解析器的朴素实现和 DPS 实现。对于这个特定程序，即使不考虑目标传递，使用紧凑区域也很有意义，因为只要数据保存在内存中（已解析文档通常如此），就能降低应用的垃圾回收负载。因此，我们对朴素版本进行了两次基准测试：一次将结果保留在垃圾回收堆中，并使用 `强制求值` （其名称以星号为后缀）进行求值；另一次则生成结果，并将其复制到紧凑区域中（`copyCR`）。

对于较小的输入，DPS 实现起初不如朴素版本高效；但垃圾回收开始后，它很快就显著优于另外两种受测实现（参见约在 2^{16} 处，已复制内存值相较于已分配内存值和峰值内存值迅速上升的情况）。

对于大小为 2^{19} 到 2^{25}, 的数据集，尽管 DPS 实现的内存分配量高于`parseSExpr。force`，但其耗时减少了 35-55%，峰值内存占用减少了 10-45%。这表明，DPS 版本在垃圾回收堆中分配的大多数数据都是`Ampar`和 `Dest`等包装器，它们只存活一代，因此垃圾回收器可以很早就将其丢弃（正如预期），而无需将其复制到下一代；朴素版本分配的数据节点则不同，垃圾回收器必须通过复制来保留它们。

有些出乎意料的是，对于这个特定示例，即使面对最大的数据集，`parseSExpr.copyCR` 版本的表现也不如`parseSExpr.force` 版本。不过，在现实场景中，数据解析完成后通常还会继续处理一段时间，而不是立即丢弃，因此我们预计`copyCR` 实现的垃圾回收时间（进而是运行时间）会优于`force`版本，但它可能仍不及 DPS 实现，因为后者省去了将结果从垃圾回收堆复制到紧凑区域的步骤。

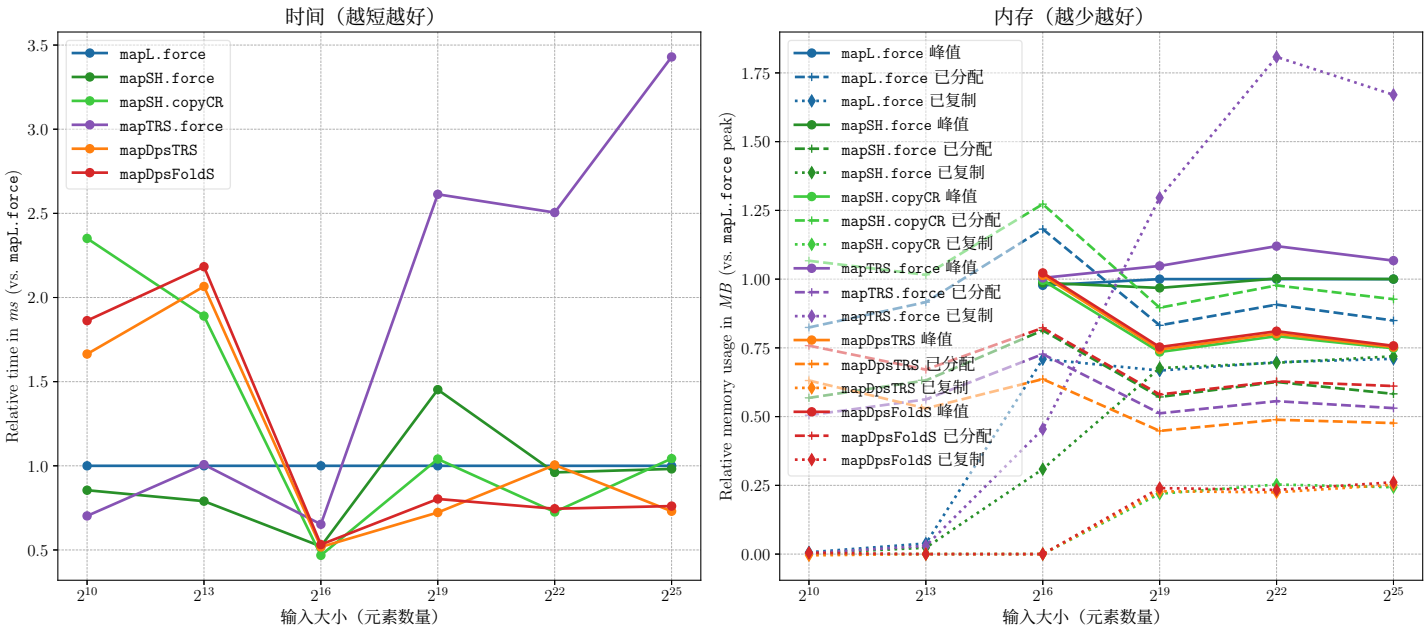


图 3.15：列表映射函数的基准测试

3.6.4 将函数映射到列表上

在 OCaml 这类严格求值的函数式语言中，列表*map* 函数的实现方式至关重要，因为朴素的非尾递归实现会使栈空间随着已处理列表的长度线性增长。为缓解这一问题，可以采用严格的尾递归实现（`mapTRS`），但在处理结束时需要额外执行一次 $\mathcal{O}(n)$ 操作来反转累加器。在 Haskell 中，这个问题或许没那么重要，因为默认惰性求值策略能让朴素的*map*（完全惰性时使用`mapL`，或严格求值头部、惰性求值尾部时使用`mapSH`）以恒定栈空间高效运行。尽管如此，还是来看看我们的 DPS 版本表现如何。

借助目标，*map* 也可以采用尾递归方式实现（`mapDpsTRS`），如第 2.2.2 节所述，无需像`mapTRS`那样执行反转操作，因为列表是通过自顶向下方法直接构建的。有趣的是，借助目标，*map* 还可以实现为左折叠（`mapDpsFoldS`），下文也将对此进行展示！在下面的代码片段中，变量名前的`!`表示严格绑定，因此，赋给该变量的表达式会在`let`表达式的主体求值之前，先求值到弱头范式。

```
1 追加:: UDest [t] ⊸ t → UDest [t]2 d `追加` x = 分情况 d &填充 @'(:) of (dh,
dt) → dh &fillLeaf x ⨟ dt 3 4 mapDpsTRS :: (a → b) → [a] → UDest [b] ⊸ () 5
mapDpsTRS _ [] d = d &fill @'[]6 mapDpsTRS f (x : xs) d = let !y =f x ; !d' = d `追
加` y in mapDpsTRS f xs d' 7 8 mapDpsFoldS :: (a → b) → [a] → UDest [b] ⊸
() 9 mapDpsFoldS f l dl = 填充 @'[] (foldl' (\d x → let !y = f x in d `追加` y)
```

图 3.15 展示了不同实现的结果。首先，我们确认最简单直观的 `mapL.强制求值` 表现非常出色，这得益于它的简洁性。我们还看到，对于像 Haskell 这样的语言，严格的函数式尾递归实现 `mapTRS` 并不特别有趣（即使是图表中未展示、但结果相近的严格头部、惰性尾部变体），因为无论在时间还是内存方面，它的表现都逊于其他所有实现。

对于 `mapSH`，情况要复杂一些。采用 `强制求值` 求值方法时，其表现随输入大小而异：有些方面优于参考实现 `mapL,强制求值`，有些方面则较差；但采用复制到紧凑区域的方式来实现深度求值时，对于 2^{16} 和 2^{22} 大小的数据，它的运行时间达到可能的最短，领先基于目的地的实现几个百分点，峰值内存使用量也同样出色。不过，我们确实无法解释为什么这一特定实现采用复制到紧凑区域的方式会得到如此好的结果；但它在若干种数据规模下表现最佳，因此值得纳入比较。

最后，基于目的地的实现 `mapDpsTRS` 和 `mapDpsFoldS` 都表现出色：对于大多数从 2^{16} 大小开始的数据集，与参考实现相比，它们的运行时间减少了 20–45%，峰值内存最多节省 25%。然而，正如我们之前所说，DPS Haskell 在处理较小列表时的性能仍相当令人失望。降低目标传递带来的开销，或许应当成为未来改进 DPS Haskell 实现的重点。

顺带一提，Bour、Clément 和 Scherer ⟦2021⟧ 指出，在 OCaml 中，等效的 `mapDpsTRS` 在时间性能上始终优于等效的 `mapTRS`（耗时减少 15%–50%）。不过，他们的目的地传递系统只用于编译器中间语言级别，这或许能进行比我们在源代码级别所能实现的更激进的优化。

3.7 结论

实践证明，使用目的地进行编程在实际的函数式编程中确实有一席之地，OCaml 编译器近期采用了 *尾递归模 Cons* ⟦Bour、Clément 和 Scherer， 2021⟧ 便是明证。在本章中，我们展示了如何借鉴 λ_d（第2章）中的大部分思想并加以调整，从而在不改变 Haskell 核心原则的前提下，实现安全的目的地传递。我们还展示了第2章中的大多数示例几乎无需调整，便可在 DPS Haskell 中实现；同时强调，我们针对 Haskell 实现的目的地传递，如何帮助提升垃圾回收代价高昂的程序的性能。为此，我们采用了 Yang 等人提出的 紧凑区域 ⟦Haskell 内存管理方案；此外，我们的工作还扩展了紧凑区域的高效使用方式。

最后，我们还展示了本文中介绍的一些算法如何通过目的地传递获得实际收益。不过，我们的 DPS Haskell 实现仍有一些局限，导致基于目的地的算法并不总能达到其理论性能。例如，紧凑区域与垃圾回收堆完全隔离，因此在填充基于目的地的数据结构时，必须将数据从后者复制到前者，从而产生开销。此外，创建区域、创建 ampar，以及在过程中创建或消除包装器所需的样板代码开销相对较高；尤其是在处理小型示例时，这无法与 GHC 运行时系统为惰性数据结构提供的高度优化机制相匹敌。

在本章中，我们只能将 DPS Haskell 用于承载不受限数据的结构。这样选择是为了保持 API 相对简单，同时避免作用域逃逸问题（第 2.3 节）。在下一章中，我们将重新审视这一限制，并探讨如何修改 DPS Haskell，以便安全地在带孔结构中存储线性资源。

第4章

扩展 DPS Haskell 以支持线性数据结构

第3章中开发的 DPS Haskell API 方法存在的主要问题是，通过目标填充构建的数据结构无法存储线性数据。因此，任何使用目标构建的数据结构（例如，为了提升速度或改善操作的复杂度），其通用性都不如通常的函数式数据结构；后者既能存储线性数据，也能存储不受限数据。

我们回顾一下，这一限制源于目标的使用方式十分灵活，却没有年龄系统以细粒度地防止作用域逃逸。因此，我们不得不将 不受限性 用作屏障，确保目标不会逃逸出其作用域。

然而，先前的研究（如 ⟦Minamide、 1998⟧, 或 ⟦Lorenzen、Leijen、Swierstra 和 Lindley 的工作）允许高效队列（见第 2.2.3 节）或任何带孔结构存储线性数据，而且不会出现作用域逃逸问题。这主要是因为这些研究没有目标这一概念；在这些工作中，我们只能通过与不完整结构本身（即 ampar）交互，来填充结构中的空洞。因此，我们应当设法让 Haskell 实现至少具备与这些研究同等的表达能力。

本章首先将界定线性数据的目的地所带来的挑战与风险，随后逐步探索各种设计思路，在确保安全性的同时，扩展我们的 DPS Haskell API 以支持这些目的地。首先，我们将考察一种受限模型：线性数据的 ampar 只能有一个空洞（即右侧只能有一个目的地），仿照 Minamide 的做法。随后，我们将借助线性透镜，考察一种扩展，使线性数据的任意 ampar 都不受限制（尤其是允许有多个空洞）。

4.1 线性数据的目的地面临的挑战：作用域逃逸、更新 d

从这里开始，我们将用线性目标指类型为 $\lfloor_1 T\rfloor$ 或对应的 `Dest 1 t` 的对象，也就是说，这种目标能够容纳线性数据。同样，非线性目标指类型为 $\lfloor_\omega T\rfloor$ 或 `Dest ω t` 的对象。无论哪种情况，目标本身始终是一种必须以线性方式管理的资源。最后，为了让阐述更清晰，本章大部分内容都省略紧凑区域的细节。

如果我们希望允许用线性数据填充目标，就必须极其谨慎地处理潜在的作用域逃逸（参见第 2.3 节）。只有在较旧作用域中的<sty le id='1'>线性 目标在较新的作用域中被填充时，才会发生作用域逃逸。这是因为只有线性目标本身才能被填充

为其他目标。正如第 2.3 节所述，用严格较旧的目标填充较新的目标是安全的；反过来，用较新的目标（或年龄相同的目标）填充较旧的目标才会引发问题。

一种可能的补救办法是，禁止使用 `fillLeaf` 或 `fillComp` 函数填充线性目标（将这些函数的类型签名改为仅适用于非线性目标）。这样，线性目标就只能通过对其来源的 ampar 进行操作来填充，正如 [Lorenzen、Leijen、Swierstra 和 Lindley，2024；Minamide，1998] 所述。这样，我们就能确保只能填充最新鲜的线性目标（目标在通过 `updWith`于新作用域中变得可访问时便开始变旧；因此，位于已闭合 ampar 右侧的目标比上下文中的任何其他目标都更新鲜）。我们将用基名 `extend` 指代这组直接对 ampar 进行操作、以填充其中目标的运算符。另一方面，非线性目标仍可使用各种 `fill`函数。需要注意的是，使用空心构造器填充目标不会导致任何作用域逃逸问题，因此在填充线性和非线性目标时，使用 `fill @'Ctor` 都是安全的。

来看看更新后的 API 会是什么样子：

```
1 数据 Ampar s t2 数据 Dest m t -- m 是重数，可以是
1 或 ω3 newAmpar:: 令牌 ⊸ Ampar s (Dest 1 s)4 updWith::
Ampar s t ⊸ (t ⊸ u) ⊸ Ampar s u56    -- 用于非线性目
标7 fillLeaf ::t → Dest ω t → ()89 -- 用于线性目标10
extendLeaf ::t ⊸ Ampar s (Dest 1 t) ⊸ Ampar s ()
```

其思路是，`extendLeaf` 可以用来开启一个新作用域，填充属于这个新作用域的线性目标，然后立即关闭该作用域，而不让用户介入。因此，我们可以认为，在`extendLeafval (Ampar struct d)` 中，被`extendLeaf`填充的线性目标`d` 总是比存入其中的`val` 严格更新，从而避免作用域逃逸（回想第 2.3 节中合理的示例：将较旧的目的地存入较新的目的地，不会有任何问题）。

不过……这行不通：

```
1 newAmpar tok0 `updWith` \(d0 ::Dest 1 (Dest 1 t)) →2 newAmpar tok1 `updWith` \(d1:: Dest
1 t) →3 (newAmpar tok2 `updWith` \(d2:: Dest ω ()) → d2 &填充Leaf () ⨟ d0) &扩展Leaf
d1
```

在这个格外复杂的例子中——尽管根据上文所述的 API，它仍然可以通过类型检查[30]——一个旧的目标 `d0` 被隐藏在一个较新的目标 `d2` 之中，而我们刚刚已经消耗了后者。因此，我们有一个 ampar `(newAmpar tok2 `updWith` \d2 → d2 &fillLeaf () ⨟ d0)`，它的右侧包含一个目标 `d0`，这个目标并不属于它自身，而是来自先前的作用域。然而，只看类型的话，它与右侧仅包含自身目标的 ampar 无法区分。因此，我们可以对这个被诅咒的 ampar 使用 `extendLeaf`，从而导致作用域逃逸（因为 `d1` 被存储在最外层的 ampar 中）！

如果没有恰当的年龄系统来确保作用域中不存在作弊行为，那么捕获式`updWith` 函数与安全的线性目标之间似乎不可避免地存在冲突。

[30]然而，它会被 λ_d 拒绝，因为当 `d0` 在最内层作用域中用作变量时，其年龄会是 $\uparrow^2$，而规则 λ_d–ty/id 不允许这样做。

非捕获式updWith函数如果我们想防止较旧的目标藏在较新的 ampar 中，就可以要求updWith 不捕获任何线性资源。这样一来，在updWith 的作用域中，唯一可访问和可使用的目标，就是直接来自当前打开的 ampar 的目标。

这可以通过将类型签名从 `updWith :: Ampar s t ⊸ (t ⊸ u) ⊸ Ampar s u` 改为 `updWith :: Ampar s t ⊸ (t ⊸ u) → Ampar s u`来实现。注意这个无限制的函数箭头 →：它表示用户提供的线性函数不能捕获任何线性资源（但仍必须线性地使用其参数）。施加这一严格限制会带来许多后果。

首先，显然，我们不能在updWith的作用域内使用来自外层作用域的任何线性资源。尤其是，我们不能使用来自顶层线性作用域的令牌（我们在第 1.8 节中说明了只有一个顶层线性作用域有多方便）。相反，如果我们想在updWith的主体中创建新的 ampar，就必须通过调用withToken来新建一个嵌套的线性作用域。或者，我们也可以调整updWith ，让它向内层作用域提供一个线性令牌，这样便得到updWithToken `::Ampar s t ⊸ ((t, Token) ⊸ u) → Ampar s u`（updWithToken 操作的 ampar 是一种线性资源，我们可以据此建立线性链，并通过新令牌将其传递下去）。

顺带一提，即使对 updWith 加以限制，我们仍然无法使用 fillLeaf 或 fillComp 函数来填充线性目标，主要有两个原因。首先，我们根本无法将线性资源引入作用域，因此很少有可用于填充线性目标的有用内容。其次，允许 fillLeaf作用于线性目标，可能会使来自同一作用域的兄弟目标之间发生交互，如第 2.3 节所述。

在前面的章节中，为防止真正的作用域逃逸而施加的限制，也恰好阻止了兄弟目标之间发生此类交互；但这里并非如此。我们来看一个例子：

```
fromAmpar'(newAmpar @(t, Dest 1 t) tok `updWith` \(d :: Dest 1 (t,
Dest 1 t)) →分情况 (d &填充 @'(,)) of (dt :: Dest 1 t, ddt :: Dest 1
(Dest 1 t)) →ddt&fillLeaf dt    )
```

在这里，我们为一个 t 和一个 Dest 1 t构成的对新建一个 ampar。用空心对构造器填充根节点目标 d 后，我们得到两个目的地：dt 和 ddt。如果我们可以使用一个左侧接受线性目标的 fillLeaf 函数，实际上就能把 dt 放进 ddt。由于我们已经消耗了这两个目的地，并且返回了单位，看起来这个 ampar 已经完成了；但实际上，我们从未为这个对的第一个字段提供过类型为 t 的值！因此，尽管结构仍不完整，我们却可以对它调用 fromAmpar' 并读取它！

这表明，即使对 updWith 施加了限制（以防止捕获线性资源），允许 fillLeaf 或 fillComp 作用于线性目标仍然是不安全的。

4.2 补全适用于单孔 ampar 的运算符族

我们刚刚看到，要安全地使用线性目标，就需要直接作用于封闭的 ampar、而非目标的运算符。事实上，[Minamide、1998], 或 [Lorenzen、Leijen、Swierstra 和 Lindley 2024] 正是如此：它们只允许含有一个空洞的不完整结构，并且通过操作整个不完整结构本身来扩展结构，而不是操作目标（因为这些工作中并不存在目标）。

因此，就像我们刚才在上面使用 `extendLeaf` 所做的那样，我们将为仅含一个空洞的 ampar 设计新运算符，也就是说，它们的右侧恰好有一个目标。

通用的 扩展 函数 与**填充** `@'Ctor` 函数（见第3.4.2节和第3.5.4节）类似，我们希望有一个**扩展** `@'Ctor` 函数，使我们能够用一个任意的新空心构造器来扩展只有一个空洞的 ampar。不过，与**填充** `@'Ctor` 不同，后者可以返回数量不定的目的地（例如，**填充** `@'(:)` 会返回两个新目的地），我们必须设计一个**扩展** `@'Ctor` 函数，将多个值作为输入，直接填入空心构造器除一个字段外的所有字段，从而只留下一个新空洞，并且只创建一个新目的地[31]。

为了体现其通用性，这类函数的类型签名会变得有些繁琐，如下所示：

```
扩展 :: ∀ lCtor k t m s fiTys kthFiTy othFiTys.
                (OnlyLinear lCtor t,
 FieldTypes lCtor t ~ fiTys,
 At k fiTys ~ kthFiTy, Remove k fiTys ~ othFiTys) ⇒
 othFiys %m→ Ampar s (Dest m t) ⊸ Ampar s (Dest m kthFiTy)
```

这里有很多内容需要弄清楚。`OnlyLinear lCtor t` 是一个约束，要求所选的数据构造器只能具有线性字段（稍后会详细介绍）。`FieldsTypes` 是一个类型族，它会返回给定构造器的字段类型（由`lCtor` 以及该构造器所属的类型`t` 指定）。例如，`FieldsTypes '(:) [Int]` 会解析为`(Int, [Int])`。`At k` 是一个类型族，它会返回元组类型的第`k`个元素（即类型）；类似地，`Remove k` 会返回给定的同一元组类型，但移除了第`k`个元素。

符号 `~` 表示类型相等约束，可像上文那样用于函数或类型类的类型签名中，将类型变量绑定到复杂的类型级表达式。上文中，我们说`fiTys` 表示`lCtor`的所有字段类型，`kthFiTy` 是其中第`k`个，而 `othFiTys` 则表示除第`k`个以外的所有其余字段类型；如果剩余类型不止一个，就将它们组成元组。

例如，在 `extend @'(:) @0 @[整数]`, `kthFiTy` 是 `整数`，而 `othFiTys` 是 `[整数]`（因为只剩下一种类型，所以不是元组形式）。相反，在 `extend @'(:) @1 @[整数]`, `kthFiTy` 是 `[整数]`, ，而 `othFiTys` 只是 `整数`。

实践中，这样的类型签名意味着 `extend` 接收一个 ampar，其右侧是类型为 `t`、重数为 `m`[32] 的目标。随后，它会分配一个新的空心构造器（由 `lCtor` 指定），消耗`othFiTys`，重数为 `m`，以填充新空心构造器的所有字段，除了第 `k` 个字段；然后返回一个重数相同（即 `m`）的目标，该目标对应于类型为 `kthFiTy` 的剩余字段。

例如，差分列表上的 `追加 :: DList t ⊸ t ⊸ DList t` 函数可以直接用 **扩展** 表示（参见下面的代码摘录）。到目前为止，`DList` 只能存储非受限元素，但现在，我们可以使用差分列表来承载线性元素！这样一来，`DList` （使用目标）就与常规列表或差分列表的常规纯函数式编码一样通用。

[31]或者，也可以使用 `updWith` 访问一个线性目标，然后对其使用 `fill @'(:)`，从而创建两个新的线性目标。但这样一来，我们就无法使用上文提到的 `extendLeaf` 函数来用完整值填充这些目标，因为 `extendLeaf` 仅适用于右侧只有单个目标的 ampar，而 `fillLeaf` 仅适用于非线性目标。[32]这样，我们就可以对右侧目标为线性或非线性的 ampar 使用 `extend`，尽管通过使用`fill` 系列函数，可以更灵活地直接操作非线性目标。

```
类型 DList t = Ampar [t] (Dest 1 [t])追加
::DList t ⊸ t ⊸ DList t dlist `追加` x =
dlist&扩展 @'(:) @1 x
```

我们还可以使用 扩展 来定义和使用 拉链树（参照 [Huet, 1997；Lorenzen、Leijen、Swierstra 和 Lindley， 2024]），以容纳线性数据：

```
data 树 t = Nil | Node (树 t) t (树 t) type 拉链树 t =Ampar (
树 t) (Dest 1 (树 t)) 左节点 ::拉链树 t ⊸ t ⊸ 树 t ⊸ 拉链树
t 左节点 parent x rtree = parent &扩展 @'Node @0 (x, rtree)
右节点 ::拉链树 t ⊸ 树 t ⊸ t ⊸ 拉链树 t 右节点 parent
ltree x =parent &扩展 @'Node @2 (ltree, x)
```

请注意，extend 不支持包含 0 个字段的构造器（因为extend 需要返回一个目标，而将零元构造器写入空洞时不会新建目标）。不过，零元构造器和其他完整值一样，也可以写入结构中，只需使用extendLeaf：

```
extendLeaf :: ∀ s t m. t %m→ Ampar s (Dest m t) ⊸ Ampar s ()
```

这里我们也将 extendLeaf 设为重数多态，相较于第 4.1 节中介绍的初始版本有所不同，但其行为保持不变。

事实上，在调用 extendLeaf 时向用户返回一个 Ampar s () 毫无意义；直接提取类型为 s 的已完成结构会更好：

```
fromAmparWithLeaf :: ∀ s t m. t %m→ Ampar s (Dest m t) ⊸ s
fromAmparWithLeaf = fromAmpar'. extendLeaf -- 我们用它来实现拉链
树的剩余运算符 完成 ::ZipperTree t ⊸ Tree t done parent =parent
&fromAmparWithLeaf Nil
```

构造器字段的重数以及 extendUr 在上文的 extend 定义中，我们考虑了由 lCtor 表示的构造器只有线性字段，这由约束 OnlyLinear lCtor t 指明。实际上，我们消费其他字段的值以及返回新目的地时所采用的重数m，与父目的地的重数相同。

在 λ_d 中，每个数据构造器都有线性字段，除了指数构造器 Mod_m，它有一个模式为 m 的字段。不过在 Haskell 中，构造器默认只有线性字段；但如果相应的数据类型使用 GADT 语法定义，且字段类型后跟着一个非受限箭头 →，则可以拥有非受限字段。非受限字段是一种存放不受限数据的方式，无需显式添加 Ur 层。

理想情况下，我们应进一步细化 扩展 函数，使其对每个字段的重数具有多态性。然而，迄今为止，我们无法通过 GHC.Generics获取字段重数的信息——而我们正是借助它，才能让 DPS Haskell 支持用户定义类型。因此，目前看来，要进一步修改 扩展 函数以适应字段的任意重数并不容易。

这还会带来另一个结果：我们必须为 Ur构造器添加一个专用版本的 扩展 函数：

```
extendUr :: ∀ s t m. Ampar s (Dest m (Ur t)) ⊸ Ampar s (Dest ω t)
```

鉴于 m·ω=ω 对于任意 m，extendUr 函数可以作用于具有线性或非线性目标的 ampar，但始终返回具有非线性目标的 ampar。

在这种情形下，使用 extendUr 是获得初始非线性目标 Dest ω t的主要方式，因为 newAmpar 最初只会生成右侧带有线性目标的 ampar。

使用 extendComp 组合不完整结构 在这种情形下，我们还需要最后一个函数extendComp，它将两个 ampar 接合起来：

```
extendComp :: ∀ s t u. Ampar t u ⊸ Ampar s (Dest 1 t) ⊸ Ampar s u
```

在 extendComp 中，父 ampar 中的目标必须是线性的，正如我们在第 2.5.2 节中提到的。否则，便可能进行如下漏洞利用：

```
extendCompFaulty :: ∀ s t u m. Ampar t u ⊸ Ampar s (Dest m t) ⊸ Ampar s u
parentAmpar :: Ampar (Ur t) (Dest ω t) = newAmpar tok1 &extendUr -- 解析为 Ampar (Ur h) (Dest →h)
childAmpar :: Ampar Int (Dest 1 Int) = newAmpar tok2 -- 解析为 Ampar (h') (Dest →h')
resAmpar :: Ampar (Ur t) (Dest 1 t) = parentAmpar & extendCompFaulty freshAmpar -- 解析为 Ampar (Ur h') (Dest →h')
Ur (linearRes') = resAmpar &fromAmparWithLeaf linearResourceDoNotDuplicate in
(linearRes', linearRes')
```

借助上述代码，我们创建了一个 ampar resAmpar，其中线性目标指向一个非线性空洞。这是因为，组合所产生的 ampar 会继承子 ampar 右侧的内容（此处为线性目标 Dest →h' :: Dest 1 Int）。然而，子 ampar 的左侧已被移植到父 ampar 的非线性空洞 h 中。因此，h' 会传递性地变为非线性，而其对应的目标仍是线性的。这使得线性资源可以被复制，如最后一行所示，因此我们必须加以阻止。出于同样的原因，在这种设置下，我们无法定义函数 fillComp :: Ampar t u ⊸ Dest ω t ⊸ u。[33] 在第3章中，由于所有目标在设计上都是非线性的，因此不会出现这个问题。

不过，我们可以定义 extendComp 和 fillComp 的替代版本，使其作用于非线性目标，并采用略有不同的类型签名，从而确保这次是安全的：

```
extendUComp :: ∀ s t u. Ampar (Ur t) u ⊸ Ampar s (Dest ω t) ⊸ Ampar s u
fillUComp :: ∀ t u. Ampar (Ur t) u ⊸ Dest ω t ⊸ u
```

这里，子 ampar 必须是无限制的结构（如前缀 Ur 所示）。这确保 u 中出现的任何目标都已是无限制的。因此，我们消除了线性性漏洞，并恢复了第3章中的相同组合行为。

[33]请回想一下，自本章第一节起，我们就已经排除了为线性目标定义类似填充的函数。

4.3 具有多个空洞、存储线性数据的 ampar

将 ampar 右侧的目标限制为仅一个并不理想。事实上，我们的表达能力仍与现有工作处于同一水平，同时也失去了使 λ_d 从一开始就具备创新性的部分灵活性。

然而，在 ampar 的右侧支持多个线性目标会引入相当大的复杂性。在上一节中，每个 ampar 只有一个目标；我们要求用户直接与 ampar 交互，而不是与目标本身交互，从而避免了作用域逃逸。由于不存在歧义，这种方式很直接——唯一的目标是 扩展操作唯一可能作用的对象。相比之下，如果我们现在允许右侧出现任意类型，但仍希望限制用户直接访问目的地（原因与上文相同；updWith 现在是非捕获的，并且我们希望防止作用域逃逸以及兄弟目标之间的恶意交互），就必须引入一种机制，让用户指明他们想要操作的具体目标。一种方法是要求用户提供一个线性透镜。首先，让我们定义什么是透镜。

透镜是获取器和设置器组成的对，可在函数式编程语言中方便地处理不可变数据结构。类型为Lens t1 t2 u1 u2 的透镜有两个分量（至少在概念上如此）：

- 一个获取函数，用于访问类型为u1 的值，该值位于类型为t1的结构中的某处，并由此确立透镜的焦点；
- 一个函数，通过将类型为u1 的聚焦值替换为类型为u2的值，将类型为t1 的结构转换为类型为t2 的结构。特别地，当u1=u2时，t1=t2，因此该函数充当聚焦值的设置器。

在最简单的表达形式中，非线性透镜由以下数据类型表示 ：

```
data LensNL t1 t2 u1 u2 =LensNL { 视图:: t1 → u1 -- 只从 t1 中获取 u1，丢弃其余部分 , 更新 ::t1 → u2 → t2 -- 丢弃 t1 中的 u1，并用 u2 替换它，以构造 t2      }
```

给定一个具有 3 个字段的数据结构Triple ，让我们创建一个透镜来聚焦第一个字段：

```
数据 Triple s t u  = Triple { 第一个 :: s，第二个:: t，第三个 :: u }
fstNL ::LensNL (Triple s t u) (Triple s' t u) s s' fstNL  = LensNL { 视图 = \(Triple x y z) → x , 更新  = \(Triple x y z) → x'  → Triple x' y z }
```

对于上述这类非线性透镜 LensNL ，如果我们使用 视图 组件，那么我们会消耗一个类型为 t1 的值，但除了类型为 u1的部分之外，其余大部分都会被丢弃。如果我们使用 更新 组件，情况则相反；我们会丢弃类型为 u1 的部分，但保留原始 t1的其余全部内容。

对于我们现在要介绍的线性透镜，不能单独使用视图 和更新 分量；否则，我们就可能丢弃类型为t1的原始结构中的部分内容，正如刚才所见。因此，我们将两个函数视图 和更新 合并为一个函数：

```
newtype Lens t1 t2 u1 u2 = Lens (t1 ⊸ (u1, u2 ⊸ t2))
```

借助类型为`Lens t1 t2 u1 u2`的线性透镜，我们可以将类型为`t1`的结构拆分为一对：一个类型为`u1`的值（即焦点），以及一个类型为`u2 ⊸ t2`的闭包。这个闭包携带原始`t1`结构的其余部分；只要给它一个`u2`，它就能生成类型为`t2`的结构。换句话说，第二个类型为`u2 ⊸ t2`的分量，实际上只是缺少一个`u2`便可补全的`t2`的函数式表示。若要遵循线性性，这一对中的两个分量都必须被消耗。

将我们的`Triple`示例更新为线性透镜，过程相当直接：

```
fst:: Lens (Triple s t u) (Triple s' t u) s s'
fst  = Lens (\(Triple x y z) → (x, \x' → Triple x' y z)) -- 捕获 x ↗ ↖捕获 y 和 z
```

用于选择要操作的目标的透镜 现在我们引入一组新函数，其基本名称为 `extendFocused`，它们可以使用线性透镜处理含有多个空洞（因而有多个目标）的 ampar。这些线性透镜有双重用途：它们不仅让用户能够选择要操作的目标，还为我们提供了一种方式，能够将新建的目标（来自新建空心构造器的字段）或填充操作产生的副产品，重新插入到 ampar 右侧此前被消耗的目标所在的确切位置。这得益于透镜的双向特性：它们既是获取器，也是设置器。

我们先来看通用的 `extendFocused @'Ctor`函数：

```
extendFocused :: ∀ lCtor u t1 t2 s m. (OnlyLinear lCtor u) ⇒ 透镜 t1 t2 (Dest m u) (DestsOf lCtor m u) → Ampar s t1 ⊸ Ampar s t2
extendWithNodeOnFst:: Ampar s (Triple (Dest m (树 t1)) t2 t3)
                    ⊸ Ampar s (Triple (Dest m (树 t1), Dest m t1, Dest m (树 t1)) t2 t3)
extendWithNodeOnFst ampar  = ampar& extendFocused fst @'Node
```

这里，`extendFocused` 接收一个类型为 `Lens t1 t2(Dest m u) (DestsOf lCtor m u)` 的透镜；也就是说，它聚焦于 ampar 的类型为 `t1` 的右侧中的一个 `Dest m u`，并让我们为结果 ampar 构造一个类型为 `t2` 的新右侧，其中类型为 `Dest m u` 的目标会被类型为`DestsOf lCtor m u` 的值替换（即一个包含新潜在目的地的元组）。

透镜由 `extendFocused` 作为非线性参数传入 →），因此无法捕获任何线性资源。因此，透镜被强制为纯粹的结构操作，可以根据需要在类型为 `t1` 的各种值上重复应用。

Nonet 不过，我们本可以用 updWith 和 fill 来实现 extendWithNodeOnFst `@'Node`[34]

事实上，在设计 `extendFocusedLeaf` 函数时，线性透镜的额外价值便显而易见。该函数首次（在 Haskell 实现中）使我们能够在右侧的 ampar 任意结构中存在某个线性目标时，使用一个值填充该目标：

```
extendFocusedLeaf :: ∀ u t1 t2 s m. u %m→ Lens t1 t2 (Dest m u) () → Ampar s t1 ⊸ Ampar s t2

extendWithValOnFst :: t1 ⊸ Ampar s (Triple (Dest 1 t1) t2 t3) ⊸ Ampar s (Triple () t2 t3)
extendWithValOnFst x ampar = ampar &extendFocusedLeaf fst x
```

[34]如前所述，`fill @'Ctor` 不会导致作用域逃逸，因此可安全用于线性目标和非线性目标

这里不会发生作用域逃逸。由于updWith 是非捕获的，我们可以确定，ampar 右侧的目标就是指向其左侧空洞的目标（从而消除了第 4.1 节末尾提出的问题）。因此，与extendLeaf类似，extendFocusedLeaf 函数只能用现有值（也就是较早的值）填充较新的目标。同样，我们无法让兄弟目标相互作用。这意味着，我们已经实现了本章开头提出的目标：我们可以在 DPS Haskell 中安全地构建包含多个线性空位的结构！

为完整起见，我们最后定义 extendFocusedUr、extendFocusedComp 和 extendFocusedUComp，遵循与 extendFocusedLeaf相同的模式：

```
extendFocusedUr :: ∀ u t1 t2 s m. 透镜 t1 t2 (Dest m (Ur u)) (Dest ω u) → Ampar s t1 ⊸ Ampar s t2
extendFocusedComp :: ∀ u t1 t2 s u1 u2. Ampar u1 u2 ⊸ 透镜 t1 t2 (Dest 1 u1) u2 → Ampar s t1 ⊸ Ampar s t2
extendFocusedUComp :: ∀ u t1 t2 s u1 u2. Ampar (Ur u1) u2 ⊸ 透镜 t1 t2 (Dest ω u1) u2 → Ampar s t1 ⊸ Ampar s t2
```

这两个函数都与各自的单空洞版本 extendUr 和 extendComp 非常相似。在 extendFocusedUr 中，我们消耗一个类型为 `Dest m (Ur u)` 的目标，并将其替换为一个类型为 `Dest ω u`的目标。在 extendFocusedComp 中，我们消耗一个类型为 `Dest 1 u1`的目标，并将子 ampar 左侧（类型为 `u1`）写入其中，然后用子 ampar 右侧（类型为 `u2`）的内容替换这个被消耗的目标。extend 函数与相应的 extendFocused 函数之间唯一的区别在于：前者在顶层的单个目标上执行填充操作，而后者则在 ampar 右侧的焦点位置执行该操作。

4.4 最终 API 与广度优先树遍历（更新版）

基于线性透镜的 extendFocused 函数系列，解决了如何安全地用线性数据填充目标这一核心难题。尽管由此得到的接口并非尽善尽美——尤其是在易用性方面——但考虑到必须使用现有 Haskell 类型系统这一限制，该接口在表达能力与复杂度之间仍取得了合理的平衡。

例如，DPS 系统支持目标的两层嵌套，因此我们可以构造并使用由目标组成的线性目标（类型 `Dest 1 (Dest m t)`）。事实上，这完全足以实现一些非平凡的示例，例如树的广度优先遍历，不过这次我们使用的是 Haskell 中基于目的地的高效队列！让我们最后再回顾一下这些示例。本章最后一部分还会考虑紧凑区域（主要是在函数的类型签名中到处添加 `r` 和 `Region r` ）。高效队列（以及构成这些队列的线性数据所用的差分列表）的代码见清单 4.1，更新后的广度优先树遍历代码见清单 4.2。

这些实现直接源自本文档中我们围绕列表、高效队列和广度优先遍历示例所做的各种迭代与改进。

值得注意的是，在这些示例中，我们同时利用了 fill 和 extend 这两类函数。例如，我们复用了第 4.2 节中介绍的差分列表实现；对于该实现，我们不需要 updWith带来的灵活性，而是依赖 extend 函数，以便在差分列表中存储线性数据（这与 [Lorenzen、Leijen、Swierstra 和

```
类型 DList r t =ampar r [t] (Dest r 1 [t])

newDList :: 区域 r ⇒ 令牌 ⊸ DList r t
newDList = newAmpar @[t]

dListToList :: 区域 r ⇒ DList r t ⊸ [t]
dListToList dlist = fromAmpar' (dlist &extendLeaf [])

追加 :: 区域 r ⇒ DList r t ⊸ t ⊸ DList r t
追加 dlist x =dlist &扩展 @'(:) @1 x


--------------------------------------------------------------------------------

数据 EffQueue r t = EffQueue [t] (DList r t)

新建EffQueue :: 区域 r ⇒ 令牌 ⊸ EffQueue r t
新建EffQueue tok = EffQueue [] (newDList tok)

单例 :: 区域 r ⇒ 令牌 ⊸ t ⊸ EffQueue r t
单例 tok x = EffQueue [x] (newDList tok)

队列转列表 :: 区域 r ⇒ EffQueue r t ⊸ [t]
队列转列表 (EffQueue 前端 后端) = 前端 ++ dListToList 后端

入队 :: 区域 r ⇒ EffQueue r t ⊸ t ⊸ EffQueue r t
enqueue(EffQueue front back) x = EffQueue front (back `追加` x)

dequeue :: 区域 r ⇒ EffQueue r t ⊸ Maybe (t, EffQueue r t)
dequeue (EffQueue 前端 后端) = 分情况 前端 of
  [] → 分情况 tokenBesides 后端 of (后端, tok) → 分情况 (toList 后端) of
    [] → 丢弃令牌 ; Nothing
    (x : xs) → Just (x, (EffQueue xs (newDList tok)))
  (x : xs) → Just (x, (EffQueue xs 后端))
```

清单 4.1：扩展 DPS Haskell 中用于线性数据的高效队列

```
1 data Tree t = Nil | Node t (Tree t) (Tree t) 2 3 relabelDps ::Region r ⇒ Token ⊸ Tree t → Tree Int 4
relabelDps tree = fst (mapAccumBfs (\st _ → (st + 1, st)) 1 tree) 5 6 mapAccumBfs :: ∀ r s t u. Region r ⇒
Token ⊸ (s → t → (s, u)) → s → Tree t → (Tree u, s) 7 mapAccumBfs tok f s0 tree =8 分情况
fromAmpar (newAmpar @(Ur (Tree u)) tok `updWith` \du → go s0 (singleton (Ur tree, du &fillUr))) 9 of (Ur
outTree, Ur st) → (outTree, st) 10 where 11 go ::s → EffQueue r (Ur (Tree t), Dest r ω (Tree u)) ⊸ Ur s 12
go st q = 分情况 dequeue q of 13 Nothing → Ur st 14 Just ((utree, dtree), q') → 分情况 utree of 15 Ur Nil →
dtree &fill @'Nil ⨟ go st q' 16 Ur (Node x tl tr) → 分情况 (dtree &fill @'Node) of 17 (dy, dtl, dtr) →18 let 表达式
q'' = q' `入队` (Ur tl, dtl) `入队` (Ur tr, dtr) 19 (st', y) = f st x 20 in dy &fillLeaf y ⨟ go st' q''
```

清单 4.2：在扩展 DPS Haskell 中使用高效队列进行广度优先树遍历

Lindley, 2024；Minamide, 1998]）。另一方面，广度优先树遍历仍然使用类似 `updWith` 和 `fill` 的函数实现，这些函数对于操作队列中存储的目的地至关重要。[35]

支持线性目标的 DPS Haskell 完整最终 API 在清单 4.3 中，我们总结了本章介绍过的所有函数。这些函数对于创建尽可能灵活且仍然安全的 Haskell API，以执行目的地传递而言都是必需的。

撰写论文时，我们尚未实现这套完整的 API，但我们认为实现它应该不会遇到什么特殊的技术问题。事实上，`extend` 操作符的行为直接源自相应的`fill` 函数，而这些函数在 [Bagrel, 2023b] 中已经有了具体实现。

[35]不可能仅使用具有透镜支持的 `extend` 函数来实现广度优先遍历。如果确实可行，则需要更强大的 光学（即包含透镜在内的更广泛抽象族），而生成的代码很可能会更加冗长。

```
1 数据 Ampar r s t 2 newAmpar :: ∀ r s. 区域 r ⇒ 令牌 ⊸ Ampar s (目标 r 1 s) 3 tokenBesides :: ∀ r s t. 区域 r ⇒ Ampar r s t ⊸ (Ampar r s t, 令牌) 4 toAmpar :: ∀ r s. 区域 r ⇒ 令牌 ⊸ s ⊸ Ampar r s () -- 现在对 s 线性 5 fromAmpar :: ∀ r s t. 区域 r ⇒ Ampar r s (Ur t) ⊸ (s, Ur t) 6 fromAmpar' :: ∀ r s. 区域 r ⇒ Ampar r s () ⊸ s 7 updWith :: ∀ r s t u. 区域 r ⇒ Ampar r s t ⊸ (t ⊸ u) → Ampar r s u -- 现在为非捕获 8 updWithToken :: ∀ r s t u. 区域 r ⇒ Ampar r s t ⊸ ((t, 令牌) ⊸ u) → Ampar r s u 9 10 数据 Dest r m t 11 类型族 DestsOf lCtor r m t -- 现在支持多态重数 12 13 -- ==仅适用于非线性目标的 API ==14 fillLeaf :: ∀ r t. 区域 r ⇒ t → 目标 r ω t ⊸ () 15 fillUComp :: ∀ r t u. 区域 r ⇒ Ampar r (Ur t) u ⊸ 目标 r ω t ⊸ u 16 17 -- 适用于具有线性或非线性目标的 ampar 的 API ==18 fill :: ∀ lCtor r m t. (区域 r, OnlyLinear lCtor t) ⇒ 目标 r m t ⊸ DestsOf lCtor r m t 19 fillUr :: ∀ r m t. 区域 r ⇒ 目标 r m (Ur t) ⊸ 目标 r ω t 20 21 -- 右侧只有一个目标（简化版，但表达力较弱） 22 extend :: ∀ lCtor r k t m s fiTys kthFiTy othFiTys. 23 (区域 r, OnlyLinear lCtor t, 24 字段类型 lCtor t ~ fiTys, 25 索引访问 k fiTys ~ kthFiTy, Remove k fiTys ~ othFiTys) ⇒26 othFiTys %m→ Ampar r s (目标 r m t) ⊸ Ampar r s (目标 r m kthFiTy) 27 extendUr :: ∀ r s t m. 区域 r ⇒ Ampar r s (目标 r m (Ur t)) ⊸ Ampar r s (目标 r ω t) 28 extendLeaf :: ∀ r s t m. 区域 r ⇒ t %m→ Ampar r s (目标 r m t) ⊸ Ampar r s () 29 extendComp :: ∀ r s t u. 区域 r ⇒ Ampar r t u ⊸ Ampar r s (目标 r 1 t) ⊸ Ampar r s u 30 extendUComp :: ∀ r s t u. 区域 r ⇒ Ampar r (Ur t) u ⊸ Ampar r s (目标 r ω t) ⊸ Ampar r s u 31 32 -- 右侧可为任意类型；目标由线性透镜聚焦 33 extendFocused :: ∀ lCtor r u t1 t2 s m. 34 (区域 r, OnlyLinear lCtor u) ⇒35 透镜 t1 t2 (目标 r m u) (DestsOf lCtor r m u) → Ampar r s t1 ⊸ Ampar r s t2 36 extendFocusedUr :: ∀ r u t1 t2 s m. (区域 r) ⇒37 透镜 t1 t2 (目标 r m (Ur u)) (目标 r ω u) → Ampar r s t1 ⊸ Ampar r s t2 38 extendFocusedLeaf :: ∀ r u t1 t2 s m. (区域 r) ⇒39 u %m→ 透镜 t1 t2 (目标 r m u) () → Ampar r s t1 ⊸ Ampar r s t2 40 extendFocusedComp :: ∀ r u t1 t2 s u1 u2. (区域 r) ⇒41 Ampar r u1 u2 ⊸ 透镜 t1 t2 (目标 r 1 u1) u2 → Ampar r s t1 ⊸ Ampar r s t 42 extendFocusedUComp :: ∀ r u t1 t2 s u1 u2. (区域 r) ⇒43 Ampar r (Ur u1) u2 ⊸ 透镜 t1 t2 (目标 r ω u1) u2 → Ampar r s t1 ⊸ Ampa
```

2

r r s t2

清单 4.3：支持线性目标的扩展 DPS Haskell API

4.5 结论

本章中，我们花时间探讨了放宽第3章所设限制后会发生什么。该限制规定，目标只能填入非线性数据（因此，ampar 也只能用于存储非线性数据）。

我们谨慎地逐步放宽了这一限制，每一步都避免作用域逃逸及类似风险，从而改进了 DPS Haskell API。首先，我们允许用户构建用于存储线性数据的 ampar，但每个 ampar 只能有一个空洞；这一做法借鉴了基础性研究 ⟦Lorenzen、Leijen、Swierstra 和 Lindley（2024）；Minamide， 1998⟧。由此诞生了一组扩展函数，它们直接作用于 ampar，而非目标。

第二步，我们提出了另一组函数，称为`extendFocused`。这些函数允许用户用线性数据填充任意形状的 ampar，其中空洞或目标的数量不受限制。它们同样直接作用于 ampar，并允许用户在 ampar 内选择要聚焦的目标，通过线性透镜进行操作——这是一种便捷对象，在不可变函数式数据结构中兼具获取器和设置器的作用。

最终，我们得到了一个系统，它在 Haskell 现有类型系统所允许的安全设计范围内，尽可能接近 λ_d（第2章），并且相较于初始 DPS Haskell API 提升了表达能力。当然，填充线性目标时的操作体验不如 λ_d 流畅，但新获得的灵活性使我们得以实现一个修订版的广度优先树遍历。这一次，该算法使用的算法和底层数据结构与第 2.4 节提出的初始版本完全相同。

经过所有这些改进后，我们得到的最终 API 尚未在 Haskell 中实现。不过，我们认为实现它不会特别困难，因为底层操作的大部分艰巨工作已在第 3.5 节完成。

相关工作

在函数式语言安全、受控的内存更新领域，带孔的函数式结构 和目的地传递 是两个密切相关的主题，因为目的地传递 为操作此类结构提供了一种灵活且有原则的方法。不过，尽管两者在概念上有所重叠，函数式目标传递 和带孔的函数式结构 在本质上仍略有不同。

函数式目标传递 涵盖了这样一类技术：通过安全或至少受控的接口，公开对一次写入内存单元的更新。事实上，在以往的工作中，这种接口并未经常直接向程序员公开；相反，在这些情况下，编译器会在内部采用目的地传递 来优化函数式程序。这包括提升函数式链表数据结构（工作 1）和函数式数组（工作 2）的性能：它们采用常见的纯函数式风格编写，却能通过幕后进行的变异减少内存分配、提升性能，而无需用户显式控制。最终，目的地也可用于理论 λ 演算的操作语义中（工作 3）。

相比之下，带孔的函数式结构 着重于允许不完整数据结构（也称带孔结构）在纯函数式环境中作为一等值存在，供用户显式操作，并在之后补全或细化。补全这些结构必然涉及某种形式的内存更新，但许多系统（工作 4 和 5）并未明确引入一等写指针（目标）的概念来进行变异。与之相对，本论文探讨了如何有效地结合带孔的函数式结构 与函数式目标传递 ，利用各自的优势，实现对函数式数据更具表现力的操作模式，同时始终确保安全。

这两个概念都固有地涉及变异和未初始化数据，因此需要强大的静态保证来确保安全性。线性和唯一性类型系统因此常用于对目标或带孔结构施加严格控制，防止对不完整结构进行不安全的更新和过早读取。然而，正如本项工作所展示的，仅凭线性性不足以涵盖我们所需的全部安全性和表达能力。在这种情况下，模态类型系统（Work 6 和 7）尤其有助于我们轻松地为特定需求设计出一致且易用的类型系统。其他互补方法，例如基于权限的类型系统（Work 8）以及生命周期管理技术仿照Rust（Work 9），也在对资源实施细粒度控制方面发挥着重要作用，而这种控制可用于管理我们的目标或带孔结构。

1 构造器模尾递归

⟦Bour、Clément 和 Scherer， 2021⟧（以及 ⟦Allain 等人， 2024⟧, 他们在前者的基础上扩展并给出了正确性证明）关注的是一个长期存在的问题：链表上的映射函数不是尾递归的，因此会消耗栈空间。映射函数可以改为尾递归形式，但代价是需要对列表执行额外的反转

操作。他们指出，对于递归调用仅出现在数据构造器应用之下的函数（例如，映射函数中的 *cons* ），存在一种系统化的转换方法，可将其转换为采用目标传递的尾递归实现。

在这里，用户层没有目标，只有中间表示中才有。不过，程序员可以通过编程接口来指定：为函数添加注解，例如

```
let[@尾部_mod_cons] 递归 map =
```

以要求编译器执行转换。如果无法执行，编译器就会报错，因此优化行为完全可预测。自 4.14 版起，OCaml 编译器便已包含此功能；这是我们所知的生产级函数式编译器中内置目的地的唯一实例。

我们的方法 λ_d 和 DPS Haskell API 都允许在用户层将尾模构造器变换的结果（即目标传递式尾递归程序）写成类型良好的程序，而这在 OCaml 中是不可能的。不过，我们没有提供类似的自动变换方式。

另一方面，构造器模尾递归能力不足，无法处理我们的差分列表，因为它缺少对带孔结构的一等支持；这类结构只能在变换函数的执行期间暂时存在。尾模构造器变换也无法应用于我们的广度优先树遍历示例（参见第 2.4 节和第 3.3.2 节）。

2 目标传递风格与高效内存管理

Shaikhha 等人 ⟦2017⟧ 为一种函数式数组编程语言提出了一种基于目标的中间语言，并配备了目标专属优化，其性能接近 C语言。

这是迄今为止关于目标传递风格对函数式语言性能益处的又一项重要证据，尽管他们的工作面向数组编程，而本博士论文稿则聚焦于链表数据结构。因此，他们可以受益于一些对我们而言或许价值较低的优化，例如为多个数组分配一个连续内存块。

他们的工作与我们的工作之间的另一个区别在于，他们的语言纯粹是一种中间语言：手动使用该语言进行编程是不健全的，因为它并不提供任何特定的安全保证。另一方面，我们提出的是一个理论系统，同时也提供了一个具体的 Haskell API，使程序员能够直接使用目标，并确保其健全性。

我们认为，这两个方面相辅相成：良好的编译器优化对于减轻程序员的负担并实现高级抽象十分重要；而能够在代码中使用目标，则赋予程序员更多控制权，以备需要时使用。

3 半公理化序列演算

在 ⟦DeYoung、Pfenning 和 Pruiksma 的系统中， 2020⟧ 构造器返回到一个目标，而不是分配内存。与本章介绍的其他系统截然不同的是，该系统完全建立在 Curry-Howard 同构之上。具体而言，它为一种混合了 Gentzen 风格的演绎规则和 Hilbert 风格的公理的序列演算提供了一种解释。因此，他们的系统具有完全对称的 *par* 联结词；而且，与我们的 ⌊⊤⌋ 类型不同，他们的对偶化联结词是自反的。

这种优雅的代价是，计算可能会尝试对空洞进行模式匹配，此时必须等待空洞被填充。因此，空洞的语义类似于未来或承诺。反过来，这要求其演算的语义完全并发，而 λ_d 只需要顺序执行模型，以及一种非常温和的状态操作形式（对求值上下文进行线性替换）。

4 带孔数据结构的函数式表示

将线性类型作为函数式演算的基础，使不完整数据结构能够作为一等值存在并组合起来，这一想法可以追溯到 [Minamide, 1998]。无论在理论层面还是在实现中，我们的系统都深受其启发。在 [Minamide, 1998], 带有空洞的一等结构称为孔抽象。孔抽象由一种特殊的线性函数表示，并受到专门的限制。与任何函数一样，在应用孔抽象之前，我们不能对其输出进行模式匹配（也不能将其传递给另一个函数）；但它们还有一项限制：不能对其参数——空洞变量——进行模式匹配，因为该变量只能直接用作数据构造器或其他孔抽象的参数。因此，孔抽象的类型`(T, S)hfun`（一个类型为`S`、缺少一个`T`才能完整的结构）是一种较弱形式的线性函数类型`T ⊸ S`。

在 [Minamide 中, 1998], 结构始终只能表示为具有单个空洞的形式。不过，这项限制相当表面化。作者对此并未发表评论，但我们认为，这项限制的存在只是为了方便阐述：该语言会被降级为一种不含函数抽象的语言，并通过组合子进行组合；而编写一个用于单参数函数组合的组合子更为容易。

然而，我们在第 4.3 节中看到，借助线性透镜，我们实际上可以为具有多个空洞的结构设计组合子（代价是增加一些语法负担）。因此，采用具有多个空洞的数据结构，并不会从根本上改变他们的系统。

更重要的区别在于，他们的系统基于一种线性函数类型，而我们在第 2 章和第 3 章中的系统则利用了线性逻辑“par”类型的表达能力。在经典线性逻辑中，线性蕴涵 `T ⊸ S` 被重新解释为 $S\,T^{\perp}$。同样，我们将 `(T,S)hfun` 重新解释为 $S \parr \lfloor T \rfloor$，或者 `Ampar s (Dest t)` —— 一种较弱的 “par”。

一个关键的结果是，目标——作为空洞的一等表示——会自然地出现在 λ_d 或 DPS Haskell 中，而 [Minamide 1998] 则没有这些目标。这意味着，使用 [Minamide 1998], 可以实现第 2.2.3 节和第 3.3.1 节中基于差分列表和队列的示例，但无法实现第 2.4 节和第 3.3.2 节中的广度优先遍历示例，因为该示例要求能够将目标存储在某种结构中。

不过，我们仍然保留了 Minamide [1998] 对孔抽象施加的主要限制。例如，我们不能对未应用的 `S` in（`(T,S)hfun`）进行模式匹配；因此，在 λ_d中，我们无法直接操作 `S ⅋T` 的左侧 `S`，只能操作右侧 `T`（DPS Haskell 中也有相应的限制）。类似地，孔变量只能用作构造器或孔抽象的参数；这一点在 λ_d 中体现为：操作目标的唯一方式是通过填充操作。

操纵目的地，尤其是存储目的地的能力确实需要付出代价： λ_d 需要引入额外的年龄概念，以确保目的地能在不同作用域中得到可靠使用。若没有这样的机制（如 DPS Haskell），就必须谨慎限制线性数据目的地的使用，以保持可靠性。尽管有这些额外要求，我们的系统——无论是理论上的还是实际的——都比 Minamide [1998]的系统更为通用。其系统可以自然地嵌入 λ_d（见第 2.8 节），也可以用 DPS Haskell 表达（见第 4.2 节）。

5 上下文模尾递归与命令式二叉搜索树的函数式本质

Leijen 和 Lorenzen ⟦2025⟧ 以及 Lorenzen、Leijen、Swierstra 和 Lindley ⟦2024⟧ 近期开展的一系列研究沿着两个主要方向展开。第一个方向通过考察上下文模尾递归变换，对工作 1 进行推广；在这种变换中，上下文可以实例化为各种形式。例如，当上下文是单空洞数据构造器上下文（等价于工作 4 中的孔抽象）时，他们重新得到了构造器模尾递归变换。但该框架的适用范围更广：它还能处理幺半群上下文、半环上下文、指数模态上下文等。因此，它通过形式化且系统化的方法，能够将更广泛的一类通常非尾递归的函数转换为高效的尾递归等价函数，超越了工作 1 所述变换的适用范围。

第二个轴与我们的工作更为接近，聚焦于一阶一空洞数据构造器上下文的语义和表示，延续了 Minamide 的研究脉络（4）。具体而言，他们研究了这类一空洞数据构造器上下文（即带有一个空洞的数据结构）在各种内存模型下的行为，尤其是在构造器上下文不采用线性管理的情况下！这与我们在 λ_d 上的工作中出现的写时复制策略高度相似（参见第 2.6.3 节末尾和第 2.9 节开头），当 ampar 不必采用线性管理时便是如此。然而，他们的方法更进一步：他们在显式内存存储之上定义了形式化的操作语义，并纳入了引用计数。他们还提出了优化措施，通过检测带有空洞的结构是否只有唯一引用，从而在安全的情况下直接对其进行原地修改，以减少复制开销——这让人想起 ⟦Lorenzen、Leijen 和 Swierstra 的 2023⟧中的相关技术。

此外，他们还在 Koka 编程语言中实现了这种混合 策略（他们如此称呼它），并展示了上下文模尾递归 变换在各项基准测试中带来的性能提升。在map 示例中（参见第 3.6.4 节），他们的方法性能与我们的 DPS Haskell 实现相当，甚至更胜一筹。

另一方面，他们的系统在表达能力方面仍与 Minamide 的系统处于同一水平：目标完全是内部机制，不向用户公开。正如他们所承认的，他们的模型无法表示带有多个空洞的结构，因此和 Minamide 的模型一样，也无法支持广度优先树遍历。

6 Oxidizing OCaml

Lorenzen、White 等人 ⟦2024⟧ 提出了对 OCaml 类型系统的扩展，以支持模态。他们的模态沿着三个不同的“轴”划分，其中仿射性和局部性与我们在第2章介绍的重数和年龄相对应。与我们的重数类似，仿射性有两种模态：一次 和多次，不过在 ⟦Lorenzen、White 等人 2024⟧，的体系中，一次 支持弱化，而 λ_d 的 1 重数则是真正线性的（对于目标而言，真正的线性性至关重要，否则我们就可能丢弃它们而不填充相应的空洞，最终读取未初始化内存）。

局部性跟踪作用域。有两种局部性模式：局部 （不会逃逸出函数）和 全局 （可以逃逸出当前函数）。作者将其局部性模式描述为对 Rust 生命周期系统（参见 Work 9）的大幅简化，但它仍满足了他们的需求。

然而，这个系统过于受限，无法像 λ_d 所需的那样精确地跟踪目标的作用域。关键问题在于，如果嵌套作用域中的目标被赋予相同的模式（将无限多个年龄映射到仅有的两种模式时，这种情况不可避免），我们就无法安全地区分它们的生命周期，从而可能重现第 2.3 节中的反例。这表明，在 Modal OCaml 中完整实现 λ_d 很可能不可行。

话虽如此，如果能引入一种机制来防止丢弃模态为一次的资源，那么第3章所述的目标传递 API 中很大一部分（甚至可能全部）仍可在其 OCaml 版本中实现，因为这些 API 仅依赖线性性，而不需要完整的年龄追踪。

Lorenzen、White 等人 [2024] 并非首次尝试在 OCaml 中构建资源控制系统。[Scherer 等人，2017] 探索了将类似 ML 的静态类型语言与另一种具有线性类型和状态的语言相结合，并对引用和原地复用进行了相当明确的阐述。

7 使用分级模态类型进行定量程序推理

Orchard、Liepelt 和 Eades III [2019b] 提出了一个函数式框架，全面支持分级模态类型，并将语言*Granule* 作为该框架的一个实例。分级模态类型指的是具有分级模态的类型系统，即模态由模式索引，例如$!_{\mathbb{m}}$，其中此处称为共效应的模式属于半环，如 [Abel 和 Bernardy, 2020; Ghica 和 Smith，2014]。

他们的框架可以用不同的半环进行实例化，并展示了一个支持细粒度线性性的实际实例（其中使用次数来自 $\mathbb{N} \cup \{\omega\}$，而不是粒度更粗的 $\{1, \omega\}$），还配有用于公共/私有数据的标记系统，以支持敏感数据流分析。此外，他们还引入了模态范围，以及模态上合适的序关系/子类型关系（与我们的类似），从而使模态的使用和组合方式更加灵活。

因此，Granule 似乎是函数式目标传递一个很有前景的目标。尽管他们展示的实例缺少年龄概念或显式的作用域控制，使得完整嵌入 λ_d 可能难以实现，但另一方面，它似乎能够支持我们在第 3 章和第 4 章为 Haskell 开发的许多、甚至可能全部目标传递 API（至少在类型层面上）。

不过，他们的通用框架可以用我们在第 2.5.1 节中开发的半环进行实例化，并作为 λ_d 类型系统的基础。但我们仍需扩展运行时值集合，以涵盖带有空洞和目标的结构；同样，也需要引入一种状态更新机制，例如我们在第 2.6.3 节中用于求值上下文的线性替换。

8 Mezzo 中的权限编程

Protzenko 和 Pottier [2013] 引入了 Mezzo 编程语言。该语言采用面向用户的可变数据结构（与我们的语言不同，我们的语言将可变性隐藏起来）。Mezzo 的可靠性由一个强大的能力系统保障，该系统尤其规定了如何访问和共享可变结构或引用。在这一点上，Mezzo 与 Rust 编程语言十分相似。

与我们的工作尤其相关的是，Mezzo 能够在构造完成后，安全地将可变数据结构“冻结”为不可变数据结构。例如，Mezzo 标准库中的列表便运用了这一原则：不可变列表上的高效 map 函数通过目标传递和内部使用可变列表来实现，

以此完成目标传递。能力系统保证了在这一底层实现中对目标的安全、正确使用。在 ⟦Larus, 1989⟧ 中，也能找到一个较早的例子：在可变环境中将目标传递用作性能优化。

9 Rust 生命周期

Rust 编程语言采用了一套生命周期系统（例如参见 ⟦Pearce, 2021⟧），其作用与我们在 λ_d（第2章）中使用的年龄系统类似。

Rust 生命周期是符号化的。借用和移动会对符号化生命周期产生约束（形式为 $\alpha \leqslant \beta$ 的不等式）。例如，引用的生命周期必须长于存储该引用的任何结构的生命周期。若不强制执行这些约束，Rust 就会面临第 2.3 节所述的同类问题：借用可能超出其所引用数据的生命周期；同样，如果未能妥善阻止作用域逃逸，我们系统中的目标也可能超出其所指向结构的生命周期。在 Rust 中，借用检查器必须检查这些约束是否有解；而在 λ_d 中，年龄是显式设定的，无需进行分析。

这两个系统的另一个区别是 λ_d的年龄（以及一般情况下的模态）是相对的。若某个部分的年龄与其父级不同，就必须使用显式模态$!_{\uparrow^k}$，这表示该部分的作用域比父级 k 更早。另一方面，Rust 的生命周期是绝对的，某个部分的生命周期会独立于其父级的生命周期进行跟踪。

结论

贡献

历经这段奇特却令人兴奋的旅程，我们可以自信地说：安全的函数式目标传递并非自相矛盾——它确实存在！

目标传递的核心，是传递一个写指针，让函数将结果直接写入内存单元，而不是将其返回。这种方法通过避免中间分配提升性能，并通过支持以新颖方式构造数据来增强表达能力——前者主要体现在命令式上下文中，后者则主要体现在函数式上下文中。然而，乍看之下，目标传递似乎相当不纯：它与内存变异紧密耦合，表面上也与函数式纯性格格不入。

我们在 Haskell 中进行的早期实验——实际上早于 λ_d 的形式化发展——让我们意识到，线性类型可以驯服目标的不纯特性。线性性可以确保一个目标恰好使用一次，从而保证值一旦写入内存单元，就不会被覆盖。这使目标传递与不可变性相容，而不可变性正是函数式编程的基石。

然而，不久之后我们发现了作用域逃逸，并由此意识到，即使对这些目的地强制执行严格的线性纪律，线性数据的目的地仍会给类型安全带来重大挑战。当线性资源被存储在线性容器中，而该容器的生命周期超出引入该资源的作用域时，就会发生作用域逃逸。线性接口通常依赖这样的假设：每种资源类型都配有一组特定的类似析构函数的函数，并且每个资源都必须调用其中一个。如果资源类型是不透明的，线性性会迫使我们在资源变得可用的作用域内将其消耗掉，而唯一的办法就是调用指定的析构函数之一。但如果我们可以通过在线性方式将资源存储在其作用域之外来伪造资源已被消耗——也就是执行作用域逃逸——而不调用析构函数，那么这一预期就会被打破。在我们的例子中，当来自父作用域的目的地`Dest m t`（即线性资源）被填入目的地`Dest 1 (Dest m t)`（即线性容器）时，就会发生作用域逃逸。但作用域逃逸并非目的地特有的问题：它会威胁任何依赖线性性来保障安全性的资源 API。反过来说，只要采用线性接口，任何可写形式的存储都可能引入作用域逃逸，无论是目的地、OCaml 风格的`ref`等。

在早期的 DPS Haskell 中，我们通过彻底禁止线性数据的目的地来避免作用域逃逸。即便如此，我们仍然可以从目的地传递中获益不少：实现尾递归的 map、高效的差分列表，以及其他涉及一等带孔结构的模式。

然而，这一限制削弱了表达能力。正如广度优先遍历示例（第3章）所示，如果没有线性数据的目的地，就无法表达某些模式。显然，我们需要一种更完整的解决方案——在兼顾线性数据的目的地与内存安全的同时，化解二者之间的矛盾。

为了取得进展，我们暂时搁置了实现方面的考量，转而设计了一种理论演算，λ_d，该演算以目的地传递为核心，因此甚至不需要传统的函数式数据构造器。其关键成果是对 λ_d 的可靠性进行了机械化证明。为实现这一点，我们的演算依赖于几个显著特性：

· 一种类似于线性 Haskell 中的线性类型系统，可确保目标既不会被复制，也不会被丢弃——使我们能够在静态阶段判断一个结构何时已被完全填充（发生作用域逃逸时除外）；

· 一种年龄系统，可精确追踪引入资源的作用域。年龄系统是用于防止作用域逃逸问题的主要技术手段；

· 一种模态类型系统，将线性性与年龄控制结合到同一个模态半环中，其灵感来自近期文献中的相关工作；

· 一种针对带孔结构的新颖类型规则，其中结构中的空洞及其关联的目标均表示为类型上下文中的命名绑定。当孔绑定和目的地绑定具有相同名称时，它们会相互抵消。该机制使我们能够在静态阶段确保带孔结构中未使用的目标数量恰好等于空洞数量——不多不少；

· 一种基于显式操作求值上下文的操作语义。具体而言，我们使用对求值上下文进行线性替换的方式，来建模一种轻量级的状态更新形式。与基于存储的语义相比，我们的方法更简单，同时仍能忠实地捕捉目标引发的内存变更行为；

有了这一理论基础，我们便可以回到最初的动机：在 Haskell 中实践目的地传递。Haskell 对线性类型的支持使其成为自然之选[36]，，但由于缺乏年龄追踪，我们不得不探索新的权衡。具体而言，我们需要在目标传递 API 的简洁性与所需的表达能力之间取得平衡。在 λ_d 中，两者兼得——但这需要借助一种专门设计、具有年龄概念的类型系统。

在第3章和第4章中，我们逐步探索了 Haskell 中功能日益强大的目标传递 API，每一种都进一步缩小了与 λ_d 的差距。到第4章结束时，我们恢复了一个表达能力足够强的系统，能够实现 λ_d 中所有作为动机的示例，所需的调整仅仅是将一种语法转换为另一种语法。

我们工作的最后一项重要贡献同样具有实际意义。在开发 DPS Haskell 的过程中，我们在紧凑区域中实现了该 API——这是一块位于 Haskell 堆中的独立内存空间，垃圾回收器对它的管理相对宽松。我们最初的动机是确保目的地填充原语引起的内存变更不会干扰垃圾回收器和运行时系统所要求的不变式。然而，这一设计选择带来了我们未曾预料到的其他好处。紧凑区域 常用于更高效地存储长生命周期的数据结构，避免对它们反复执行垃圾回收。事实上，我们的 DPS Haskell API 提供了一种更直接、更高效的方式，能够在紧凑区域中安全地分配这类结构。第 3.6 节中的基准测试支持了这一观点：在紧凑区域内实现的目的地传递，在特定场景中是一种很有价值的工具；它既能在需要时支持更直接、类似命令式的编程（而不损害内存安全），也能让紧凑区域得到更高效的利用，例如我们在第 3.5.2 节开发的解析器示例。

[36]选择 Haskell 也与本博士研究的工业背景有关：研究经费来自一家将工业级函数式编程融入其基因的公司；此外，我的导师 Arnaud Spiwack 也参与了线性 Haskell ⟦Bernardy 等，2018；Spiwack，2023a；Spiwack 等人，2022⟧。

在我们探索过的所有方法中，第3章提出的方法或许最适合在实践中采用，在简洁性与易用性之间取得了良好平衡。λ_d背后的构想——通过目的地传递来构建每个数据结构——与传统函数式编程风格大相径庭。这种差异源于我们必须严谨地探索目的地传递的理论基础和安全性考量，但也意味着，λ_d 所关注的问题在日常编程的广阔领域中相当小众。与此同时，第4章开发的 API 致力于拓展仅依靠线性性作为安全机制所能实现的功能；但以目前的形式来看，它过于复杂且杂乱，不适合作为工业级库。它展示了我们所取得成果的广度，但仍有可能进一步简化和完善。

带着这些思考，我们希望通过一段植根于类型理论、语言设计和实际实现问题的旅程，为读者清晰而愉快地概述函数式目标传递。

未来展望

仍有许多值得探索的方向，可以进一步完善这项工作。最直接的方向或许是继续改进现有的 DPS Haskell 实现。具体而言，这包括精细调优其性能，而性能仍是这一方法的主要卖点之一。鉴于它还兼具目的地传递接口和更优的紧凑区域分配机制这双重作用（如上文所述），这一点就更加重要。

从更长远的时间尺度来看，正如下文所述，我们可以设想在实践和理论两方面进行进一步改进。

为带有非打包字段的数据结构提供目的地传递支持 改进的第一个方向是将目标传递支持扩展到带有非打包字段的数据结构，即直接嵌入子元素而非指向子元素的字段。

到目前为止，我们一直将注意力限制在纯链式结构上，这是有充分理由的：它们能够高效地组合（即合并）带孔结构。

在我们的理论研究和实现中，我们的目标一直是通过一次简单的内存写入完成目标填充。此外，自 Minamide 的奠基性工作以来，可组合性一直是带孔结构的一项关键特性，对于构建基于 ampar 的差分列表等结构至关重要。

使用`extendComp` 组合两个 ampar（或使用`fillComp`将父 ampar 的目标填入子 ampar）时，如果涉及的字段是间接寻址的，那么组合归结为将子级的地址写入父级的空洞中。见下文（我们使用`@addr =`语法标注内存位置，并用 表示空洞）：

```
1 extendComp :: ∀ s t u. Ampar t u ⊸ Ampar s (Dest 1 t) ⊸ Ampar s u 2 3 父级 Ampar :: Ampar [t] (Dest 1 [t]) = newAmpar tok1 &extend @'(:) @1 1 4 -- 内存中: Ampar 0x05ba (Dest 0x05ca) 5 -- @0x05ba = 0xcb1e : 6 -- @0xcb1e = I# 1# 7 8 子 ampar :: Ampar [t] (Dest 1 [t]) = newAmpar tok2 &extend @'(:)@1 2 9 -- 内存中: Ampar 0x0840 (Dest 0x0850) 10 -- @0x0840 = 0xbc4e : 11 -- @0xbc4e = I# 2# 12 13 resAmpar :: Ampar [t] (Dest 1 [t]) = parentAmpar &extendComp childAmpar 14 -- 内存中: Ampar () (Dest 0x0850) 15 -- @0x05ba = 0xcb1e : 0x0840 16 -- @0xcb1e = I# 1# 17 -- @0x0840 = 0xbc4e : 18 -- @0xbc4e = I# 2#
```

要执行组合，只需更新*cons* 构造器在`0x05ba` 处的第二个字段。生成的 ampar 继承自`childAmpar`的目标，它仍然正确地指向`0x0850`处的空洞。

然而，涉及非打包字段时，情况会变得更加复杂：

```
extendComp :: ∀ s t u。 ampar t u ⊸ ampar s (Dest 1 t) ⊸ ampar s u

-- GHC 不支持多态字段使用 UNPACK，因此我们无法这样做：-- data Pair
a b =Pair{-# UNPACK #-} !a {-# UNPACK #-} !bdataIntPair  = IntPair {-#
UNPACK #-} !Int {-# UNPACK #-} !Intdata NestedPair  = NestedPair {-#
UNPACK #-}!Int {-# UNPACK #-} !IntPair

parentAmpar :: Ampar NestedPair (Dest 1 IntPair)
  = newAmpar tok1 &扩展 @'NestedPair @11
-- 内存中：Ampar 0x05ba(Dest 0x05ca)
-- @0x05ba =   NestedPair 1# □

childAmpar :: Ampar IntPair (Dest 1 Int) =newAmpar tok2 &扩展 @'IntPair @1
2-- 内存中：Ampar 0x0840(Dest 0x0850)--@0x0840 =   IntPair2#
                              □

resAmpar ::Ampar NestedPair (Dest 1 Int) = parentAmpar &extendComp childAmpar -- 内存中：Ampar 0x05ba(Dest 0x0850) -- @0x05ba = NestedPair 1#2# -- @0x0840 = IntPair 2#
                              □
                              □
```

现在，要执行组合，我们必须将子结构的字段复制到父级的空洞中，因为父级字段被标注为已解包。例如，这里的有效载荷 2# 已复制到0x05ca处的空洞中。但这样做会使从子结构继承的目标失效：它仍指向0x0850，而resAmpar 中剩余的空洞现在位于0x5d2！

要解决这一问题，就需要在整个非打包字段中每当发生组合时，动态跟踪并更新目标指针。但这种机制会带来显著开销，并削弱我们当前模型的简洁性和效率。

一种更切实可行的未来方向——我们很可能会在 ⟦Bagrel, 2024b⟧ 中采用——是区分指向非打包字段的目标与指向间接字段的目标。[37] 随后，我们会将 fillComp 和 extendComp 的使用限制在目标指向间接字段的情况。此方法既保留了目标传递和不完整结构组合的大部分表达能力，也支持非打包字段。新限制仅适用于这些此前不受支持的非打包字段，不会影响系统的其余部分，因此目标仍可保持为简单指针，无需复杂的更新机制。

DPS Haskell，紧凑区域之外 未来工作的另一个方向，是直接在 Haskell 的标准垃圾回收堆中实现目的地传递支持，而不是依赖紧凑区域提供的保护边界。

主要挑战在于内存安全与 GHC 垃圾回收器内部机制之间的紧密交互，尤其是对不可变性假设的依赖时机和位置。GHC 使用复制式垃圾回收器，并采用多种对性能至关重要的优化；若违反其预期

[37]字段是否被拆包或间接引用，可以使用 GHC。Generics。

的不可变性假设，可能会引发微妙且潜在后果严重的错误。例如，如果带孔结构被垃圾回收器移动，那么任何引用这些结构的存活目标都必须相应更新，以避免产生悬空指针。

另一项挑战在于，GHC 的运行时系统十分复杂且仍在积极演进，关键实现细节往往分散在各种文档中，因此要全面且及时地理解这一系统，可能需要编译器和运行时领域经验丰富的贡献者提供意见。

不过，使 DPS Haskell 实现能够直接在标准垃圾回收堆上运行，或许可以显著降低采用门槛。程序员将能够有选择地使用目标传递，而无需承担与紧凑区域相关的开销和限制。这也可能带来显著的性能提升，因为不再需要将数据复制到紧凑区域中。例如，在我们使用目标的 map 函数中，输入列表中的每个值目前都必须复制到区域中，尽管只有脊柱需要目标传递。这一方向仍不确定——运行时影响十分微妙，工程工作量也不小——但如果能够实现，其回报可能非常可观。

实用 DPS Haskell API 的安全性证明 在攻读博士学位期间，我们实现了两个主要目标：证明函数式目标传递可以做到安全，并为以下观点提供证据：在工业级函数式语言中，目标传递的实际实现是可行的，并且能够带来性能优势。

最后一个顺理成章的步骤，是将这两条研究脉络统一起来：利用 λ_d （第2章）中建立的理论框架，证明第 3 章和第 4 章介绍的实际实现确实安全。

要完成这样的证明，需要将 DPS Haskell API 中的每个运算符转换为 λ_d 中对应的运算符，并证明对于任意类型正确的 DPS Haskell 程序，都存在合适的年龄赋值，使转换后的 λ_d 程序能够通过类型检查。这里要回顾的是，尽管 Haskell 以按需调用著称，我们的 DPS 实现依赖紧凑区域，在这些区域中，任何数据结构都会被强制求值至范式，这与 λ_d 的按值调用语义相符。因此，在这方面应该不存在不匹配。

不过，主要障碍在于，尽管 λ_d 和各种 DPS Haskell API 有着共同的根源，但它们之间存在一些细微却关键的差异。例如，在第 3 章的版本中，由于线性数据没有目的地，这会产生深远影响：目的地年龄各不相同的数据可以安全地存储在常见的 Haskell 数据结构中，例如标准库列表。这种灵活性在 λ_d 中很难得到合理解释，因为其中没有非基于目标的数据结构这一概念。同样，在第 4 章的 API 中，我们既依赖非捕获的 `updWith` 运算符（不同于原始的 `upd`$\ltimes$），也依赖 `extend` 函数；后者实际上会转换为一种受控但会捕获的 `upd`$\ltimes$ 版本。

因此，关键或许不在于原样复用 λ_d，而在于设计一种略作调整、针对我们要验证的特定 API 量身打造的形式语言。现有形式化工作的大部分内容——尤其是用于管理类型上下文、洞重命名和替换的基础设施——很可能只需少量修改即可复用。虽然原有证明（第 2.7, ⟦Bagrel和 Spiwack，2025b⟧）并非以可复用性为目标编写（因为我们原本认为 λ_d 以当前形式已经足够通用），但只有相对较小的一部分直接依赖于各个运算符的类型规则。例如，将这些规则调整为与第 3 章 DPS Haskell API 中的规则相符，会耗费不少时间，但很可能是可行的。作为回报，这条路径为我们提供了一个颇具前景的机会，可以推广我们的形式化开发并扩大其影响。

参考文献

Abel, Andreas and Bernardy, Jean-Philippe（2020年8月）。“类型系统中模态的统一视图。”载于：《ACM编程语言论文集》4.ICFP。doi：10.1145/3408972。网址：https://doi.org/10.1145/3408972。Allain, Clément 等（2024年11月）。尾递归模 Cons、OCaml 与关系分离逻辑。英文。doi：10.1145/3704915。网址：https://arxiv.org/abs/2411.19397v1（访问于 01/08/2026）。Andreoli, Jean-Marc（1992年6月）。“线性逻辑中运用聚焦证明的逻辑编程。”载于：《逻辑与计算期刊》2.3，第 297–347 页。issn：0955-792X。doi：10.1093/logcom/2.3.297。网址：https://doi.org/10.1093/logcom/2.3.297（访问于 12/05/2024）。Atkey, Robert（2018）。“定量类型理论的语法与语义。”载于：第 33 届 ACM/IEEE计算机科学逻辑研讨会论文集。LICS '18。英国牛津：计算机协会，第 56–65 页。isbn：9781450355834。doi：10.1145/3209108.3209189。网址：https://doi.org/10.1145/3209108.3209189。Bagrel, Thomas（2023a）。支持空心构造器分配的 GHC。Software Heritage，swh:1:dir:84c7e717fd5f189c6b6222e0fc92d0a82d755e7c；origin=https://github.com/tweag/ghc；visit=swh:1:snp:141fa3c28e01574deebb6cc91693c75f49717c32；anchor=swh:1:rev:184f838b352a0d546e574bdeb83c8c190e9dfdc2。（访问于 10/19/2023）。Bagrel, Thomas（2023b）。linear-dest，一个为 DPS 编程提供支持的 Haskell 库。Software Heritage，swh:1:rev:0e7db2e6b24aad348837ac78d8137712c1d8d12a；origin=https://github.com/tweag/linear-dest；visit=swh:1:snp:c0eb2661963bb176204b46788f4edd26f72ac83c。（访问于 10/19/2023）。Bagrel, Thomas（2024年1月）。“目标传递风格编程：一种 Haskell 实现。”载于：第 35 届法语应用语言日（JFLA 2024）。法国圣雅屈德拉梅尔。网址：https://inria.hal. science/hal-04406360。Bagrel, Thomas（2024年11月）。tbagrel1 提交的零拷贝紧凑区域基本操作：拉取请求 #683 · ghc-proposals/ghc-proposals。英文。网址：https : / / github . com /ghc - proposals / ghc - proposals / pull / 683（访问于 04/01/2025）。Bagrel, Thomas and Spiwack, Arnaud（2025年4月）。“目的地演算：纯函数式内存写入的线性 -演算。”载于：《ACM编程语言论文集》9.OOPSLA1。doi：10.1145/3720423。网址：https://doi. org/10.1145/3720423。Bagrel, Thomas and Spiwack, Arnaud（2025年4月）。目的地演算：使用 Coq证明助手证明进展性与保持性。doi：10.5281/zenodo.14982363。网址：https://doi.org/10.5281/zenodo.14982363。Bernardy, Jean-Philippe 等（2018年1月）。“线性 Haskell：高阶多态语言中的实用线性性。”载于：《ACM编程语言论文集》2.POPL。arXiv:1710.09756 [cs], pp. 1–29。issn：2475-1421。doi：10.1145/3158093。网址：http://arxiv.org/abs/1710.09756（访问于 06/23/2022）。Bierman, G.M.（1994年8月）。论直觉主义线性逻辑。技术报告。UCAM-CL-TR-346。剑桥大学计算机实验室。doi：10.48456/tr-346。网址：https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-346.pdf。

Biernacka, Małgorzata and Danvy, Olivier（2007年5月）。“上下文敏感演算与抽象机之间的语法对应。”载于：理论计算机科学。纪念 John C. Reynolds 七十寿辰专辑 375.1，第 76–108 页。issn：0304-3975。doi：10.1016/j.tcs.2006.12.028。网址：https://www.sciencedirect.com/science/article/pii/S0304397506009170（访问于 12/17/2024）。Bour, Frédéric, Clément, Basile, and Scherer, Gabriel（2021年2月）。“尾递归模 Cons。”载于：*arXiv:2102.09823 [cs]*。arXiv：2102.09823。网址：http://arxiv.org/abs/2102.09823（访问于 03/22/2022）。Brunel, Aloïs 等（2014）。“核心定量共效应演算。”英文。载于：编程语言与系统。Zhong Shao 编。柏林、海德堡：Springer，第 351–370 页。isbn：978-3-642-54833-8。doi：10.1007/978-3-642-54833-8_19。Curien, Pierre-Louis and Herbelin, Hugo（2000）。“计算的对偶性。”载于：第五届 *ACM SIGPLAN*国际函数式编程会议论文集。ICFP '00。美国纽约：计算机协会，第 233–243 页。isbn：1581132026。doi：10.1145/351240.351262。网址：https://doi.org/10.1145/351240.351262。Danvy, Olivier and Nielsen, Lasse R.（2004年11月）。“归约语义中的重新聚焦。”英文。载于：*BRICS* 报告系列 11.26。issn：1601-5355，0909-0878。doi：10.7146/brics.v11i26.21851。网址：https://tidsskrift.dk/brics/article/view/21851（访问于 12/17/2024）。DeYoung, Henry, Pfenning, Frank, and Pruiksma, Klaas（2020）。“半公理化序列演算。”载于：第五届计算与演绎形式结构国际会议（*FSCD 2020*）。Zena M. Ariola 编。第 167 卷。莱布尼茨国际信息学论文集（LIPIcs）。德国达格施图尔：达格施图尔城堡—莱布尼茨信息学中心，29:1–29:22。isbn：978-3-95977-155-9。doi：10.4230/LIPIcs.FSCD.2020.29。网址：https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSCD.2020.29。Felleisen, Matthias（1987）。“lambda-nu-cs 转换演算：命令式高阶编程语言中控制与状态的语法理论。”AAI8727494。博士学位论文。美国：印第安纳大学。网址：https://www2.ccs.neu.edu/racket/pubs/dissertation-felleisen.pdf。Ghica, Dan R. and Smith, Alex I.（2014）。“资源半环中的有界线性类型。”英文。载于：编程语言与系统。Zhong Shao 编。柏林、海德堡：Springer，第 331–350 页。isbn：978-3-642-54833-8。doi：10.1007/978-3-642-54833-8_18。Gibbons, Jeremy（1993）。“线性时间广度优先树算法：折叠与压缩的算术练习。”英式英语。载于：第 71 号。编号：第 71 号。网址：https://www.cs.ox.ac.uk/publications/publication2363-abstract.html（访问于 10/18/2023）。Gibbons, Jeremy 等（2023年8月）。“软件架构中的阶段。”英文。载于：第一届 *ACM SIGPLAN*函数式软件架构国际研讨会论文集。美国华盛顿州西雅图：ACM，第 29–33 页。isbn：9798400702976。doi：10.1145/3609025.3609479。网址：https://dl.acm.org/doi/10.1145/3609025.3609479（访问于 10/02/2023）。Girard, J.-Y.（1995）。“线性逻辑：其语法与语义。”英文。载于：线性逻辑进展。Jean-Yves Girard、Yves Lafont 和 Laurent Regnier 编。剑桥：剑桥大学出版社，第 1–42 页。isbn：978-0-511-62915-0。doi：10.1017/CBO9780511629150.002。网址：https://www.cambridge.org/core/product/identifier/CBO9780511629150A008/type/book_part（访问于 03/21/2022）。Girard, Jean-Yves（1987）。“线性逻辑。”载于：理论计算机科学 50.1，第 1–101 页。issn：0304-3975。doi：https://doi.org/10.1016/0304-3975(87)90045-4。网址：https://www.sciencedirect.com/science/article/pii/0304397587900454。Girard, Jean-Yves, Scedrov, Andre, and Scott, Philip J.（1992年4月）。“有界线性逻辑：一种实现多项式时间可计算性的模块化方法。”载于：理论计算机科学 97.1，第 1–66 页。issn：0304-3975。doi：10.1016/0304-3975(92)90386-T。网址：https://www.sciencedirect.com/science/article/pii/030439759290386T（访问于 12/05/2024）。Harrington, Dana（2006年3月）。“唯一性逻辑。”英文。载于：理论计算机科学 354.1，第 24–41 页。issn：03043975。doi：10.1016/j.tcs.2005.11.006。网址：https://linkinghub.elsevier.com/retrieve/pii/S0304397505008522（访问于 03/24/2025）。

Hood, Robert 和 Melville, Robert（1981 年）。“纯 LISP 中的实时队列操作。”载于：《信息处理快报》13.2，第 50–54 页。ISSN：0020-0190。DOI：https : / / doi . org / 10 . 1016 / 0020 - 0190(81) 90030 - 2。URL： https : // www.sciencedirect.com/science/article/pii/0020019081900302。Howard, W H（1969年）。公式即类型的构造观念。英文。Armando B. Matos 于 2017 年转录为 LaTeX。URL：https://www.dcc.fc.up.pt/~acm/howard2.pdf（访问于 2025 年 5 月 5 日）。Huet, Gérard（1997年9月）。“拉链。”英文。载于：《函数式编程杂志》7.5，第 549–554 页。ISSN：1469-7653，0956-7968。DOI：10.1017/S0956796897002864。URL：
h
t
t
p
s://
w
w
w.
cambridge.org/core/journals/journal-of-functional-programming/article/zipper/0C058890B8A9B588F26E6D68CF0CE204（访问于 2025 年 5 月 20 日）。Hughes, Jack、Marshall, Daniel 等（2021年）。“作为分级模态类型的线性指数模态。”英文。载于。Hughes, Jack、Vollmer, Michael 和 Orchard, Dominic（2021年12月）。“推导分级线性类型的分配律。”载于：《理论计算机科学电子论文集》353。arXiv：2112.14966 [cs], pp. 109–131。ISSN：2075-2180。DOI：10.4204/EPTCS.353.6。URL：http://arxiv.org/abs/2112.14966（访问于 2025 年 12 月 9 日）。Hughes, John（1986年1月）。“列表的一种新颖表示及其在函数 ‘reverse’ 中的应用。”载于：Inf. Process. Lett. 22，第 141–144 页。Larus, James Richard（1989年）。“重构符号程序，以便在多处理器上并发执行。”AAI9006407。博士学位论文。加州大学伯克利分校。Leijen, Daan 和 Lorenzen, Anton（2025年）。“上下文模尾递归：一种等式方法（扩展版）。”载于：正投稿至 JFP’25（）。URL：https://antonlorenzen.de/trmc-jfp.pdf（访问于 2025 年 5 月 13 日）。Lorenzen, Anton、Leijen, Daan 和 Swierstra, Wouter（2023年8月）。“FP^2：完全原地函数式程序。”英文。载于：《ACM编程语言论文集》7.ICFP，第 275–304 页。ISSN：2475-1421。DOI：10.1145/3607840。URL：https://dl.acm.org/doi/10.1145/3607840（访问于 2023 年 12 月 11 日）。Lorenzen, Anton、Leijen, Daan、Swierstra, Wouter 和 Lindley, Sam（2024年6月）。“命令式二叉搜索树的函数式本质。”载于：Proc. ACM Program. Lang. 8.PLDI。DOI：10.1145/3656398。URL：https: //doi.org/10.1145/3656398。Lorenzen, Anton、White, Leo 等（2024年8月）。“使用模态内存管理改造 OCaml。”载于：Proc. ACM Program. Lang. 8.ICFP，253:485–253:514。DOI：10.1145/3674642。URL：https://dl.acm.org/doi/10.1145/3674642（访问于 2025 年 1 月 6 日）。Marshall, Danielle 和 Orchard, Dominic（2022年）。“如何求类型的逆。”载于：第 36 届欧洲面向对象编程会议（ECOOP 2022）。Karim Ali 和 Jan Vitek 编。第 222 卷。《莱布尼茨国际信息学论文集》（LIPIcs）。ISSN：1868-8969。德国达格施图尔：达格施图尔城堡—莱布尼茨信息学中心，5:1–5:27。ISBN：978-3-95977-225-9。DOI：10 . 4230 / LIPIcs . ECOOP . 2022 . 5。URL：https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2022.5（访问于 2025 年 12 月 9 日）。Marshall, Danielle、Vollmer, Michael 和 Orchard, Dominic（2022年）。“线性性与唯一性：一场友好协定。”英文。载于：《编程语言与系统》。Ilya Sergey 编。第 13240 卷。丛书名：《计算机科学讲义》。Cham：施普林格国际出版公司，第 346–375 页。ISBN：978-3-030-99335-1 978-3-030-99336-8。DOI：10.1007/978-3-030-99336-8_13。URL：https://link.springer.com/10.1007/978-3-030-99336-8_13（访问于 2025 年 3 月 24 日）。Minamide, Yasuhiko（1998年1月）。“带孔数据结构的函数式表示。”载于：第 25 届 ACM SIGPLAN-SIGACT编程语言原理研讨会论文集。POPL ’98。美国纽约州纽约市：计算机协会，第 75–84 页。ISBN：978-0-89791-979-1。DOI：10.1145/268946.268953。URL：https://doi.org/10.1145/268946.268953（访问于 2022 年 3 月 15 日）。Okasaki, Chris（2000年9月）。“广度优先编号：从一个小型算法设计练习中得到的启示。”载于：第五届 ACM SIGPLAN函数式编程国际会议论文集。ICFP ’00。美国纽约州纽约市：计算机协会，第 131–136 页。ISBN：978-1-58113-202-1。DOI：10.1145/351240.351253。URL：https://dl.acm.org/doi/10.1145/351240.351253（访问于 2023 年 10 月 12 日）。Orchard, Dominic、Liepelt, Vilem-Benjamin 和 Eades III, Harley（2019年7月）。“使用分级模态类型进行定量程序推理。”载于：面向使用分级模态类型进行定量程序推理的语言实现，3.ICFP，110:1–110:30。DOI：10.1145/3341714。URL：https://dl.acm.org/doi/10.1145/3341714（访问于 2024 年 12 月 5 日）。

Orchard, Dominic、Liepelt, Vilem-Benjamin 和 Eades III, Harley（2019年7月）。“使用分级模态类型进行定量程序推理。”载于：面向使用分级模态类型进行定量程序推理的语言实现，3.ICFP，110:1–110:30。DOI：10.1145/3341714。URL：https://dl.acm.org/doi/10.1145/3341714（访问于 2025 年 6 月 2 日）。Pearce, David J.（2021年4月）。“Rust 中引用生命周期与借用的轻量级形式体系。”载于：ACM《程序设计语言与系统汇刊》43.1，3:1–3:73。ISSN：0164-0925。DOI：10.1145/3443420。URL：https://dl.acm.org/doi/10.1145/3443420（访问于 2025 年 1 月 15 日）。Petricek, Tomas、Orchard, Dominic 和 Mycroft, Alan（2014年8月）。“共效应：上下文依赖计算演算。”载于：第 19 届 ACM SIGPLAN函数式编程国际会议论文集。ICFP '14。美国纽约州纽约市：计算机协会，第 123–135 页。ISBN：978-1-4503-2873-9。DOI：10.1145/2628136.2628160。URL：https://dl.acm.org/doi/10.1145/2628136.2628160（访问于 2024 年 12 月 5 日）。Pierce, Benjamin C.（2002年）。《类型与程序设计语言》。第 1 版。麻省理工学院出版社。ISBN：0262162091。Protzenko, Jonathan 和 Pottier, François（2013年9月）。“在 Mezzo 中进行权限编程。”载于：第 18 届 ACM SIGPLAN函数式编程国际会议论文集。arXiv：1311.7242 [cs], pp. 173–184。DOI：10.1145/2500365.2500598。URL：http://arxiv.org/abs/1311.7242（访问于 2023 年 10 月 16 日）。Ramsey, Norman、Jones, Simon Peyton 和 Lindig, Christian（2005年2月）。“C–语言规范 2.0 版（CVS 修订版 1.128）。”英文。载于。Scherer, Gabriel 等（2017年7月）。ML 与线性语言的 FabULous 互操作性。英文。URL：https : //arxiv.org/abs/1707.04984v3（访问于 2026 年 1 月 8 日）。Sewell, Peter 等（2007年）。“Ott：面向语义学研究者的有效工具支持。”载于：第 12 届 ACM SIGPLAN国际函数式编程会议论文集。ICFP '07。德国弗赖堡：计算机协会，第 1–12 页。ISBN：9781595938152。DOI：10.1145/1291151.1291155。URL：https://doi.org/10.1145/1291151.1291155。Shaikhha, Amir 等（2017年9月）。“用于高效内存管理的目标传递风格。”英文。载于：第 6 届 ACM SIGPLAN函数式高性能计算国际研讨会论文集。英国牛津：ACM，第 12–23 页。ISBN：978-1-4503-5181-2。DOI：10.1145/3122948.3122949。URL：https://dl.acm.org/doi/10.1145/3122948.3122949（访问于 2022 年 3 月 15 日）。Sørensen, Morten Heine B. 和 Urzyczyn, Paweł （ 2006 年 7 月）。《 Curry-Howard 同构讲义》。英语（美国）。第 1 版。第 149 卷。《逻辑与数学基础研究》。爱思唯尔科学出版社。ISBN：978-0-444-52077-7。URL：https://www.cs.cmu.edu/~rwh/courses/clogic/www/handouts/curry-howard.pdf（访问于 2025 年 5 月 5 日）。Spiwack, Arnaud（2023年11月）。由 aspiwack 提出的线性约束提案 · 拉取请求 #621 · ghc-proposals/ghc-proposals。英文。URL：https://github.com/ghc-proposals/ghc-proposals/pull/621（访问于 2025 年 3 月 25 日）。Spiwack, Arnaud（2023年3月）。线性约束：作用域问题。英文。博客。URL：https://tweag.io/blog/2023-03-23-linear-constraints-linearly/（访问于 2025 年 3 月 25 日）。Spiwack, Arnaud（2024年1月）。Ur 不能是 newtype。英文。博客。URL：https://www.tweag.io/blog/2024-01-18-linear-desugaring/#ur-cant-be-a-newtype（访问于 2025 年 4 月 1 日）。Spiwack, Arnaud 等（2022年8月）。“线性限定类型：针对能力与唯一性的泛型推断。”载于：《ACM编程语言论文集》6.ICFP，95:137–95:164。DOI：10.1145/3547626。URL：https://dl.acm.org/doi/10.1145/3547626（访问于 2023 年 10 月 16 日）。Wadler, Philip（1992年4月）。“线性逻辑无可替代。”载于：程序语义数学基础研讨会（MFPS 8）。研讨会论文。英国牛津。Yang, Edward Z. 等（2015年8月）。“使用紧凑范式实现高效通信与收集。”英文。载于：第 20 届 ACM SIGPLAN 国际函数式编程会议论文集。加拿大不列颠哥伦比亚省温哥华：ACM，第 362–374 页。ISBN：978-1-4503-3669-7。DOI：10.1145/2784731.2784735。URL：https://dl.acm.org/doi/10.1145/2784731.2784735（访问于 2022 年 4 月 4 日）。Yorgey, Brent A. 等（2012年1月）。“为 Haskell 引入提升。”英文。载于：第 8 届 ACM SIGPLAN语言设计与实现中的类型研讨会论文集。美国宾夕法尼亚州费城：ACM，第 53–66 页。ISBN：978-1-4503-1120-5。DOI：10.1145/2103786.2103795。URL：https://dl.acm.org/doi/10.1145/2103786.2103795（访问于 2025 年 6 月 3 日）。