arXiv:2512.22684v1 [cs.PL] 27 Dec 2025

用证据编译渐进类型

JOSÉ LUIS ROMERO，智利大学计算机科学系（DCC）PLEIAD 实验室，智利
CRISTÓBAL ISLA，智利大学计算机科学系（DCC）PLEIAD 实验室，智利MATÍAS TORO，智利大学计算机科学系（DCC）PLEIAD 实验室，智利ÉRIC TANTER，智利大学计算机科学系（DCC）PLEIAD 实验室，智利

在结构类型语言中高效支持健全渐进类型颇具挑战。迄今为止，Grift 编译器是该场景下唯一一种接近底层的渐进式类型实现，它利用强制转换执行运行时检查，并进一步通过单调引用扩展，实现对静态类型数据结构的高效访问。在语言设计与语义方面，抽象渐进类型（AGT）方法论已证明颇有成效：它不仅阐明了现有设计，还通过推导多种类型系统和语言特性的渐进化方案推动创新。AGT 的运行时语义以抽象解释为基础，受 Curry-Howard 启发，其核心概念是用于一致判断的证据；这些证据会在归约过程中不断演变，以监测良类型性的合理性。尽管这种基于证据的语义表达力强、适用性广，但尚不清楚它是否能成为实现高效渐进式类型的可行途径。

在这项工作中，我们通过设计、实现并评估一个名为 GrEv 的基于证据的编译器，来探讨这一问题。我们阐释了如何弥合形式语义与 GrEv 编译器实现之间的差距，并确定了新颖的单调语义。我们使用 Grift 基准测试套件对 GrEv 的性能进行了实证评估。结果表明，基于证据的编译器可以媲美基于强制转换的编译器，甚至更快；而且在静态到动态谱系的不同配置下，其性能表现更加稳定。除了丰富渐进类型编译器的实现空间之外，这项工作还为探索文献中通过 AGT 形式推导出的众多高级渐进类型系统的高效实现开辟了一条直接途径。

1 引言

渐进式类型通过支持不精确类型、尽可能多地进行静态检查、乐观地处理不精确性，并将乐观假设推迟到运行时检查，从而调和静态类型与动态类型 [40, 43]。文献探讨了渐进式类型的多种形式化方法和实现方法：或是在现有的动态类型语言之上增设渐进类型系统 [48, 55, 57]——这一趋势在业界日益盛行，TypeScript 等语言便是其中的例子——或是像最初构想的那样，通过允许不精确性来放宽静态类型系统 [40]。Takikawa 等人 [45] 曾警示，渐进式类型在某些配置下可能带来巨大的开销，从而推动了对渐进式类型变体和实现技术的进一步研究 [14, 27, 34, 35, 52, 53, 56, 57]。最值得关注的维度包括：类型系统是结构化的 [19, 40, 41] 还是名义化的 [34, 35],；渐进式类型是粗粒度的（按模块划分） [48] 还是细粒度的（按类型注释划分）；健全性保证是深层、浅层还是可选的 [20], ；语言是函数式的还是面向对象的 [41, 46],；以及渐进式类型是构建在动态类型后端还是静态类型后端之上——在后一种情况下，如果希望最终得到安全的语言，就不能像 TypeScript 那样简单地忽略类型信息。

在这项工作中，我们关注面向具有可变数据结构的高阶语言的结构化细粒度健全渐进式类型，其中静态类型代码无需运行时类型检查即可高效编译。若要研究如何实现

作者联系信息：José Luis Romero，PLEIAD 实验室，智利大学计算机科学系（DCC），智利圣地亚哥，joromero@dcc.uchile.cl；Cristóbal Isla，PLEIAD 实验室，智利大学计算机科学系（DCC），智利圣地亚哥，cisla@dcc.uchile.cl；Matías Toro，PLEIAD 实验室，智利大学计算机科学系（DCC），智利圣地亚哥，mtoro@dcc.uchile.cl；Éric Tanter，PLEIAD 实验室，智利大学计算机科学系（DCC），智利圣地亚哥，etanter@dcc.uchile.cl。

例如，OCaml 渐进式类型版本的高效实现，那么这一设置就十分重要。尽管在生产级编译器中实现高效集成仍是一个开放问题，但近年来已有多项进展。最值得注意的是，Grift [27] 已成为研究贴近底层的渐进式类型实现及其优化的基准。Grift 展示了使用强制转换 [22] 相较于基于类型的强制转换的优势。随后，Almahallawi [4] 扩展了 Grift，以评估单调引用的性能 [44], 。这是一种确保堆分配结构的运行时类型信息随时间推移只能变得更加精确的机制，从而能够进一步优化静态类型数据的操作。Tsuda 等人 [53] 探索了在 Grift 中采用强制转换传递风格来恢复尾调用优化；最近，Campora 等人[14] 通过在 Grift 中实现一种名为判别式类型的优化技术，对其进行了研究。

与实现技术方面的研究并行，渐进式类型文献也深入探讨了面向不同语言特性（可变引用 [40, 44], 对象 [2, 41, 46], 多态 [1, 24, 25, 50],受限延续 [33],效应处理器 [36], 等）的渐进式类型语言设计与语义，以及面向高级类型系统（效应类型系统 [7], 类型状态 [58],安全类型 [47], 所有权 [39], 细化类型 [30],等）的相关研究。在这一领域，抽象渐进类型（AGT）方法论 [19] 已证明能够以坚实的原则为渐进式语言设计奠定基础：AGT 基于抽象解释 [17], ，尤其是将渐进类型视为静态类型集合的抽象，通过伽罗瓦连接将静态类型关系和运算符提升为与之对应的一致关系和运算符，从而指导现有静态类型系统的渐进化。AGT 还从 Curry-Howard 的角度说明了运行时检查：随着程序通过一致传递性 运算符不断约简，类型推导中的一致判断之证据 也随之演变，运行时检查正是这一过程的结果；从这个意义上说，它是在运行时重演该语言的类型安全证明。

AGT 既用于阐明现有设计，也推动了渐进式类型领域的创新。迄今为止，人们已研究了适用于多种特性和类型系统的证据概念与一致传递性 [5, 6, 8, 29, 30, 49–52, 59]。这种方法虽然严谨且富有表现力，但其设计目标并非效率，而是源自元理论。AGT 语言的基于证据的语义的形式化描述，已从对类型推导或带注释的内禀类型项进行归约，逐渐转向一种更简单、也更接近实现的描述：其中所有值都附有证据，一致性判断仅限于类型标注，而运行时语义则系统地应用一致传递性来合并类型标注 [50, 52]。这一更简单的描述也被近期的一些工作采用 [5, 59], ，它提出了一种相当直接的实现策略，尽管这种策略可能较为简单粗糙且效率低下。鉴于 AGT 概念框架在表达能力上的优势，我们不禁要问：在编译器中直接实现基于证据的语义，是否真的会极其低效？还是说，它能够与强制转换相媲美？在这项工作中，我们通过构建一款新的、基于证据的渐进类型语言编译器来回答这个问题。该语言支持一等函数和可变数据结构。具体而言，本文作出以下贡献：

- 我们解释如何弥合形式主义与基于证据的语义底层实现之间的差距，特别是证据值的底层表示，以及如何在编译器中实现基于证据的归约规则。我们还解释如何在基于证据的框架中实现 Grift 的一些基本渐进式类型优化。
- 我们介绍了文献中与受保护 语义和单调 语义相对应的不同设计 [44], ，并指出一种新的单调语义——单调值——在这种语义中，所有堆分配值，尤其是闭包，均以单调方式处理。
- 我们实现了一种新的提前编译器，称为 GrEv，目标是 LLVMIR，并且与 Grift 使用相同的源语言。GrEv 使用 OCaml 实现，部分运行时

组件使用 C 编写。GrEv 支持三种语义：受保护值、单调数据结构（包含受保护闭包）以及单调值。

- 我们通过实证评估了 GrEv 与 Grift 在 Grift 基准测试套件中，针对所有受保护和单调语义变体的性能 [4, 27]。结果表明，基于证据的编译器能够达到与基于强制转换的编译器相当、甚至更优的性能，并且在静态到动态的谱系中，其性能在不同配置下似乎更加稳定。对两种编译器不同语义模式的比较证实了单调方法在渐进式类型中的价值。我们还报告了浮点数优化的情况：该优化已在 Grift 中实现，在 GrEv 中则是可选的，其影响显然颇具争议。

最后，尽管本文只关注如何将简单类型系统渐进化，但像 GrEv 这样的基于证据的编译器，为探索 AGT 文献 [5, 6, 8, 29, 30, 49–52, 59] 中所研究的高级渐进类型系统的高效实现打开了大门。

§2 介绍渐进式类型和抽象渐进类型（AGT）的背景，§3 介绍基于证据的编译器 GrEv 的设计与实现及其语义变体，§4 报告 GrEv 和 Grift 的性能评估。最后，我们讨论相关工作（§5），并作出结论（§6）。GrEv 的实现以及性能评估中使用的实验材料可在线获取：https://pleiad.cl/grev/。

2 抽象渐进类型背景

我们先简要介绍渐进式类型（§2.1），再介绍抽象渐进类型方法论的主要要素（§2.2），具体来说是基于证据的语义（§2.3）。

2.1 渐进式类型

静态类型和动态类型各有互补的优点与缺点。静态类型可以及早检测错误，但可能会保守地拒绝有效程序；动态类型则提供了灵活性，但代价是运行时检查和潜在错误。渐进式类型将这两种范式结合在同一种语言中，使程序员能够在局部选择静态检查或动态检查，并沿着这一谱系平滑地演进程序 [41]。

渐进式类型通过渐进类型之间一种称为（不）精确性 ($G_1 \sqsubseteq G_2$)的关键关系，将这种灵活性形式化。直观而言，该关系刻画了类型所提供的静态信息量。精度最低的类型通常称为未知类型 ?，而静态类型则是精度最高的类型。结构化渐进类型允许未知类型出现在类型结构的任意层级，例如 ? → 布尔类型，表示函数一定返回布尔值，但接受任意类型的值作为参数。渐进类型系统会乐观地处理不精确性，采用能够容忍不精确性的类型关系。例如，类型一致性 (~)放宽了标准相等关系，只要求两个类型在已知部分一致，例如 ? → 布尔类型 ~整数类型 → ? 。重要的是，一致关系通常不具备传递性：整数类型 ~ ? 且 ? ~ 布尔类型，但整数类型 ≁ 布尔类型；否则，任何程序都不会被静态判定为类型错误。对于更高级的类型系统，例如包含子类型的系统，需要设计相应的乐观对应关系，如一致子类型 [41]。

Cons请看下面这个例子，它说明了渐进式类型中的三种不同情形：

```
1 令 f : ?→整数类型 = 函数g → g(1) 在                    (* add1: 整数类型 → 整数类型 *)
2 f(add1)              (* 类型检查通过，运行时没有错误 *)
3 not(f(add1))         (* 类型错误 *)
4 f(真)                (* 类型检查通过，但在运行时失败 *)
```

首先，f 的函数体能够通过类型检查，因为g 的类型是?，这与限制最少的函数类型 ? → ? 一致，而? 自身也与声明的返回类型 整数类型一致。在第 2 行，应用 f(add1) 能够通过类型检查，因为 整数类型 → 整数类型 ~ ?，并成功求值得到 2。在第 3 行，not的应用要求一个布尔值，因此会被静态拒绝，因为f 已被静态确定会返回一个整数类型。最后，第 4 行的应用类型正确，因为 布尔类型 ~ ?，但在将 true作为函数应用之前，它应该在运行时失败。

确保第 4 行在运行时失败，是在实践中实现渐进式类型时面临挑战的核心。如果是在安全的动态语言之上加装类型系统来实现渐进式类型（例如 TypeScript 或 Typed Racket），那么可以让底层运行时系统负责确保安全；这种方法称为可选类型 [11]。然而，如果是在 OCaml 这类静态类型语言中加入渐进类型，那么渐进式类型的实现就必须负责确保安全，在乐观地接受操作（如上面的应用 g(1)）之前执行运行时检查。

渐进式类型的开创性论述 [40] 采用了后一种方案，通过 elaboration 到一种带有显式运行时标记和检查的内部语言，为渐进式语言设计语义；这种内部语言称为强制转换演算。在静态已知信息与未知信息的边界处插入类型转换，以确保任何对静态假设的违背都会被检测并报告为运行时错误，而非导致段错误。考虑到第 4 行中的用法，上述程序的一种可能 elaboration 是（e : G ⇒ G' 表示将e 从G 转换为G'）：

```
令 f : ?→整数类型 = 函数 g → (g : ? ⇒ ?→?)(1 : 整数类型 ⇒ ?) : ? ⇒
整数类型 于f(真 : 布尔类型 ⇒?)
```

请注意，所有利用一致性判定程序为渐进式良类型的位置都会插入类型转换。向未知类型的向上转换可以理解为“标记”操作，而从未知类型向更精确类型的向下转换则构成运行时检查，这些检查可以读取先前标记的信息。在这里的归约过程中，将真 : 布尔类型 ⇒ ?（一个带标记的布尔值）代入函数体后，归约表达式假 : 布尔类型 ⇒ ? : ? ⇒ ?→? 会触发运行时错误，因为布尔值无法转换为函数类型。

如引言中所述，这里阐述的渐进式类型的核心原则——不精确性、一致性和运行时检查——相当直接，但引入运行时检查可能会对性能产生巨大影响。此外，将这些原则扩展到更复杂的语言以及更复杂的类型关系（例如子类型、多态、效应、依赖关系等），要正确处理可能相当微妙。无论在语义层面还是实现层面，渐进式类型都带来了严峻挑战。

2.2 抽象渐进类型

抽象渐进类型（AGT）方法论提供了一个有原则且通用的框架，用于基于抽象解释 [17],从静态类型语言出发，系统地设计渐进类型语言。其核心思想是将渐进类型视为静态类型集合的抽象，并通过伽罗瓦连接对其进行形式化定义（包括具体化 γ 以及可靠且最优的抽象 α）。例如，将未知类型视为任意类型的抽象（$\gamma(?)$ = 类型），并将这一思想推广到所有类型构造器，即可得到 GTLC 的标准解释 [40, 43],，例如 $\gamma(? \to 整数类型) = \{T \to 整数类型 \mid T \in 类型\}$，以及$\alpha(\{整数类型 \to 布尔类型, 整数类型 \to 字符串\}) = 整数类型 \to ?$ 。渐进类型之间的精度概念 [43]也由此自然得出，即 $G_1 \sqsubseteq G_2 \triangleq \gamma(G_1) \subseteq \gamma(G_2)$。

有了这个在语法上捕捉渐进类型含义的伽罗瓦连接，AGT 通过两个步骤推动静态类型系统的渐进化。首先，对于静态语义，只需通过伽罗瓦连接提升所有类型关系（例如，相等关系、子类型、

包含关系等）和部分类型运算符（例如，等同、子类型连接、并集等）。这种提升采用存在量词方式，捕捉了渐进式类型关注良类型性是否可能这一事实。例如，一致性可以自然地理解为相等关系的存在量词提升：$G_1 \sim G_2 \triangleq \exists T_1 \in \gamma(G_1)\ T_2 \in \gamma(G_2)\ T_1 = T_2$，这意味着两个类型至少都表示同一个静态类型。

其次，对于动态语义，AGT 利用（类型安全论证的）证明归约与项归约之间的 Curry-Howard 对应，来处理可能出现的运行时错误：我们不再只关注合理性，而是追踪这种合理性的局部证据，归约则要求将这些局部依据组合起来，尝试为传递步骤提供依据（这是一种称为一致传递性的偏函数，也可以直接用伽罗瓦连接来定义 [19]）；不兼容性则表示有依据的运行时类型错误。鉴于其在本文中的核心作用，我们将在下文（§2.3）进一步介绍基于证据的语义。AGT 在阐释三元组等既有概念 [42], 以及在具有挑战性的场景中设计新的运行时语义运算符方面，同样已证明行之有效。

总的来说，AGT 已成功应用于广泛的特性和类型系统，包括记录与子类型 [6, 19], 类型与效应 [7, 8], 细化类型 [30, 54], 集合论类型与联合类型 [15, 51], 信息流类型化 [49], 参数多态 [28, 29,50], 灵活数据类型 [32], 概率λ演算 [59],以及敏感性类型 [5]。

2.3 基于证据的语义

如上文所述，AGT 不通过翻译为强制转换演算 [40], 来给出渐进式语言的动态语义，而是为渐进式程序提供一种定义在渐进式类型推导 [19]之上的直接动态语义。其核心思想是对由证据增强的一致性判断的渐进式类型推导 [23] 应用证明归约；根据 Curry-Howard 对应，这会为渐进项诱导出一种规范的归约概念。

为了直观了解其工作方式，考虑项 $t(v) := (\lambda x : ?.x + 1)\ v$。该项在渐进式类型系统下类型正确，但在运行时可能会失败，具体取决于 v 是什么；其类型推导 D 如下：

$$\dfrac{\dfrac{\dfrac{\dfrac{x:? \in x:?}{x:? \vdash x:?} \quad \dfrac{}{x:? \vdash 1: \text{整数类型}} \quad \varepsilon_1 \vdash ? \sim \text{整数类型}}{x:? \vdash x+1 : \text{整数类型}}}{\vdash (\lambda x:?.x+1) : ? \to \text{整数类型}} \quad \dfrac{}{\vdash v : G_v} \quad \varepsilon_2 \vdash G_v \sim ?}{\vdash (\lambda x:?.x+1)\ v : \text{整数类型}}$$

在该推导中，我们将使此程序逐步满足类型要求的两个一致性判断所依据的证据 暂且视为不透明。从逻辑上说，使用蕴含的证明可以通过归约来规范化：将假设的证明代入蕴含证明中对变量的使用处——这在 Curry-Howard 对应下等价于 β-归约。在静态环境下，这一步的类型保持论证（隐式地）利用了等式的传递性；但此处使用的是一致性，而一致性并不普遍满足传递性。因此，归约步骤需要通过一个传递性步骤，在 D 的一致判断之间建立一个新的 一致性判断；这是否合法取决于两个判断的局部依据，即 ε_1 和 ε_2：如果它们可以通过一致传递性组合，记作 $\varepsilon_2 \circ \varepsilon_1$，那么归约就可以继续进行，并将产生的证据用作结果推导树中的依据：

$$\mathcal{D} \longmapsto \dfrac{\dfrac{}{\vdash v : G_v} \quad \dfrac{}{\vdash 1 : \mathsf{Int}} \quad \varepsilon_2 \circ \varepsilon_1 \vdash G_v \sim \mathsf{Int}}{v + 1 : \mathsf{Int}} \quad \text{if } \varepsilon_2 \circ \varepsilon_1 \text{ defined}$$

如果 $v := 10, G_v :=$ 整数类型，一切都没问题。但如果 $v :=$ false， $G_v :=$ 布尔类型，那么 $\varepsilon_2 \circ \varepsilon_1$ 未定义，因此归约会产生一个错误，也就是说，D $\longmapsto$ **错误**，如果 $\varepsilon_2 \circ \varepsilon_1$ 未定义。

这种对渐进式类型的运行时语义所作的有原则的阐述，可以扩展到更丰富的语言和更复杂的类型系统，许多以这种方式构建的形式系统便是明证。给定一种合适的证据表示，AGT 就能给出一致传递性的定义。随后，语言设计者可以提出算法定义，并证明它们等价于通过抽象解释框架推导出的“真实基准”。

在本文中，我们关注简单类型，其中唯一的一致性判断就是一致性，即类型相等关系的存在量词提升（相等关系具有对称性）。在这个极其简单的设定中，证据可以表示为单个渐进类型，它概括了一致性判断所指称的共同类型。例如，$\langle \text{整数类型} \to \text{布尔类型} \rangle \vdash ? \to$ 布尔类型 $\sim$ 整数类型 $\to ?$。类似地，一致传递性就是两个类型之间的精度交：$\langle G_1 \rangle \circ \langle G_2 \rangle = \langle G_1 \sqcap G_2 \rangle$。

简化表示。为了形式化渐进式程序的运行时语义，同时避免在一般推导树上书写归约规则，Garcia 等人 采用 内蕴项 [16],——一种与类型推导同构的扁平表示。尽管这种表示足以满足要求，但由此得到的描述相当特殊，使用起来也很繁琐。此外，它确实隐含地涉及从源项到其内蕴表示的一种（表述上的）转换。

近期关于 AGT [5, 28, 50, 59] 的研究通过避免使用内蕴项，简化了论述。取而代之的是，先通过一种简单的类型导向展开，生成在每处使用一致性时都显式带有类型标注的项——这很符合基于强制转换的子类型语义的思路 [37]。所有值都带有证据标注，且只有标注表达式携带证据。因此，形式化后的语言十分熟悉，其类型规则和归约规则也大大简化。具体来说，判断 $\Gamma \vdash t : G$ 的类型规则是标准的，只有标注规则例外：它使用一致性判断并携带其证据：

$$\text{(eGapp)}\ \frac{\Gamma \vdash t_1 : G_{11} \to G_{12} \qquad \Gamma \vdash t_2 : G_{11}}{\Gamma \vdash t_1\ t_2 : G_{12}} \qquad \text{(eG::)}\ \frac{\Gamma \vdash t : G_1 \qquad \varepsilon \vdash G_1 \sim G_2}{\Gamma \vdash \varepsilon t :: G_2 : G_2}$$

对于求值而言，标注的目标类型在计算上无关紧要，因此值的形式为 $v ::= \varepsilon u$，函数应用和标注的归约规则为：

$$\begin{array}{rll} \varepsilon_1(\lambda x.t)\ \varepsilon_2 u \longmapsto cod(\varepsilon_1) t[(\varepsilon_2 \circ dom(\varepsilon_1))u/x] & & \text{if defined, otherwise } \textbf{error} \\ \varepsilon_2(\varepsilon_1 u) \longmapsto (\varepsilon_1 \circ \varepsilon_2)u & & \text{if defined, otherwise } \textbf{error} \end{array}$$

举例来说，考虑前面的项 t(false)、它的展开式 ( ) 以及归约过程：

$$\begin{array}{lr} t \rightsquigarrow \langle ? \to \mathsf{Int} \rangle (\lambda x : ?.\langle \mathsf{Int} \rangle x + \langle \mathsf{Int} \rangle 1)\ \langle \mathsf{Bool} \rangle \mathsf{false} & \\ \longmapsto \langle \mathsf{Int} \rangle (\langle \mathsf{Int} \rangle \langle \mathsf{Bool} \rangle \mathsf{false} + \langle \mathsf{Int} \rangle 1) & (dom(\langle ? \to \mathsf{Int} \rangle) = \langle ? \rangle \text{ and } \langle \mathsf{Bool} \rangle \circ \langle ? \rangle = \langle \mathsf{Bool} \rangle) \\ \longmapsto \textbf{error} & (\langle \mathsf{Bool} \rangle \circ \langle \mathsf{Int} \rangle = \langle \mathsf{Bool} \sqcap \mathsf{Int} \rangle \text{ is undefined}) \end{array}$$

这种表述方式为实现带证据的渐进式类型提供了一种相当直接的策略，至少适用于概念验证用的解释器。正如本文后续部分将探讨的那样，设计编译器还涉及不少其他考量。

最后值得强调的是，尽管本文具体研究的是带有一致性的简单渐进类型这一特定情形（因此上文采用了简化的形式化表述），但 AGT 的通用框架和基于证据的语义定义适用于各种类型系统。构建一种基于证据的编译器，其中证据表示和一致传递性的实现由编译流水线的其余部分抽象地处理，为研究各种渐进类型系统的底层实现提供了一个颇具吸引力的通用基础。本文首先在简单类型环境下探讨一个问题：使用证据编译渐进式程序，能否与使用强制转换的方式相媲美。

$e ::=$ 函数 $(x: G) \to e \mid e\ e \ldots \mid x$ (函数和变量)$\mid n \mid f \mid e \odot e$ | 运算符 $e \ldots$ (整数、浮点数和运算符)$\mid b$ | 如果 e 那么 e 否则 e (布尔值和条件表达式)$\mid$ 对于 $x = e$ 到 e 执行 e 结束 (循环)$\mid$ let $x: G = e$ 在 $e \mid$ let rec $f: G = e$ 在 e (let 绑定)$\mid$ 引用 $e \mid !e \mid e := e$ (引用、单元类型)$\mid$ 向量 $e\ e \mid$ vset $e\ e\ e \mid$ vget $e\ e$ (向量)$\mid (e) \mid e.(e)$ (元组)$\mid$ 类型 $N = [\mid C \text{ of } G]^+ \mid C\ e \ldots$ (变体定义和构造)$\mid$ 匹配 e 与 $[\mid C\ x \ldots \to e]^+$ (变体消去)$\mid e :: G \mid e; e$ (类型标注和序列)$G ::=$ 整数类型 | 布尔类型 | 浮点数 | 引用$[G]$ | 向量$[G]$ (类型)$\mid$ 单元类型 $\mid G * G \ldots \mid (G) \to G \mid N \mid ?$

图 1. GTLC+源代码语法

3 GrEv 编译器

我们通过实现一个名为 GrEv 的编译器，探究如何高效实现基于证据的渐进语言。与 Grift 一样，GrEv 专注于结构化、细粒度且健全的渐进式类型，并以低级、不安全的后端为目标，这意味着渐进式类型机制负责执行类型相关安全检查。

我们首先简要介绍源语言 GTLC+，并回顾此类语言编译的基本要素，同时考察静态类型和动态类型两种情形（§3.1）。随后，我们重点介绍基于证据的渐进式类型的编译：证据值（§3.2）、elaboration 与求值（§3.3）。GrEv 的介绍最后简要说明渐进类型优化（§3.4），并与 Grift 进行比较（§3.5）。

3.1 源语言与编译的基本要素

GTLC+ [27] 是一种核心语言，支持一等函数、基本类型（包括整数类型、布尔值和浮点数），以及不同种类的数据结构：引用（即 堆单元）、向量、不可变元组和变体（即 代数数据类型）。表达式包括 let 绑定（可以是递归的）、基本运算符的应用（如加法、打印等）、函数应用、条件表达式、循环、数据结构访问以及类型标注。语法包含类型注解，但可以省略。类型是结构化的，具有用于函数和数据结构的类型构造器以及基础类型，还包括未知类型 ?。GrEv 中的递归类型是同构递归的，且仅用于定义变体，例如 **类型** 流 = SCons **由整数类型** * (**整数类型** → 流)。

除了 GrEv（它将缺失的类型注解视为 **?** 并支持健全渐进类型）之外，我们还实现了 StaticGrEv，这是 GTLC+ 的标准编译器，要求所有类型注解都必须存在且完全精确（不能使用未知类型），并遵循静态类型语言编译的基本原则；此外还实现了完全动态类型版本 DynGrEv，它会忽略所有类型注解，并按标准的动态类型语言编译器的方式进行编译。在静态类型编译器中，基本类型的值是即时值（也就是说，它们可以存放在栈和寄存器中），而数据结构则在堆上分配，并通过指针访问。一等函数以扁平闭包的形式实现，即 也就是说，在堆上分配的结构，其中包含指向函数体的代码指针，以及词法捕获变量的值（环境）。该编译器利用静态类型信息，避免任何运行时类型

检查。相比之下，在动态类型编译器中，为确保安全，所有对值的访问都需要进行标签检查，因此所有值——无论是即时值还是堆分配值——都会额外标记类型标签。这意味着在 64 位架构上，整数被限制为 63 位，其中一位用作标签。由于浮点数无法缩减至 63 位，因此必须在堆上分配以添加标签，从而导致浮点运算的性能低于可以将浮点数视为即时值的静态类型编译器。

StaticGrEv 的编译流水线是标准的：词法分析、语法分析、类型检查、 α重命名、ANF 转换、闭包转换（含一些优化 [26]），以及降低到 LLVM IR。

3.2 使用证据值进行编译

为了实现一个基于证据的 GTLC+ 编译器，我们需要将 §2.3 中的基于证据的形式体系映射为一种低级实现策略。

主要要点是，所有值都是 证据值， εu并且语言的消去形式（函数应用、类型标注、投影等）需要使用 一致传递性， $\varepsilon_1 \circ \varepsilon_2$组合涉及值的证据，以便最终生成一个证据值作为结果；如果检测到运行时类型不一致，则引发错误。

表示值。 在形式体系中，值的描述方式是统一的，但在实现中，有些值是立即值（例如 整数）——这意味着它们可以存放在栈或寄存器中——而另一些值则是堆分配的（例如元组和向量）。在形式化描述中，通常只有在需要处理可变性时才会引入堆；但在编译器中，由于寄存器只能容纳 64 位字，任何无法装入一个字的值都必须使用堆。[1]

此外，形式语义通常通过替换元函数 $t[v/x]$ 来实现具有词法作用域的一等函数 $\lambda x.t$：在函数体 t中，将形式参数 x的每个自由出现替换为实际参数 v。相比之下，任何效率尚可的实现都不会在运行时执行这种代码转换，而是使用能够捕获其词法环境的闭包。闭包无法实现为即时值，因为它至少需要一个标签（代码指针），还可能需要若干值来表示其捕获的词法环境。

证据值。由于 GTLC+ 的证据可以是任意大小的类型，因此通常无法将证据表示为位标签；所以，一种通用且统一的方案是将所有证据值放在堆上，并连同它们的两个组成部分 ε和 u一起存储。此规则的主要例外是整数和布尔值。由于与它们关联的证据必然就是它们的（原子）类型，因此这些证据值可以实现为即时值： 也就是说，如果 $v = \varepsilon u$ 且 u 是整数，那么 $\varepsilon = \langle$**整数类型**$\rangle$，无论 v : 整数类型 还是 v : ?。因此，可以像通常那样使用位标签来实现整数和布尔值的证据。如 §3.1 中所述，浮点数更具挑战性，因为它们无法同时带标签并作为即时值；因此，初步而言，在渐进式语言中，浮点数需要装箱（我们将在 §3.4 讨论一种相关的优化）。

数据结构。 与 Grift 一样，GrEv 中的所有数据结构都在堆上分配。我们来关注引用单元；其他数据结构的处理方式也类似。从概念上说，我们可以将证据与单元数据一同放在堆上，也可以将证据与指向该单元的指针关联起来。不过，将指针和证据打包无法放入一个字中，因此后一种做法实际上意味着使用一个（堆分配的）代理。这些表示方式的差异会对语义产生影响：前一种对应于单调引用，这是 Siek 等人 [44], 最先提出的；后一种则对应于传统的守卫引用 [40]。

[1] 除非该值是局部的，此时它可以分配在栈上；例如，Grift 和 GrEv 都是简单的编译器，不会尝试将结构分配在栈上，也不会通过逃逸分析来实现这一点。

值 在更精确的类型标注下:	**单调** 原地更新证据	**受保护** 使用新事件进行浅拷贝
ref	ev | val	ev | 值_指针
向量	ev | len | 值...	ev | len | 值_指针
元组	ev | 值...	ev | 值_ptr
变体	ev | ctr_id | 值...	ev | ctr_id | 值...
闭包	ev | 函数_指针 | 值.....	ev | 函数_指针 | 值_指针

表 1. 单调与受保护语义中装箱的证据值在内存中的表示 s.

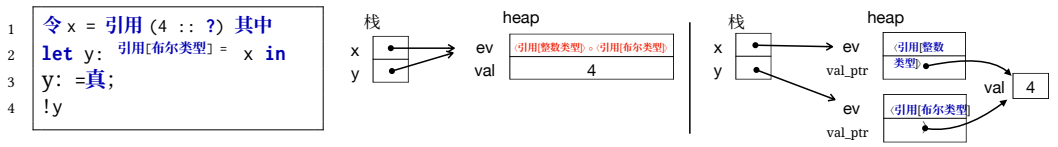

图 2. 单调引用与受保护引用

对于单调引用，堆中某个值的证据至少与指向该值的任何引用一样精确。当引入一个类型更精确的别名时，堆上的证据也会相应更新；如果检测到不一致，则会引发错误。这种方法有两项与性能相关的优势：*(a)*由于运行时类型信息（证据）由所有别名唯一共享，因此无需创建代理；*(b)* 对静态类型引用的访问可以省去运行时检查。为说明单调引用和守卫引用之间的区别，请看图 2所示的程序。当绑定到 x 的引用被创建时，其值的静态类型未知，其在堆中关联的证据为⟨**引用[整数类型]**⟩。随后，当以类型 y 创建**引用[布尔类型]**, 时，该信息会与堆单元当前的证据合并，从而在此处导致第 2 行出现运行时错误。

相比之下，受保护引用对程序员来说更加灵活，但每次访问都必须进行动态检查，即使是从静态类型代码中访问也是如此。当 x 和 y 被引入时，会分配两个代理，并分别附带证据 ⟨**引用[整数类型]**⟩ 和 ⟨**引用[布尔类型]**⟩，二者都指向同一个堆单元。因此，第 3 行的赋值和第 4 行的解引用都能成功，最终结果为 真。图 2右侧的示意图展现了这种对比：左侧展示单调情形，类型更新会直接传播到堆中；右侧展示受保护情形，代理会中介对底层内存单元的所有访问。

重要的是，单调引用满足渐进保证 [43], , 因此尽管比守卫引用更加严格，仍能实现静态与动态类型检查之间的细粒度迁移。此外，Siek 等人 [44] 解释了如何通过一种语法约束，在单调引用之上恢复灵活性：始终以未知类型为值赋值，并将更精确的类型标注延后到引用的客户端。

Grift 将单调引用的概念推广到了向量和元组 [4]。Grift 通过编译器标志同时支持受保护语义和单调语义。类似地，GrEv 对数据结构也支持这两种语义（表 1）。受保护引用是一种代理，由附加到引用的证据和指向实际数据的指针组成（值_ptr）。进行更精确的类型标注时，会创建一个带有新证据的新代理，并复制数据指针。相比之下，单调引用保存证据和实际数据；进行更精确的类型标注时，证据会

原位更新。因此，GrEvenjoy 中的单调引用具有上述优势 *(a)*，即无需为每个别名分配代理。不过，Siek 等人 [44]指出，要获得上述优势 *(b)*，必须递归地将更新深入传播到堆单元，以妥善处理嵌套结构。这个过程本身可能代价高昂，而且需要特别小心以避免循环。与单调引用的深层更新 [44], 不同，GrEv 采用浅层处理方式，也就是说，更新后的证据信息不会递归传播到嵌套结构中。这意味着 GrEv 无法获得单调引用的优势 *(b)*。（我们在 §4 的性能评估中比较了所有这些不同的语义。）

向量类似于引用，但有以下区别：由于向量类型不携带长度信息，向量会存储自身的长度，并保存多个值（的指针）。元组不必携带长度，因为相关证据中已有该信息。对于变体，GrEv 为每个变体类型分配一个唯一标识符，并为每个构造器分配一个唯一标识符；类型标识符存储在与每个值关联的证据（事件）中，而构造器标识符（ctr_id）则存储在值本身中（表 1）。由于变体类型本质上是新的原子类型，其证据完全精确，因此无论通过何种类型访问它们，证据都不会发生变化。

闭包。 我们观察到，关于受保护数据结构与单调数据结构的讨论同样适用于闭包：一种做法是将证据附加到闭包上，并让其单调演化；另一种做法是通过代理，从而允许同一闭包的不同别名具有不同的类型约束。据我们所知，这一观察是新颖的：Siek 等人 [44] 并未探讨一等函数的单调语义，因为他们通过替换而非闭包来对其建模；而 Grift 只实现了受保护闭包。

简要来说，对于单调闭包，只要闭包本身尚未被归属为与其使用方式不兼容的精确类型，无类型恒等函数就可以接受任意类型的参数。下方左侧的 `f` 同时用于整数类型和布尔类型参数：

```
令 f : ?→?  = 函数 x  → x于
print_int(f 10)
打印_布尔类型(f 真) (* 正确 *)
```

```
令 f : ?→?  = 函数 x  →x 在
let g : 整数类型→整数类型 = f in
打印_整数类型(g 10); 打印_布尔类型(f 真) (* 失败 *)
```

但在右侧，被 `f` 和 `g`共同别名的闭包已单调地演变为**整数类型→整数类型** 函数，这是由于对`g`的类型注解，因此在运行时将true 应用于**f** 会失败。

为了评估单调闭包的性能影响，GrEv 支持两种闭包语义。表 1 展示了闭包的内存表示：证据，其中包含元数、代码标签（fun_ptr），该标签指向函数主体经闭包转换后生成的编译代码，以及环境（单调模式下直接存储，受保护模式下则通过指针存储）。

语义模式。总之，Grift 支持两种语义模式：受保护模式（Grift/G），使用代理数据结构和代理闭包；以及单调结构模式（Grift/MS），使用非代理数据结构和代理闭包。作为比较，GrEv 支持三种模式：GrEv/G，使用代理数据结构和闭包；以及两种新模式：GrEv/MV（单调值），使用非代理数据结构和闭包；GrEv/MC（单调闭包），保留代理数据结构，但使用非代理闭包。我们纳入 GrEv/MC，专门用于隔离并测量单调闭包的性能影响（§4）。

3.3 基于证据的编译：elaboration 与归约

基于证据的形式语义与 AGT 由一个插入类型标注的精化阶段，以及随后用于归约的小步语义组成。在推导编译策略时，运行时语义必须改为通过动态精化阶段来实现。GrEv

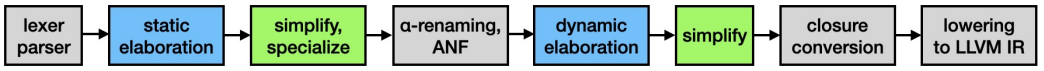

图 3。GrEv 编译流水线。标准阶段以灰色表示，引入证据标注的 elaboration 阶段以蓝色表示，渐进式类型优化以绿色表示。

编译流水线如图3所示：下面我们简要讨论两个精化阶段（蓝色框），并在 §3.4中讨论后续的优化阶段（绿色框）。

静态精化。 如 §2.3 中所述，凡使用一致性的地方都会插入类型标注，但源代码中已有的类型标注除外。此静态精化阶段（图 3）会影响值、函数应用、数据结构访问和原语操作，如下所示：

```
令 f = 函数 (x:?) → x* 2
于 打印_整数类型 (f假)
```

elab ⇝

```
令 f = ⟨?→整数类型⟩(函数(x:?) → ⟨整数类型⟩x* ⟨整数类型⟩(⟨整数类型⟩
    2))
  in
打印_整数类型⟨整数类型⟩(⟨?→整数类型⟩f ⟨布尔类型⟩(⟨布尔类型⟩false))
```

除了对数值、布尔值和函数值进行类型标注外，乘法的每个操作数也都进行了类型标注。同样，函数及其参数分别被标注为 ? → **整数类型** 和?，从而得到上面所示的证据。原语操作 打印_整数类型 也以同样的方式处理：调用之前，其参数必须标注为 **整数类型**。其中许多类型标注都是冗余的，可以删除（§3.4）。

动态精化。 基于证据的语义（§2.3）的归约规则会引入动态 类型标注，即在运行时确定证据的类型标注。请看函数应用的归约规则：

$$\varepsilon_1(\lambda x.t)\ \varepsilon_2 u \longmapsto cod(\varepsilon_1)(t[(\varepsilon_2 \circ dom(\varepsilon_1))u/x]) \qquad \text{if defined, otherwise } \textbf{error}$$

该规则规定，参数 $\varepsilon_2 u$ 应使用函数的定义域证据 进行类型标注，即$dom(\varepsilon_1)$；这会计算两份证据之间的一致传递性，若两者不一致则产生错误，否则会生成新的证据，并在替换进函数体时用它来修饰u。同样，结果函数体也使用函数的陪域证据 进行类型标注，即$cod(\varepsilon_1)$。

GrEv 在动态细化阶段实现这一语义（图 3），方法是在调用点将参数标注为定义域证据，并在调用函数之前判断一致传递性是否成立。陪域证据的标注则插入函数体本身，在从函数调用返回之前检查一致传递性。经过一些转换后的前一个程序（图 3）被 elaboration 如下：

```
令 f = (函数 (x:?) →令 x_标注 = 整数类型x，令 res =
x_标注* 整数类型2，然后对 res 应用标注，使用 cod(self)
作为证据)中，令 arg_0 = 对 布尔类型true 应用标注，使用
dom(0, f)作为证据，然后打印_整数类型 整数类型(f arg_0)
```

该 **标注** 形式在降级阶段处理：它将第一个参数关联的证据与作为第二个参数传入的证据组合起来。如果一致传递性未定义，则引发错误；否则返回新的证据值。这里我们使用 `cod(self)` 和`dom(0, f)` 从闭包证据中提取余域证据和（第一个）定义域证据（`self`、`cod`和`dom` 也在降级阶段处理）。需要强调的是，这些信息

只能在运行时提取。其他消去形式（例如对数据结构的访问）也以类似方式处理。

3.4 渐进类型优化的基础

GrEv 支持几种基本的渐进类型优化，这些优化也在 Grift 中实现，我们将在此简要介绍并举例说明。

展开简化。 展开的形式化定义会统一插入带有初始证据的类型标注，因此可能产生一串类型标注，也可能产生不必要的类型标注。GrEv 会将任意一串类型标注简化为一个带有组合后证据的类型标注。它还会在源代码的证据已经至少与该类型标注本身的证据一样精确时，消除该类型标注；这是因为证据组合具有单调性。例如（为清晰起见，包含类型标注）： `false::?` elab ⟨布尔类型⟩(⟨布尔类型⟩`false::`布尔类型)`::?` simpl ⟨布尔类型⟩`false::?`。接着，运行示例如下简化：

let f = fun (x:?) → x * 2 in print_int (f false) 编译 ... simpl let f = ?→int(fun (x:?) → intx * int2) in print_int (f boolfalse)↝ ↝

除了值上的证据外，唯一相关的证据是 x 旁边的证据，因为它的类型是 `?`。

专用类型标注。对未知类型的项进行任何消去形式（应用、投影等）操作，都需要将该项标注为相应的芽 [31], 即允许该消去形式的最不精确类型（例如，`?→?` 用于一元函数，`vec[?]` 用于向量）。例如：

```
fun (x:?, y:?) → x y  ⇝ (elab & simpl)  fun (x:?, y:?) → ⟨?→?⟩x y
                      ⇝ (specialize)    fun (x:?, y:?) → (checkfun x) y
```

GrEv 将类型标注专门化为芽 `?→?`，生成直接检查顶层类型构造器（上文写作 `checkfun`）的代码。该构造器可在运行时从绑定到 x 的证据值中获取，无需更新证据或创建代理。这项优化对完全无类型代码尤其有效，因为它能使最终执行的顶层标签检查与安全的动态语言所需的检查相同，而不会执行更多检查。具体而言，在守卫语义中，它避免创建不必要的（代价高昂的）代理。

不必要的动态类型标注。源代码中的类型标注以及 elaboration 引入的类型标注都是静态的，也就是说，它们的目标类型在静态时已知。相比之下，归约会引入动态类型标注，其目标类型只有在运行时才知道。例如，如 §3.3 所述，求值函数应用时，参数会被标注为所应用闭包的定义域证据，返回值同样会被标注为其陪域证据。编译器无法预先知道运行时从闭包证据中获得的确切类型，但鉴于该证据是单调演变的，静态信息所表示的是上界。利用这些上界，可以确定某些动态类型标注并非必要。具体而言，这项优化对静态类型代码极为有效，因为此时上界是一个静态类型（即完全精确），因此可以完全避免动态类型标注，从而生成无需运行时类型检查的高效代码。

例如，考虑一个应用`f 1`，其中 `f` 的类型为 整数类型→整数类型。elaboration 的结果是`f ascribe(`⟨整数类型⟩`1, dom(0,f))`。这里，动态 ascription 并非必需，因为 ⟨整数类型⟩ 不可能获得更高的精度，因此可以将应用简化为 `f` ⟨整数类型⟩`1`。

浮点数优化。 与整数和布尔值相反，浮点值需要使用 64 位，因此无法在保留其即时性的同时为其添加标签。因此，动态语言使用分配在堆上的装箱浮点数。对于渐进语言，统一使用装箱浮点数实用且简单，但这会错失在静态类型代码中使用即时浮点数的机会，从而可能降低浮点运算密集场景下的性能。

在观察到 Grift 在浮点密集型基准测试上的最佳性能后，我们注意到，尽管 Kuhlenschmidt 等人并未明确报告这一点， [27], Grift 执行了一项优化，即在静态类型代码中拆箱浮点数。[2] 由于不知道这项优化是否已有名称，我们在此称之为双浮点优化（DFO）。DFO 将从**浮点数**到**?** 的即时浮点数注入视为装箱操作，实际上会在堆上分配该值；与之相应，从**?** 到**浮点数**的投影则视为拆箱操作。

我们已在 GrEv 中实现 DFO，使用证据跟踪浮点数的有类型与无类型边界，并按需执行装箱和拆箱。这确保了对静态类型浮点值进行操作的函数会被编译为直接对即时值执行浮点运算的代码。例如，调用 `print_浮点数 x` 时，如果`x:`**浮点数** ，则无需拆箱；而同一表达式中若`x:`**?** ，则会在应用打印原语之前生成拆箱操作。

遗憾的是，DFO 意味着某些原本可以移除的动态类型标注不再能够通过优化消除，因为它们是执行装箱和拆箱（以及检查）所必需的。鉴于此，且目前没有报告分析 DFO 在实际基准测试中的效应，GrEv 中的 DFO 是可选的；我们实际上会在 §4 中研究其性能影响。

3.5 与 Grift 的比较

GrEv 和 Grift 编译的源语言基本相同，不过 GrEv 采用类似 OCaml 的语法，而 Grift 使用类似 Scheme 的语法。两种编译器通常遵循相同的编译策略，例如将所有数据结构分配在堆上。递归类型的处理略有不同：它们在 GrEv 中仅用于变体类型，且是同递归的；而在 Grift 中则是通用的，且是等递归的 [37]。两种编译器都会使用 Keep 等人提出的技术，将某些闭包调用转换为直接函数调用 [26], ，但除此之外，不执行任何通用优化（例如，类型推断和特化、常量折叠、复制传播、内联等）。至于渐进式类型本身的优化，GrEv 和 Grift 执行的优化类似，详见 §3.4。GrEv 以 OCaml 实现，并编译为 LLVM IR；Grift 以 Racket 实现并编译为 C，再由 C 编译为 LLVM IR。因此，两种编译器都受益于 LLVM 执行的后端优化。在内存管理方面，GrEv 和 Grift 使用相同的 Boehm-Demers-Weiser 保守式垃圾回收器，该回收器实现了分代标记-清除算法 [10, 18]。

Grift 和 GrEv 的一个首要区别在于它们处理运行时类型信息的方式，以及执行检查的方式，即 强制转换与证据。正如我们所见，在本文考察的简单类型环境中，证据就是类型。强制转换不同于断言中使用的类型，因为它们表示操作（例如注入和投影、恒等和失败、从结构中读取和写入等）。这两种编译器的一个主要区别在于值的内存表示：GrEv 中的表示比 Grift 中的更简单、更统一。在 Grift 中，尚未进行类型转换的值不携带任何类型信息；但一旦它们被转换为未知类型，就会表示为指向堆中一个对的指针，该对由底层值及其类型构成。这就是上文解释的 DFO 通用原则，但应用于所有值，并考虑到各类值的具体特性。对于布尔值和整数，类型信息存储在最低的 3 个有效位中，不会创建引用。Grift 中的闭包

[2] GGrift 还会对其他基本类型执行类似的优化，但最明显的效果体现在浮点数上 ts.

可以是实际闭包，也可以是代理闭包。实际闭包由一个函数指针、一个指向用于类型转换闭包的函数的指针，以及闭包捕获的环境构成。代理闭包由一个指向包装器的函数指针、一个指向底层闭包的指针，以及一个指向强制转换的指针构成。当闭包进行类型转换时，Grift 需要在运行时判别它属于哪种闭包。如果是实际闭包，Grift 只需调用该闭包的类型转换函数，从而创建一个新的代理闭包。如果是代理闭包，Grift 会将此次类型转换中的强制转换与代理闭包中的强制转换组合起来，并创建一个新的代理；该代理沿用现有的包装器和底层闭包，但使用组合得到的强制转换。

就性能而言，Grift 的非均匀值表示在完全类型化配置中有利，因为它可以利用与 Grift 的完全静态类型化变体（称为 StaticGrift）相同的表示方式 [27]。不过，当某个值首次被类型转换为精确度较低的类型后，便会创建该值的代理版本，而这一优势也随之消失。对两种编译器的性能进行实验评估（§4），有助于揭示这些表示选择的影响。

4 性能评估

现在我们转向 GrEv 的性能评估。在讨论与基于证据实现渐进式类型直接相关的研究问题之前，我们首先希望确保 Grift 和 GrEv 编译器具有可比性。

> **RQ1. Grift 和 GrEv 的静态基线是否具有可比性？**（§4.2）这一点很重要，因为它可以排除这样一种可能性：我们在对两个编译器进行渐进式程序基准测试时观察到的结果，实际上主要归因于与渐进式类型管理无关的实现差异。

接下来，我们着手评估 GrEv 中渐进式类型带来的开销，并将其与 Grift 进行比较。我们将这个一般性问题细化如下：

> **研究问题2。总体而言，GrEv 的表现能否与 Grift 相媲美？**（§4.3）我们通过分析两种编译器在以下程序上的性能来研究这个问题：(a) 全类型程序，(b) 全无类型程序，以及 (c) 部分类型化程序。
>
> **RQ3。不同的语义模式如何影响性能？**（§4.4）为了评估单调语义的影响，我们在 Grift 的可用实现上运行基准测试——Grift/G（采用受保护语义）和 Grift/MS（采用单调结构与守卫闭包）——以及 GrEv 的实现：GrEv/G（受保护）、GrEv/MC（单调闭包、受保护结构）和 GrEv/MV（所有值均为单调）。
>
> **RQ4。双浮点优化的效果如何？**（§4.5）我们考察启用和未启用双浮点优化（DFO §3.4）的 GrEv，并在浮点密集型程序上，使用两种编译器的受保护语义将其与 Grift 进行比较。

4.1 实验方法

Grift 基准测试套件。我们沿用 Grift 代码仓库中的基准测试套件，其中包含 8 项基准测试，这些测试曾在最初的 Grift 论文中进行评估 [27]，；该论文仅测量了采用代理数据结构的 Grift。此外，Grift 代码仓库还新增了 2 项基准测试，用于评估单调引用的性能影响 [4]。这 8 项基准测试来自多种来源，包括 Scheme 基准测试套件（R6RS）、PARSEC 基准测试、计算机语言基准测试游戏（CLBG）、渐进式类型性能基准测试（GTP），以及两种教科书算法（矩阵乘法和快速排序）。另外 2 项基准测试涉及数据结构。图 4 概述了这些基准测试；最后四项会用到浮点数。

基准测试	源代码	描述	
数组	Grift	递归函数中的迭代数组生成与反转	RQ1、2、3
矩阵乘法	教科书	用于整数矩阵乘法的三重嵌套循环	
快速排序	教科书	对整数数组进行排序（最坏情况，即输入已排序）	
qsort-mpairs	Grift	快速排序的一种变体，用于对整数对数组进行排序	
筛法	GTP	通过埃拉托斯特尼筛法计算素数，使用流（递归类型）	
tak	R6RS	三重递归整数函数（Takeuchi）	
blackscholes	PARSEC	布莱克-斯科尔斯偏微分方程的数值计算	RQ1,4
fft	R6RS	对实数点计算快速傅里叶变换	
N体问题	CLBG	使用辛积分器计算木星行星的轨道	
ray	R6RS	对场景进行光线追踪	

图4. 基准测试套件。

实验设置。实验在一台专用机器上进行，该机器搭载 AMD Ryzen 9 9900X3D 24 核处理器，运行频率为 4.4 GHz（128 MB L3 缓存和 12 MB L2 缓存），配备 123 GB 内存，运行 Debian Linux 13.2。LLVM 版本为 19.1.7，Racket 版本为 8.18（Chez Scheme）。由于 Racket 软件包管理器中提供的 Grift 版本无法在较新的 Linux 发行版上安装，我们使用容器安装 Grift 并编译基准测试，但在容器外运行各项配置。我们使用基于 Ubuntu 20.04 的 Docker 镜像安装 Grift 所需的依赖项，并从 GitHub 上可用的源代码安装特定版本的 Grift（链接见 Grift/G 和 Grift/MS）。此 Docker 镜像中使用的 Racket 版本为 8.18（Chez Scheme），Clang 版本为 10.0.0。我们尝试使用较新版本的 Clang 运行 Grift，但未能成功。所有时间测量均为实际运行时间，取自各语言的内部计时工具。对于部分类型化配置，结果为 10 次运行的平均值；其他配置的结果则为 50 次运行的平均值。

测量性能格。Takikawa 等人 [45]最先指出，评估渐进类型语言的性能时，对于每个待求值的程序，都必须考虑通过添加或减少类型注释所能得到的所有可能版本。在GTLC+所支持的细粒度渐进类型中，所有变量绑定都可以带有类型注释，也可以不带；更糟的是，类型注释的精确程度也可能有所不同。为应对这种指数级的空间膨胀，我们采用了 Greenman 和 Migeed [21], 同样用于评估 Grift 的线性采样方法 [4, 27]。

具体而言，我们重新实现了 Grift 的“动态化器”工具，使其能够处理 GrEv 所处理的 GTLC+语法。该工具接收一个静态类型程序、样本数量以及要均匀抽样的分箱数量。它会定位程序中的所有类型注解，并生成这些类型的渐进版本，同时确保生成的配置落在目标分箱内。为确保随机性，我们对类型注解列表进行随机加权洗牌。此过程会不断重复，直到所有分箱都达到预期的配置数量。

对于每个基准测试，我们生成的配置数量与原始程序中的静态类型数量成正比。具体而言，对于包含 n 个静态类型节点的程序，我们生成 $10n$个配置。为确保生成的样本涵盖格中静态类型所占比例的不同取值范围，我们将这些配置分配到 10 个分箱中。

4.2 静态基线（RQ1）

图 5 展示了 StaticGrEv 和 StaticGrift 相对于 Racket 的性能。基准测试分为通用基准测试（左）和浮点密集型基准测试（右）。

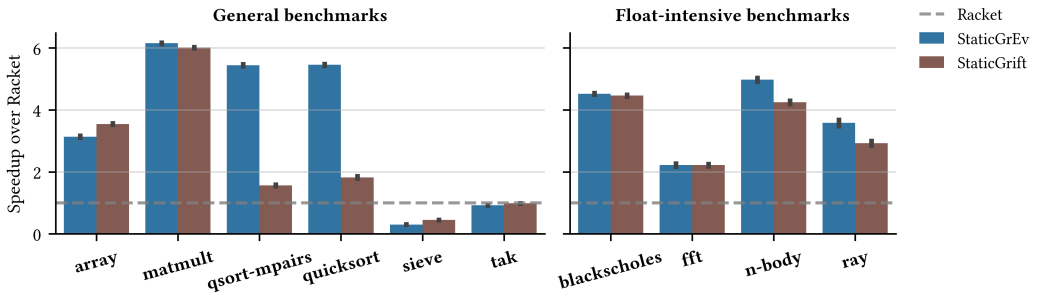

图5. 加速比 StaticGrEv 和 StaticGrift 相对于 Racket 的加速比（越高越好）。通用基准测试显示在左侧，浮点密集型基准测试显示在右侧。

虚线表示 Racket 的性能；高于该线的值表示性能优于 Racket。

对于通用基准测试，我们观察到 StaticGrEv 和 StaticGrift 总体表现相近。显著的例外是 qsort-mpairs 和快速排序：在这两项测试中，StaticGrEv 的表现明显更好（前者加速超过 5 倍，后者接近 2 倍）；而在数组基准测试中，StaticGrift 的表现略好。除筛法基准测试外，两种编译器在所有基准测试中的表现都优于或不逊于 Racket；在筛法基准测试中，两种编译器的速度都慢于 Racket。

在浮点密集型基准测试中，我们再次观察到两款编译器的整体性能相近，StaticGrEv 在 n-body 和 ray 上的性能略好。两款编译器的性能均优于 Racket，提速幅度从 2 倍到 4 倍以上不等。由于浮点数采用堆分配，Racket 在浮点密集型基准测试中处于劣势，而两款静态编译器则利用了未装箱的即时浮点数。

尽管实现方式不同，两款编译器在 10 个基准测试中的 8 个上性能相近；显著的例外是 qsort-mpairs 和快速排序，在这两项测试中，StaticGrEv 的速度明显快于 StaticGrift。

RQ1 尽管实现方式不同，Grift 和 GrEv 的静态基线——即 StaticGrift 和 StaticGrEv——在大多数基准测试中的表现相当。

这一结果证实，在渐进式程序上比较 GrEv 和 Grift 是合理的，因为两款编译器之间与渐进式类型机制无关的性能差异总体上很小。

4.3 通用性能（研究问题2）

现在，我们使用通用基准测试来比较 Grift 和 GrEv 的性能，以评估总体而言 GrEv 是否能与 Grift 竞争。在这些基准测试中，我们使用 GrEv 和 Grift 的不同变体：GrEv/G、GrEv/MC、GrEv/MV、Grift/G 和 Grift/MS。本节重点介绍关于这两个编译器的一般观察；关于各种单调语义的性能影响的具体发现，我们将在 §4.4 中讨论。

全类型程序。 图 6(a) 展示了两种编译器在全类型程序上的相对性能，以相对于 StaticGrift 的加速比表示。我们观察到，在大多数基准测试中，Grift 的性能优于 GrEv。在tak 上，两种编译器的性能相近，接近 StaticGrift。总体而言，GrEv 的表现不如 StaticGrEv——加速比从 0.25x

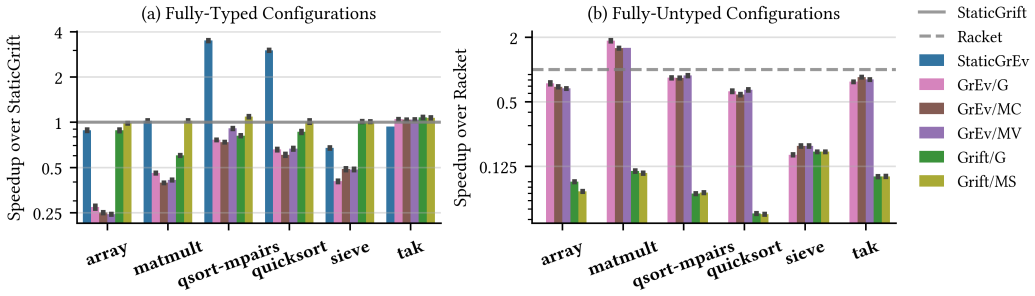

图 6. GrEv 和 Grift 在全有类型和全无类型程序上的加速比（越高越好）。水平线表示相应的基线性能：(a) 全有类型配置，加速比以 StaticGrift 为基准；(b) 全无类型配置，加速比以 Racket 为基准。

在 数组 到 tak，加速比接近 1x。GrEv 无法像 Grift 那样充分利用静态类型，并且仅在一个基准测试中实现了与 StaticGrift 相当的性能。这似乎与 §3.5 中所述有关：Grift 对尚未进行类型转换的值采用与 StaticGrift 类似的表示方式，并且仅在值发生类型转换时才创建代理。相比之下，GrEv 使用统一值表示（表 1），因此即使在完全类型化配置下，渐进式类型也会带来一定开销。

全无类型程序。图6(b) 展示了两种编译器在全无类型程序上的相对性能，即相对于 Racket 的加速比。可以看出，GrEv 在无类型程序上的表现明显优于 Grift。GrEv 的性能可与 Racket 相媲美，而 Grift 的速度始终至少比 Racket 慢 5 倍。Grift 的加速比在 0.05 倍到 0.2 倍之间，GrEv 的加速比则从接近 0.2 倍到超过 1 倍不等——GrEv 在矩阵乘法上甚至优于 Racket。唯一的例外是筛法，两种编译器在该程序上的表现都同样糟糕。鉴于筛法大量使用递归和闭包，并且 GrEv 和 Grift 都没有优化闭包，也没有执行 Clang/LLVM 所做之外的通用优化，我们推测，这种糟糕的性能可能是由于 Racket 对这些特性的实现经过了优化。

部分类型化程序。 在图 7 中，我们比较了 GrEv 和 Grift 的不同变体的性能，重点关注基准测试的部分类型化配置。配置被划分为 10 个分箱。每个分箱对应静态类型占比的一个区间，范围从 0% （ x-轴左侧的 0.0）到 100% （右侧的 1.0）。此外，对于每个基准测试，我们手动加入完全类型化和完全无类型化的配置，它们分别作为 x-轴上 0.0 和 1.0 处的两个极端。

对于部分类型化配置，我们观察到 GrEv 总体上表现更好，但完全类型化配置除外（如上文所述）。最值得注意的是，与 Grift 相比，GrEv 在不同配置间的性能差异更小，这表明 GrEv 的性能更可预测。对于 GrEv，我们看到“平坦的分布图”：各组数据点集中在某些运行时附近，在不同静态类型比例下形成水平线。这表明，某些类型注解的有无会使配置的运行时出现离散的增减，而不受整个程序中静态类型比例的影响。对于 Grift，我们观察到一些配置的静态类型比例高于 0.5，且其性能远优于其他配置。[3]

[3]对于 Grift，在完全类型化配置左侧附近似乎缺少一些配置。这与 Grift 生成配置的方式有关：我们发现 Grift 动态化器报告的精度大于 1，这自然

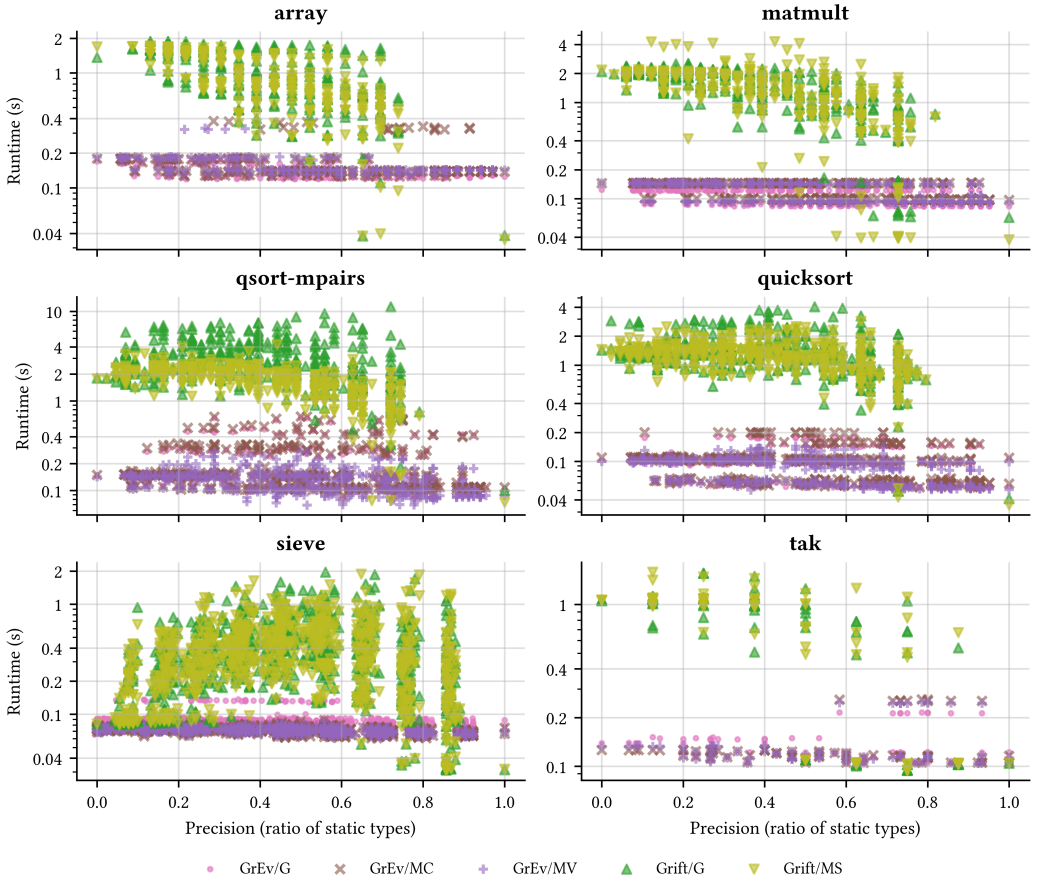

图 7. GrEv 和 Grift 在部分类型化配置下的性能。x 轴表示静态类型所占比例（0 = 完全无类型， 1 = 完全类型化）。y 轴以对数刻度显示运行时（秒），数值越低越好。每个标记代表一个单独的配置，该配置运行了 10 次。

在图8 中，我们比较了 GrEv 和 Grift 在部分类型化配置下、采用不同语义时的累积性能。 x轴表示相对于 Racket 的相对减速比，采用对数刻度，而 y轴表示性能超过该减速比的配置所占百分比。垂直线表示所有配置的性能完全相同。线条越接近垂直，表示不同配置间的性能越可预测。接近水平的线则表示从一组配置到另一组配置时，运行时会出现离散的增加。

该可视化结果证实，除在（接近）完全类型化配置上 Grift 表现更好外，GrEv 在整个范围内的整体性能都显著优于 Grift。对于绝大多数配置，GrEv 的表现都远胜 Grift（请注意， x 轴采用对数刻度）。一个值得注意的例子是矩阵乘法：GrEv 的平均速度是 Racket 的两倍

不可能。通过手动检查源代码，我们发现 Grift 动态化器报告的精度值有误，这与它按每个基准测试各自特定的常数错误计数静态类型的数量相符。为纠正这一问题，我们应用了一个调整因子，将显示的精度向左移动。在某些基准测试中，这会导致出现一段没有生成任何配置的精度区间。

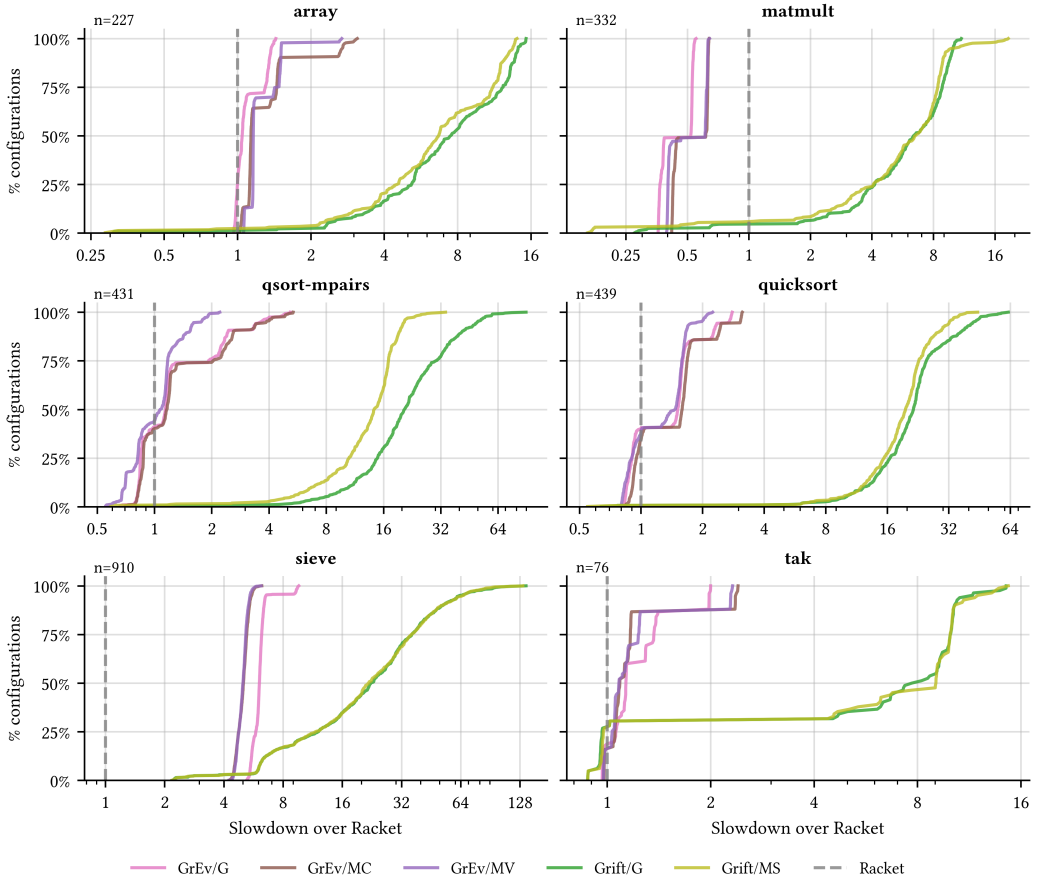

图8. GrEv 和 Grift 在部分类型化配置上的累积性能。x 轴表示相对于 Racket 的减速情况，采用对数刻度（越靠左越好）。y 轴表示减速程度高于该值的配置所占百分比。每种配置均运行了 10 次。

，而在超过 95% 的配置中，Grift 的速度比 Racket 慢（最多慢 16 倍）。GrEv 的性能也更可预测：其最佳性能与最差性能之间的差异要小得多。例如，在筛法上，Grift 在最佳和最差情况下的减速比分别约为 2 倍和 128 倍，而 GrEv/G 的性能范围则在 5 倍到 9 倍之间。

两种编译器在某些基准测试中都呈现出“不连续性”（接近水平的线段）：qsort-mpairs 以及快速排序 在 GrEv 中，以及 tak在 GrEv 和 Grift 中均有此现象。这与图 7 中的垂直间隙相对应，该区域内不存在任何配置。这表明，源程序中某些静态类型的存在会对程序整体性能产生重要影响。理想情况下，应当避免这些情况，因为这意味着程序员只需更改一个类型注释，就可能导致性能发生巨大变化。这印证了先前工作的观察，也凸显了迁移式类型化研究的重要性 [12, 13]，这类研究旨在帮助程序员在类型谱系中进行选择。就此而言，GrEv 明显比 Grift 更具可预测性，这有力地说明了 GrEv 的基于证据的方法更具优势。

此外，值得注意的是，与 Grift 相比，GrEv 中出现不连续性的配置所占比例往往要小得多（从图形上看，不连续性出现在更靠上的位置；在 sieve 和 tak 中尤为明显）。

研究问题2 总体而言，GrEv 的性能显著优于 Grift，但全类型程序除外；与 Grift 相比，GrEv 在整个类型精度范围内的表现也更加可预测。

这一结果表明，旨在将全类型程序的开销降至最低（使用非均匀值表示）这一目标，与通用性能以及整个类型精度范围内的稳定性之间存在冲突。下文对 DFO 的性能分析（§4.5）将针对浮点数的具体情况深入探讨这一问题。

4.4 语义模式（RQ3）

全类型程序。 图 6(a) 中呈现的结果显示了 Grift 中单调引用的优势：Grift/MS 的性能优于 Grift/G，在所有基准测试中都实现了接近 1 倍的加速；而 Grift/G 在数组、矩阵乘法、qsort-mpairs 和快速排序中出现了较为明显的减速。在 GrEv 的三种语义之间，我们观察到性能差异较小：GrEv/G 在数组 和 矩阵乘法上略快，GrEv/MV 在 qsort-mpairs 上略快，而两个单调版本在 筛法上都略快。总体而言，在全类型程序中，Grift 从单调结构中获得的收益更大。这是意料之中的，因为 Grift 实现了深层单调结构。这意味着，当一个向量被类型转换为静态类型时，Grift 会对该向量中的所有值执行类型转换；从此以后，读取或写入该向量时便无需再执行类型转换。相比之下，GrEv 中的浅层单调结构无法避免从该向量读取或向其写入时的检查——它们只能在类型标注时避免新的代理分配。

全无类型程序。回看图6（b），可以看到，单调语义对全无类型程序几乎没有影响。这并不特别令人意外，因为单调语义旨在利用静态类型信息，而这种情形下并不存在静态类型信息。具体而言，在矩阵乘法中，GrEv/G 的表现略优于另外两种语义；而在筛法中则略差。同样，Grift 的两种语义性能相近，其中 Grift/G 在数组中略有优势。

部分类型化程序。在部分类型化配置下（图8），单调语义的优势确实有所体现。在 Grift 中，我们观察到数组、qsort-mpairs 和快速排序的性能显著提升；其余基准测试的性能则相近，但矩阵乘法除外，在表现最差的配置中出现了明显的性能下降。对于 GrEv，我们看到qsort-mpairs、快速排序和筛法的性能显著提升。值得注意的是，在qsort-mpairs 中，GrEv/G 和 GrEv/MC 的性能都下降了 5 倍以上，而 GrEv/MV 最严重的性能下降不到 2 倍。不过，在某些基准测试中，单调语义的表现不如守卫语义。对于 GrEv，单调语义版本在数组、矩阵乘法和tak中表现较差。

GrEv/MV 表现不佳的一个基准测试是数组。该基准测试涉及使用循环和相互递归函数来生成并反转整数数组。为了理解这一令人意外的结果，我们分析了最差配置的性能，并比较了 GrEv 的三种语义在启用和未启用 Clang/LLVM 优化时的结果。我们观察到，禁用此类优化时，差异微不足道，这表明在该基准测试中，GrEv/G 的某些特性更适合进行优化。

根据基准测试的不同，我们观察到 GrEv/MC 的表现有时接近 GrEv/G（qsort-mpairs、快速排序），有时接近 GrEv/MV（矩阵乘法、筛法、tak）。这表明 GrEv/G 与 GrEv/MV 之间的差异有时源于单调闭包，有时则源于单调结构。筛法和 tak 都大量使用递归，且不使用数组，因此 GrEv/MC 的表现接近 GrEv/MV 并不令人意外。同样，qsort-mpairs 和快速排序都大量使用数组，因此在这些基准测试中，GrEv/MC 的表现更接近 GrEv/G 也在意料之中。

RQ3 对于部分类型化配置，两种编译器都能从单调语义中获益，尽管在某些基准测试中，它们的表现可能相近，甚至略差。单调闭包也有同样的效果，在某些场景下，它们能有效补充单调结构。

4.5 双浮点优化（RQ4）

为了更好地了解 §3.4 中所述双浮点优化（DFO）的影响，我们报告了 Grift（启用 DFO）与启用和未启用 DFO 的 GrEv 之间的比较。在图 9中，我们比较了 GrEv/G（未启用 DFO）、GrEv/G-DFO 和 Grift/G 在浮点基准测试（图 4）上的累积性能。

我们观察到，DFO 能显著提升性能：在所有配置下的 blackscholes，以及表现最佳的配置下的快速傅里叶变换和 N体问题中，都能看到这种提升。添加 DFO 后，表现最佳的 25% 配置的减速幅度显著下降：blackscholes 从超过 7 倍降至不到 3 倍，快速傅里叶变换则从超过 9 倍降至接近 4 倍。在快速傅里叶变换和 N体问题中，这些提升出现在几乎完全类型化的配置中。

使用这两种编译器时，某些配置的性能远差于 Racket。尤其是在 ray 基准测试中，这三种实现的最坏情况性能下降都相当显著：GrEv/G-DFO 的最坏情况性能下降超过 256 倍，Grift 超过 70 倍，而 GrEv/G 约为 60 倍。

与采用 DFO 的实现相比，GrEv/G 的性能表现明显更可预测：在全部四项基准测试中，其曲线几乎垂直，并且在其中三项基准测试中的中位性能更好（blackscholes 除外）。相比之下，Grift/G 和 GrEv/G-DFO 的最佳情况与最坏情况性能之间都存在很大差异。最显著的例子是 ray：Grift/G 的性能下降幅度从不到 0.4 倍到超过 64 倍不等，而 GrEv/G-DFO 则从不到 32 倍到超过 128 倍不等。

就最佳情况性能而言，表现最好的实现是 Grift/G。它在所有基准测试中都取得了良好的最佳情况性能，在四项基准测试中的三项（blackscholes、N体问题和 ray）中胜过 Racket；但除 blackscholes 外，它在所有基准测试中的最坏情况性能都不如 GrEv/G。

GrEv/G-DFO 并非在所有浮点基准测试中都优于 GrEv/G。在 blackscholes 中，它在所有配置下都取得了更好的性能，但在其他基准测试中，提升并不明显。在快速傅里叶变换和 N体问题上，它的最佳情况下性能更好，但最差情况下性能则显著更差。此外，在 ray 中，绝大多数配置下它的表现都不如 GrEv/G。

我们确定了两个可能导致这些性能回退的因素。首先，如第3.4节所述，GrEv/G-DFO 无法执行 GrEv/G 能够执行的一些优化，例如移除不必要的动态类型标注。其次，实现 DFO 必须处理浮点数的不同运行时表示。这使一致传递性的实现更加复杂，并带来了额外的运行时开销。在某些基准测试中，这些因素的影响可能超过 DFO 带来的收益，从而导致性能下降。此外，使用 DFO 时，当一个值

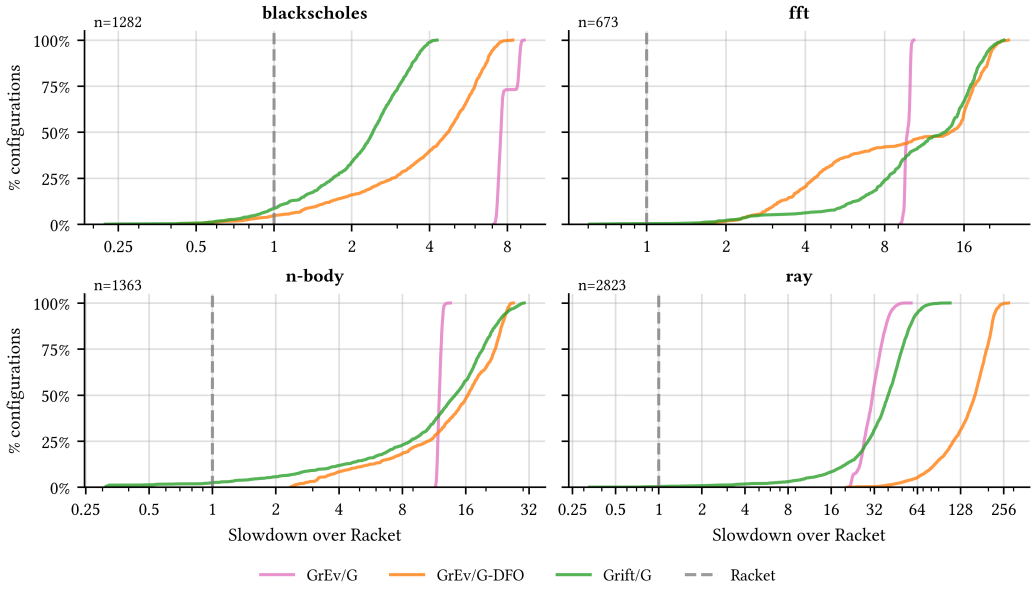

图9。GrEv 在采用和未采用双浮点优化（DFO）时的累积性能，以及浮点基准测试中部分类型化配置下的 Grift 性能。横轴表示相对于 Racket 的减速程度（采用对数刻度）。纵轴表示减速程度高于该值的配置数量。

的类型为 **浮点数** 并被标注为**?**时，需要创建一个新的堆分配浮点数——没有 DFO 时，标注为**?**是免费的。这意味着，根据有类型与无类型边界出现的位置，DFO 可能会执行许多在没有 DFO 时并不需要的内存分配。另一方面，对于包含大量针对已知为浮点数的值进行运算的程序，DFO 可以避免对未装箱浮点值进行装箱，也可以避免在堆上分配新的浮点数。

RQ4 DFO 对 GrEv 的益处并不一致：它在某些基准测试中提供了更好的最佳情况性能，但最差情况性能更差。Grift 的最佳情况性能优于 GrEv，但在四个基准测试中的三个测试里，其最差情况性能更差。GrEv/G 比 GrEv/G-DFO 和 Grift/G 更加可预测。

这项关于 DFO 性能影响的分析倾向于证实这一观察：为了提高静态类型代码的效率而使用不同的值表示，可能会导致部分类型化配置中的性能大幅下降。利用程序分析（包括动态分析）来更好地选择应使用哪种表示形式，或许是一个值得探索的方向。

4.6 对有效性的威胁

Grift 和 GrEv 都是面向小型语言的相当直观的编译器，GTLC+。这里获得的结果在处理更复杂的语言时可能无法推广，例如包含参数多态、记录子类型等特性的语言。Grift 基准测试由 Kuhlenschmidt 等人从其他基准测试套件中 [27]挑选出来，选择依据是所涵盖的特性

GTLC+。因此，有几个基准测试涉及算术运算，总体而言，结果可能会因其他类型的基准测试而有所不同。

另一个效度威胁是，GrEv 与 Grift 之间观察到的差异可能源于与渐进式类型无关的实现差异，例如 Grift 编译到 C，而 GrEv 则直接编译到 LLVM IR；又或者，我们运行 Grift 时不得不使用比运行 GrEv 时稍旧的 Clang 版本。如 §3.5 所述，Grift 和 GrEv 处理递归类型的方式也略有不同，但只有一个基准测试（筛法）使用了这些类型。总的来说，我们的第一个研究问题（§4.2）回应了这一威胁，表明 StaticGrift 和 StaticGrEv 在这些基准测试上的整体性能相近。

尽管两种编译器执行的优化基本相同，Almahallawi [4] 报告称，Grift/MS 使用哈希驻留来复用并高效比较结构类型，但没有报告该技术的具体影响。目前，我们尚未在 GrEv 中为证据实现哈希驻留，不过希望在不久的将来评估其有效性。

关于求值方法，两种编译器使用的源语言在语法上有所不同，这也意味着存在一些有效性威胁。我们必须手动将 Grift 基准测试翻译成 GrEv 源语言，而基准测试翻译中的细微差异可能会导致性能差异。为缓解这一威胁，我们非常谨慎地确保译文尽可能贴近对应的 Grift 版本，并检查它们确实会计算出相同的结果。此外，尽管部分类型化配置是通过类似的样本生成算法生成的，但两者并不完全相同；生成的样本可能会使某种实现占优。不过，我们通过为每个精度区间（静态类型的比例）生成大量样本来降低这一风险，因此这些个别差异极不可能影响总体结果。

最后，本研究仅评估了渐进简单类型的影响，结果未必适用于更复杂的类型。本研究并未探讨更高级渐进类型系统的基于证据的编译。Grift 目前也仅支持渐进简单类型。AGT 文献探讨了适用于多种类型系统的不同证据概念和一致传递性，但迄今为止只开发出了概念验证解释器。如今可以在 GrEv 的背景下研究这些进展，从而开启新的实现挑战。

5 相关工作

渐进式类型研究涵盖多个方面，从语义基础到编译器工程和性能优化。我们围绕四个主要主题展开讨论：(i) 抽象渐进类型（AGT）及其应用，(ii) 实现与性能研究，(iii) 运行时表示与优化技术，以及 (iv) 名义方法和其他提高效率的方法。

*AGT 的基础与基于证据的语义。*抽象渐进类型（AGT）框架 [19] 提供了一种系统化方法，可通过抽象解释 [17]推导渐进类型系统。该设计已扩展至多种类型系统，包括效应 [8], 、细化类型 [30, 54], 、集合论类型 [15, 51], 、安全类型 [49], 、参数多态 [28, 29, 50], 、灵活数据类型 [32], 、概率编程 [59], ，以及近期提出的敏感性类型 [5]。AGT 的应用一直主要停留在理论层面；除原型解释器外，此前尚未探索基于证据的运行时模型的高效实现。本工作通过将基于证据的语义映射到底层表示，并评估其相对于 Grift 基于强制转换的实现的效率，弥补了这一空白。

*实现与性能研究。*早期健全渐进类型的实际实现暴露出严重的性能限制。Takikawa 等人 [45] 通过广泛的

对 Typed Racket 的评估表明，健全渐进类型引入的运行时检查可能使程序速度降低几个数量级。这一结果推动了持续提升健全渐进类型效率的努力。一个重要里程碑是 Grift 编译器 [27], 它是首个面向渐进类型语言的提前编译器，支持结构类型和细粒度、健全的语义。Grift 同时支持基于类型的强制转换和节省空间的强制转换 [22], 在运行时规范化强制转换链，以控制空间和时间开销。它支持两种运行时模式： 受保护语义，即将强制转换附加到堆值的代理上；以及 单调引用 [44], 通过运行时类型信息强制其单调地演变为更精确的类型。Almahallawi[4] 证实，单调引用能够在不损害可靠性的情况下显著提升速度。我们的编译器 GrEv 借鉴了其中一些思想，但以证据取代强制转换，并探索将单调性应用于闭包，而不仅仅是数据结构。

运行时表示与优化。 提高空间效率和减少冗余检查一直是提升渐进式类型性能的核心目标。Herman 等人 [22]引入了强制转换，将运行时类型转换表示为紧凑且可组合的形式。Toro 和 Tanter[52] 随后证明，对于可变引用，强制转换与证据在表达能力和紧凑性上等价。他们还表明，采用类似的非标准归约规则，可以使基于证据的语义实现空间效率。此后，Tsuda 等人 [53]提出强制转换传递风格，将强制转换变为一等公民，从而消除对这类非标准归约的需求。他们在 Grift 中实现了这一方法，成功避免了栈溢出，但执行时间最长增加至三倍。在 GrEv 中，证据值本身已经很紧凑；然而，当前实现无法避免栈溢出。探索并评估类似的证据传递风格，是未来一项值得开展的工作。

除了基于强制转换的实现之外，Vitousek 等人 [55, 57] 还引入了 *Reticulated Python*, 这是一种基于瞬态语义的 Python 渐进式类型变体。瞬态语义仅在类型边界执行浅层的一阶检查，并确保开放世界可靠性，也就是说，渐进式类型代码可以安全地与无类型库交互。Greenman 和 Felleisen [20] 分析了传统守卫语义、瞬态语义和擦除语义各自得出的可靠性结果，并评估了它们在 Typed Racket [48]中的性能。擦除语义依赖底层动态语言的检查机制，因此不适用于我们的场景。

名义渐进类型与结构扩展。Muehlboeck 和 Tate [34] 表明，与采用结构类型体系相比，采用名义类型体系能够为健全渐进类型带来显著更高效的运行时表示。他们对一种名义类型的面向对象核心语言进行了形式化。在他们的设计中，函数类型通过接口表示，闭包则通过类表示。他们的评估采用了 Takikawa 等人 [45] 和 Vitousek 等人 [57],的基准测试，并将性能与 C# [9] 和 Reticulated Python 进行了比较。后来，Muehlboeck 和 Tate [35]扩展了这项工作，支持记录和 λ 表达式等动态类型的结构化构造。

其他优化。 Rastogi 等人 [38] 描述了一种基于推断的方法，可在保持语义的同时消除部分运行时强制转换。Vitousek 等人 [56] 将此方法改造后，用于优化 Reticulated Python 中的瞬态渐进式类型。他们在一组 Python 基准测试上进行的评估表明，部分类型化程序的性能可以接近其无类型对应程序。最近，判别式类型 [14] 引入了基于类型的优化：根据调用点可用的类型信息，为函数生成单独的快速版本和慢速版本；这一方法与 Allende 等人 [3] 在 Gradualtalk [2]中探索的方法类似。这些优化与底层运行时机制正交——无论该机制基于强制转换、类型转换还是证据——因此可以纳入 GrEv，以进一步提升性能。

6 结论

我们介绍了 GrEv，这是一款面向渐进类型语言的新型编译器，专注于结构化、细粒度且健全的渐进类型。GrEv 的目标是低级、不安全的后端（LLVM），因此渐进类型机制负责执行类型相关安全检查。GrEv 与 Grift 的区别在于，它采用了由抽象渐进类型（AGT）这一通用框架推导出的基于证据的语义。对 Grift 基准测试进行的 GrEv 性能评估表明，确实可以利用证据编译渐进式程序，使其性能总体上具有竞争力，而且往往优于基于强制转换的实现。总体而言，GrEv 在整个类型精度范围内也表现出更好的稳定性。这可能是因为与 Grift 中多种表示形式并存的情况相比，GrEv 对证据值采用了更简单、更统一的表示。

在设计和实现 GrEv 的过程中，我们还发现了两个与单调引用相关的新颖语义理念：首先，我们观察到，以浅层方式对证据执行原地更新是一种合理的替代方案。这种方式避免了深层更新的复杂性，仅保留避免代理的可能性，而无法避免所有运行时检查。其次，我们提出了单调闭包，这是将单调引用理念自然扩展到一等函数的结果，而一等函数同样是堆分配值。无论是在 GrEv 还是 Grift 中，单调设计都可能比使用代理展现出更好的性能，尽管我们观察到的性能优势既不普遍，也并非总是明显。这表明，语言设计空间仍有进一步探索的余地，而探索应由更广泛的性能评估来推动。

同样，我们在使用双浮点优化（DFO）方面的经验表明，尽管它确实能在完全类型化配置上实现顶尖性能，但在部分类型化程序上的性能下降，使其相较于直接使用堆分配浮点数而言更难以预测。这证实了这样一种矛盾：一方面追求静态类型程序上的完整性能，另一方面追求涵盖整个部分类型化程序范围的平均性能。这里同样需要开展未来工作，以更好地理解如何在这两者之间取得平衡。

除了上述内容，并结合文献中提出的若干尚待实现的优化外，本工作也为未来研究开辟了许多方向。最值得一提的是，当前 GrEv 的实现仅支持带一致性的简单渐进类型。我们认为，GrEv 为研究多种渐进类型系统的底层实现提供了颇具吸引力的通用基础设施。由于编译流水线的大部分环节都以抽象方式处理证据表示和一致传递性的实现，我们可以利用 AGT 在不同场景中的诸多应用，例如一致子类型、渐进效应、渐进式参数化或渐进敏感性类型等。

引用

[1] Amal Ahmed、Dustin Jamner、Jeremy G. Siek 和 Philip Wadler。2017年。《自由之上的免费定理：有类型与无类型的参数化性》。*ACM*编程语言论文集1，ICFP（2017年9月），39:1–39:28。[2] Esteban Allende、Oscar Callaú、Johan Fabry、Éric Tanter 和 Marcus Denker。2014年。《Smalltalk 的渐进式类型》。计算机编程科学96，1（2014年12月），52–69。[3] Esteban Allende、Johan Fabry 和 Éric Tanter。2013年。《渐进类型对象的类型转换插入策略》。载于第九届 *ACM* 动态语言研讨会论文集（*DLS 2013*）。ACM出版社，美国印第安纳州印第安纳波利斯，27–36。ACM SIGPLAN 通知，49(2)。[4] Deyaaeldeen Almahallawi。2020年。通过单调引用和强制转换实现高效渐进式类型。博士学位论文。印第安纳大学，美国。[5] Damián Árquez、Matías Toro 和 Éric Tanter。2025年。《渐进敏感性类型》。载于第*38*届 *IEEE* 计算机安全基础研讨会论文集。314–329。[6] Felipe Bañados Schwerter、Alison M. Clark、Khurram A. Jafery 和 Ronald Garcia。2021年。《推进抽象渐进类型：精确且空间高效》。*ACM*编程语言论文集5，POPL（2021年1月），1–28。

[7] Felipe Bañados Schwerter、Ronald Garcia 和 Éric Tanter。2014年。《渐进式效果系统理论》。载于第19届 ACM SIGPLAN 函数式编程会议论文集（ICFP 2014）。ACM出版社，瑞典哥德堡，283–295。[8] Felipe Bañados Schwerter、Ronald Garcia 和 Éric Tanter。2016年。《渐进式类型与效果系统》。《函数式编程杂志》26（2016年9月），19:1–19:69。[9] Gavin Bierman、Erik Meijer 和 Mads Torgersen。2010年。《为 C 添加动态类型》♯。载于第24届欧洲面向对象编程会议论文集（ECOOP 2010）（《计算机科学讲义》，6183），Theo D'Hondt（编）。Springer-Verlag，斯洛文尼亚马里博尔，76–100。[10] Hans-Juergen Boehm 和 Mark Weiser。1998年。《非协作环境中的垃圾回收》。《软件——实践与经验》9，18（1998年9月），807–820。[11] Gilad Bracha。2004年。《可插拔类型系统》。载于动态语言复兴研讨会。[12] John Peter Campora、Sheng Chen、Martin Erwig 和 Eric Walkingshaw。2018年。《迁移渐进式类型》。《ACM编程语言论文集》2，POPL（2018年1月），15:1–15:29。[13] John Peter Campora、Sheng Chen 和 Eric Walkingshaw。2018年。《类型转换与成本：协调渐进式类型中的安全性与性能》。《ACM编程语言论文集》2，ICFP（2018年9月），98:1–98:30。[14] John Peter Campora、Mohammad Wahiduzzaman Khan 和 Sheng Chen。2024年。《基于类型的渐进式类型性能优化》。《ACM编程语言论文集》。 8， 2024 2667 2699 10 1145 3632931 POPL（），–。[15] 2017 doi：。/ Giuseppe Castagna 和 Victor Lanvin。。《基于联合类型与交集类型的渐进式类型》。《ACM编程语言论文集》1，ICFP（2017年9月），41:1–41:28。[16] Alonzo Church。1940年。《简单类型理论的一种表述》。《符号逻辑杂志》5，2（1940年6月），56–68。[17] Patrick Cousot 和 Radhia Cousot。1977年。《抽象解释：通过构造或近似不动点，为程序静态分析建立统一的格模型》。载于第四届 ACM 编程语言原理研讨会会议记录（POPL 77）。ACM出版社，美国加利福尼亚州洛杉矶，238–252。[18] Alan J. Demers、Mark D. Weiser、Barry Hayes、Hans-Juergen Boehm、Daniel G. Bobrow 和 Scott Shenker。1990年。《结合分代式与保守式垃圾回收：框架与实现》。载于第十七届 ACM 编程语言原理年度研讨会会议记录，美国加利福尼亚州旧金山，1月 *1990*， 261 269 10 1145 96709 96735 Frances E. Allen（编）。ACM出版社，–。[19] 2016 doi：。/。Ronald Garcia、Alison M. Clark 和 Éric Tanter。《抽象渐进类型》。载于第43届 ACM SIGPLAN-SIGACT 编程语言原理研讨会论文集（POPL 2016），Rastislav Bodík 和 Rupak Majumdar（编）。ACM出版社，美国佛罗里达州圣彼得堡，429–442。勘误见：https://www.cs.ubc.ca/ rxg/agt-erratum.pdf。[20] Ben Greenman 和 Matthias Felleisen。2018年。《类型健全性与性能的谱系》。《ACM编程语言论文集》2，ICFP（2018年9月），71:1–71:32。[21] Ben Greenman 和 Zeina Migeed。2018年。《类型标签健全性的代价》。载于 ACM SIGPLAN 部分求值与程序操纵研讨会论文集（PEPM'18）。[22] David Herman、Aaron Tomb 和 Cormac Flanagan。2010年。《空间高效的渐进式类型检查》。《高阶与符号计算》23，2（2010年6月），167–189。[23] William A. Howard。1980年。《公式即类型：构造的概念》。载于《献给 H. B. Curry：关于组合逻辑、λ演算与形式主义的论文集》，J. P. Seldin 和 J. R. Hindley（编）。Academic Press，美国纽约，479–490。1969年文章的重印版。[24] Yuu Igarashi、Taro Sekiyama 和 Atsushi Igarashi。2017年。《论多态渐进式类型检查》。《ACM编程语言论文集》1，ICFP（2017年9月），40:1–40:29。[25] Lintaro Ina 和 Atsushi Igarashi。2011年。《泛型的渐进式类型检查》。载于第26届 ACM SIGPLAN 面向对象编程系统、语言与应用会议论文集（OOPSLA 2011）。ACM出版社，美国俄勒冈州波特兰，609–624。[26] Andrew W. Keep、Alex Hearn 和 R. Kent Dybvig。2012年。《以 O(0) 时间优化闭包》。载于2012年 Scheme 与函数式编程研讨会论文集。[27] Andre Kuhlenschmidt、Deyaaeldeen Almahallawi 和 Jeremy G. Siek。2019年。《通过强制转换实现结构类型的高效渐进式类型检查》。载于第40届 ACM SIGPLAN 编程语言设计与实现会议论文集（PLDI 2019），美国亚利桑那州菲尼克斯，2019年6月22–26日，Kathryn S. McKinley 和 Kathleen Fisher（编）。ACM，517–532。doi:10.1145/3314221.3314627[28] Elizabeth Labrada、Matías Toro 和 Éric Tanter。2022年。《渐进式 System F》。《ACM杂志》69（2022年10月），38:1–38:78。[29] Elizabeth Labrada、Matías Toro、Éric Tanter 和 Dominique Devriese。2022年。《渐进参数化性的可信封装》。《ACM编程语言论文集》6，OOPSLA1（2022年4月），70:1–70:28。[30] Nico Lehmann 和 Éric Tanter。2017年。《渐进式细化类型》。载于第44届 ACM SIGPLAN-SIGACT 编程语言原理研讨会论文集（POPL 2017）。ACM出版社，法国巴黎，775–788。[31] Meven Lennon-Bertrand、Kenji Maillard、Nicolas Tabareau 和 Éric Tanter。2022 年。《归纳构造演算的渐进化》。《 ACM 编程语言与系统汇刊》 44，2 （ 2022 年 6 月）。doi:10.1145/3495528

[32] Stefan Malewski、Michael Greenberg 和 Éric Tanter。2021年。《渐进结构化数据》。ACM编程语言论文集 5，OOPSLA（2021年11月），126:1–126:28。[33] Yusuke Miyazaki、Taro Sekiyama 和 Atsushi Igarashi。2016年。《用于带界限延续的渐进类型》。载于 STOP研讨会论文集。[34] Fabian Muehlboeck 和 Ross Tate。2017年。《健全渐进类型在名义上依然鲜活且完善》。ACM编程语言论文集 1，OOPSLA（2017年10月），56:1–56:30。[35] Fabian Muehlboeck 和 Ross Tate。2021年。《借助高效渐进类型将结构化代码转变为名义化代码》。ACM编程语言论文集。

5, 2021 1 29 10 1145 3485504 OOPSLA（ ），– 。doi: 。 / Max S.

[36] 2023
New、Eric Giovannini 和 Daniel R. Licata。。《用于效应处理器的渐进类型》。ACM编程语言论文集 7，OOPSLA1（2023年10月），1758–1786。[37] Benjamin C. Pierce。2002年。《类型与程序设计语言》。麻省理工学院出版社，美国马萨诸塞州剑桥。[38] Aseem Rastogi、Avik Chaudhuri 和 Basil Hosmer。2012年。《渐进类型推断的方方面面》。载于第39届ACM SIGPLAN-SIGACT编程语言原理研讨会论文集（POPL 2012）。ACM出版社，美国费城，481–494。[39] Ilya Sergey 和 Dave Clarke。2012年。《渐进所有权类型》。载于第21届欧洲编程语言与系统研讨会论文集（ESOP 2012）（《计算机科学讲义》，第7211卷），Helmut Seidl（编）。Springer-Verlag，爱沙尼亚塔林，579–599。[40] Jeremy Siek 和 Walid Taha。2006年。《用于函数式语言的渐进类型》。载于 Scheme与函数式编程研讨会论文集。81–92。[41] Jeremy Siek 和 Walid Taha。2007年。《用于对象的渐进类型》。载于第21届欧洲面向对象编程会议论文集（ECOOP 2007）（《计算机科学讲义》，4609），Erik Ernst（编）。Springer-Verlag，德国柏林，2–27。[42] Jeremy Siek 和 Philip Wadler。2010年。《有责备与无责备的三元组》。载于第37届ACM SIGPLAN-SIGACT编程语言原理研讨会论文集（POPL 2010）。ACM出版社，西班牙马德里，365–376。[43] Jeremy G. Siek、Michael M. Vitousek、Matteo Cimini 和 John Tang Boyland。2015年。《渐进类型的精化判据》。载于第1届编程语言进展峰会（SNAPL 2015）（《莱布尼茨国际信息学论文集》（LIPIcs），第32卷）。达格施图尔城堡—莱布尼茨信息学中心，美国加利福尼亚州阿西洛马，274–293。[44] Jeremy G. Siek、Michael M. Vitousek、Matteo Cimini、Sam Tobin-Hochstadt 和 Ronald Garcia。2015年。《用于高效渐进类型的单调引用》。载于第24届欧洲编程语言与系统研讨会论文集（ESOP 2015）（《计算机科学讲义》，第9032卷），Jan Vitek（编）。Springer-Verlag，英国伦敦，432–456。[45] Asumu Takikawa、Daniel Feltey、Ben Greenman、Max S. New、Jan Vitek 和 Matthias Felleisen。2016年。《健全渐进类型是否已死？》。载于第43届ACM SIGPLAN-SIGACT编程语言原理研讨会论文集（POPL 2016），Rastislav Bodík 和 Rupak Majumdar（编）。ACM出版社，美国佛罗里达州圣彼得堡，456–468。[46] Asumu Takikawa、T. Stephen Strickland、Christos Dimoulas、Sam Tobin-Hochstadt 和 Matthias Felleisen。2012年。《用于一等类的渐进类型》。载于第27届ACM SIGPLAN面向对象编程系统、语言与应用会议论文集（OOPSLA 2012）。ACM出版社，美国亚利桑那州图森，793–810。[47] Peter Thiemann 和 Luminous Fennell。2014年。《用于带注释类型系统的渐进类型》。载于第23届欧洲编程语言与系统研讨会论文集（ESOP 2014）（《计算机科学讲义》，第8410卷），Zhong Shao（编）。Springer-Verlag，法国格勒诺布尔，47–66。[48] Sam Tobin-Hochstadt 和 Matthias Felleisen。2008年。《Typed Scheme的设计与实现》。载于第35届ACM SIGPLAN-SIGACT编程语言原理研讨会论文集（POPL 2008）。ACM出版社，美国加利福尼亚州旧金山，395–406。[49] Matías Toro、Ronald Garcia 和 Éric Tanter。2018年。《带引用的类型驱动渐进安全》。ACM编程语言与系统汇刊 40，4（2018年11月），16:1–16:55。[50] Matías Toro、Elizabeth Labrada 和 Éric Tanter。2019年。《再探渐进参数性》。ACM编程语言论文集 3，POPL（2019年1月），17:1–17:30。[51] Matías Toro 和 Éric Tanter。2017年。《联合类型的渐进解释》。载于第24届静态分析研讨会论文集（SAS 2017）（《计算机科学讲义》，第10422卷）。Springer-Verlag，美国纽约市，382–404。[52] Matías Toro 和 Éric Tanter。2020年。《渐进引用的抽象化》。计算机编程科学 197（2020年10月），1–65。[53] Yuya Tsuda、Atsushi Igarashi 和 Tomoya Tabuchi。2020年。《强制转换传递风格中的空间高效渐进类型》。载于第34届欧洲面向对象编程会议，ECOOP 2020，2020年11月15日至17日，德国柏林（线上会议）（LIPIcs，第166卷），Robert Hirschfeld 和 Tobias Pape（编）。达格施图尔城堡—莱布尼茨信息学中心，8:1–8:29。doi:10.4230/LIPICS.ECOOP.2020.8

[54] Niki Vazou、Éric Tanter 和 David Van Horn。2018年。渐进式液体类型推断。*ACM*编程语言论文集2，OOPSLA（2018年11月），132:1–132:25。[55] Michael M. Vitousek、Andrew M. Kent、Jeremy G. Siek 和 Jim Baker。2014年。Python渐进类型的设计与评估。载于*DLS'14*，第*10*届*ACM*动态语言研讨会论文集（*SLASH 2014*的一部分），美国俄勒冈州波特兰，*2014*年*10*月*20*日至*24*日，Andrew P. Black 和 Laurence Tratt（编）。ACM，45–56。doi:10.1145/2661088.2661101[56] Michael M. Vitousek、Jeremy G. Siek 和 Avik Chaudhuri。2019年。瞬态渐进式类型检查的优化与评估。载于第*15*届*ACM SIGPLAN*动态语言国际研讨会论文集，*DLS 2019*，希腊雅典，*10*月*20, 2019*，28 41 10 1145 3359619 3359742，Stefan Marr 和[57] 2017 Juan Fumero（编）。ACM，–。doi:。/。Michael M. Vitousek、Cameron Swords 和 Jeremy G. Siek。。小型运行时中的大型类型：渐进式类型系统的开放世界可靠性与协作式归责。载于第*44*届*ACM SIGPLAN*编程语言原理研讨会论文集，*POPL 2017*，法国巴黎，*2017*年*1*月*18*日至*20*日，Giuseppe Castagna 和 Andrew D. Gordon（编）。ACM，762–774。doi:10.1145/3009837.3009849[58] Roger Wolff、Ronald Garcia、Éric Tanter 和 Jonathan Aldrich。2011年。渐进式类型状态。载于第*25*届欧洲面向对象编程会议论文集（*ECOOP 2011*）（《计算机科学讲义》，第*6813*卷），Mira Mezini（编）。Springer-Verlag，英国兰开斯特，459–483。[59] Wenjia Ye、Matías Toro 和 Federico Olmedo。2023年。渐进式概率λ演算。*ACM*编程语言论文集7，OOPSLA1（2023年10月），256–285。