arXiv:2603.19560v1 [cs.PL] 20 Mar 2026

增量式 通过快捷方式记忆化实现实时编程

MARISA KIRISAME*、THOMAS J. PORTER*、RUQING YANG、JIANQIU ZHAO、YUDIWU、IVAN WEI、CYRUS OMAR 和 PAVEL PANCHEKHA

实时编程系统旨在快速向程序员展示程序编辑所产生的动态影响。为此，程序每次被编辑后都需要重新执行；当程序规模变大或变得更加复杂时，这会带来计算上的挑战。因此，实时程序解释器的实现需要具备增量性。本文介绍了 Chordata，这是一种基于 快捷方式记忆化 的增量式程序解释器。它通过观察程序先前版本的执行，学习被称为快捷方式的重复计算模式。随后，当相同或<style id='3'>结构相似的 程序片段再次执行时，它便可以应用这些快捷方式。本文为任何采用基于重写的语义的语言提出了快捷方式记忆化的形式语义，并对关键正确性属性给出了机械化证明。接着，我们将以 CEK 机器表示的 Hazel 实时编程系统的一个变体用 Chordata 表达，并开发了若干用于学习高价值快捷方式的实用启发式方法。我们在学生解决简单编程问题时的编辑轨迹上评估了所得系统。与基线相比，Chordata 达到了 13.03× 的加速比，同时仅产生 19.97× 的内存开销。对于较小的改动和更加复杂的程序，Chordata 能够取得更大的加速效果。此外，我们还展示了 Chordata 即使在单次执行中也能提供加速，并且输入规模越大，加速效果越明显。

CCS 概念：• **软件及其工程** → **语言特性**；运行时环境；• **计算理论** → 操作语义；数据结构设计与分析。

Ad附加关键词与短语：增量计算、记忆化、实时编程

ACM 参考格式：
Marisa Kirisame, Thomas J. Porter*, RuqingYang、JianqiuZhao、Yudi Wu、Ivan Wei、Cyrus Omar 和 Pavel Panchekha。2026年。通过快捷记忆化实现增量式实时编程。1，1（2026年3月），22 页。
https://doi.org/XXXXXXX.XXXXXXX

1 引言

实时编程环境将程序编辑与执行交织在一起，[39], 其历史悠久，成果丰硕。电子表格率先实现了对编辑操作的实时反馈，[2], 此后，众多研究致力于开发其他具有不同实时性程度的系统，其中包括现代系统，如 Hazel（函数式编程）、[23], Jupyter（数据科学）[24] 和 Vercel（Web 开发）[40]。

要让环境真正具有实时感，用户必须能迅速看到编辑结果；延迟时间过长会让用户反复犹豫、容易出错，也会分散注意力 [31]。对于 Hazel 这类系统来说，这一点尤其重要，因为程序会随着每次击键即时执行，即使程序中存在空缺和错误也是如此（图 1）。

0*这些作者对本研究的贡献相同。

作者联系方式：Marisa Kirisame*，marisa@cs.utah.edu；Thomas J. Porter*，thomasjp@umich.edu；Ruqing Yang，ryangaq@connect.ust.hk；Jianqiu Zhao，qiuscau@stu.scau.edu.cn；Yudi Wu，9680384274@g.ecc.u-tokyo.ac.jp；Ivan Wei，ivanwei@umich.edu；Cyrus Omar，comar@umich.edu；Pavel Panchekha，pavpan@cs.utah.edu。

允许为个人或课堂使用而免费制作本作品的全部或部分数字副本或纸质副本，前提是不得出于营利或商业目的制作或分发副本，且副本须附有本声明及首页上的完整引文。对于本作品中由作者以外的其他方拥有版权的组件，必须尊重其版权。注明出处后，可以进行摘要引用。以其他方式复制、转载、发布到服务器或向列表重新分发，均须事先获得特定许可和／或支付费用。请向 permissions@acm.org 申请许可。© 2026年 版权归所有者／作者所有。出版权已授权给 ACM。ACM XXXX-XXXX/2026/3-ARThttps://doi.org/XXXXXXX.XXXXXXX

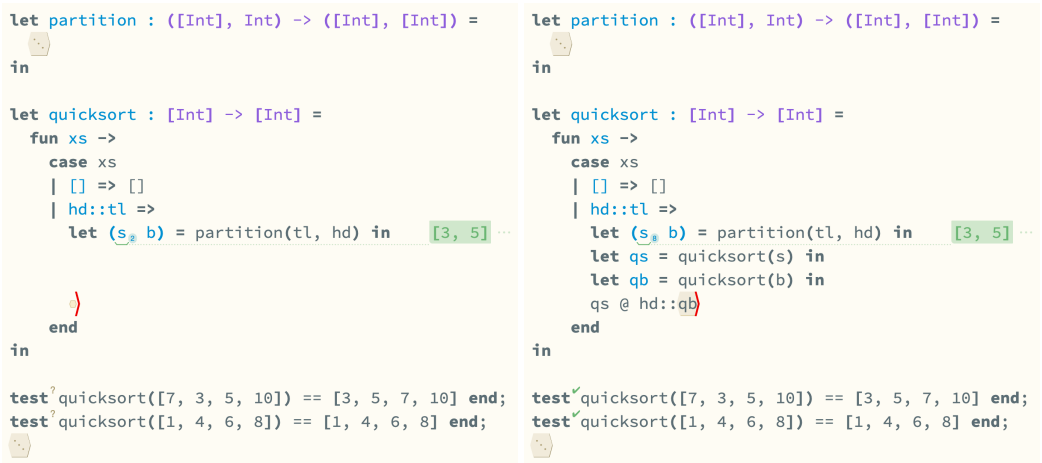


图 1. 在 Hazel 实时编程环境中实现快速排序的屏幕截图。用户编辑函数时，程序可能尚未完成（左侧红色光标处的空缺），和/或存在类型错误（未显示）。用户可以插入探针，以获取有关调用 partition 所得结果中某个组件的实时反馈，此处为 s。随着用户继续填写剩余函数（右侧），Hazel 会在每次按键后重新执行程序，并更新探针（此处显示当前共有 8 个可用的探测值，而测试所显示的初始值只有 2 个）。目前，Hazel 会在每次按键后执行完整的重新执行，即使对于较小的程序也可能产生较高的延迟，例如运行大量测试时（此处省略）。Chordata 的目标是通过从先前的执行中学习快捷方式，加速这些重新执行。

应对这一延迟挑战的一种方法是增量计算。在计算的不同领域中，常常需要缓存先前执行产生的信息，以便在之后执行结构相似的程序（或输入相似的同一程序）时加以复用，从而提升速度。例如，数据库中的差分数据流 [17]、网页浏览器中的增量布局 [15]、增量解析 [41]、类型检查 [26]、编译 [34]，甚至编程系统中的链接 [30]，都采用了这种通用方法。在实时编程系统中，迫切需要一种通用的增量式程序解释器，能够利用编辑过程中共享的程序结构来加快执行。

许多增量式编程系统，包括电子表格 [2] 和构建系统 [18]，都会在粗粒度区分的程序结构（例如模块，或笔记本或电子表格中的单元格）之间构建一个依赖图，再由此实现增量性。编辑只会触发依赖图中下游结构的重新计算。不过，这种方法有两个主要问题。

首先，在实时编程系统中，许多编辑都发生在单个结构内部的局部位置，因此无法从这种方法中受益（**粒度问题**）。例如，在图 1中，所有编辑都发生在快速排序函数内。

其次，这些方法根据计算的语法结构来确定依赖关系，因此难以应对那些使程序在语法上发生显著变化的连续编辑。对于空间和时间上相距较远的代码元素，这些方法往往无法利用其语义相似性（**远距离相似性问题**）。例如，编辑可能包括将表达式重构为辅助函数、将子表达式提升为有界变量、删除重复代码，或暂时移除代码。编辑也可能涉及将代码优化为等价形式，例如将$filter(\lambda_.\text{true}\ x)$替换为 x。我们希望能够利用这些远距离的语义相似性，即使语法已被大幅修改。

1.1 快捷方式记忆化

本文通过引入快捷方式记忆化来解决这些问题。这种方法从先前的运行中学习快捷方式，并在机器再次遇到相似的程序状态时重用它们。具体而言，快捷方式记忆化适用于通过状态之间的步进关系 $\Sigma \mapsto \Sigma'$来执行程序的抽象机。快捷方式是步进模式 $\Sigma[\bar{x}] \mapsto \Sigma'[\bar{x}]$，用于捕获程序执行的多个步骤。这些模式中的变量对状态中未经检查的部分进行抽象，因此它们的匹配范围比单一触发状态更广。

在我们将快捷方式记忆化付诸实现时，我们将其称为 Chordata，并将抽象机固定为 CEK 机器 [7]。随后，我们设计启发式方法来缩小可能快捷方式的搜索空间，选出那些经常匹配且能跳过许多程序状态的快捷方式。我们采用判别树方法，高效地将已学习到的快捷方式集合与当前程序状态进行匹配。

由于 Chordata 学习的是机器状态上的快捷方式，而不是跟踪绑定依赖，因此能否复用取决于执行的结构，而非程序的语法形式。因此，快捷方式记忆化同时解决了**粒度问题**和**远距离相似性问题**。我们在机器状态中统一处理程序代码和输入数据，因此 Chordata 能够自然地处理程序和数据的变化。这使得 Chordata 特别适用于实时编程环境：在这些环境中，程序变化迅速，但通常只是小幅改动。在评估中，我们展示了 Chordata 在选取的一组源自 Hazel 用户的实时程序编辑轨迹上，最高可实现 13.03×的加速，同时仅产生 19.97× 的内存开销。我们将在第5 节展示，重写规则也能加速单次执行，例如在处理重复数据时。

1.2 动机示例

为了说明 Chordata 的核心机制，请看下面这个位翻转函数：

$$f([]) = [] \qquad f(0 :: \ell) = 1 :: f(\ell) \qquad f(1 :: \ell) = 0 :: f(\ell)$$

想一想这个函数的机械式逐步执行过程：每次应用 f（最后一次除外）时，都会先检查列表头部的标识，然后通过压入一个栈帧来递归。最终，栈会以反转的顺序映射出原始列表，栈顶代表列表的最后一个元素，而栈上的每个返回地址都表明 f采取了哪个分支，也就是说，列表中是 0 还是 1。接着，当到达列表尾部时，输出列表便通过从栈中弹出栈帧，并为每个栈帧构造一个列表单元来生成。简而言之，在执行过程中，这个程序有两种状态——一种是沿列表向下递归时的状态，另一种是构造结果时的状态；每种状态都包含一个当前列表值和一个当前栈帧栈。

我们可以这样表示程序执行的这种底层机械视图：

$$\begin{array}{lcll} \ell & \in & \texttt{List} & ::= \quad [] \mid 0 :: \ell \mid 1 :: \ell \\ k & \in & \texttt{Stack} & ::= \quad \texttt{Empty} \mid \texttt{Cons1}\ k \mid \texttt{Cons0}\ k \\ s & \in & \texttt{State} & ::= \quad \texttt{Recurse}\ \ell\ k \mid \texttt{Pop}\ \ell\ k \mid \texttt{Done}\ \ell \end{array}$$

初始执行 $f(\ell)$ 对应于状态 Recurse ℓ Empty。在此 Recurse 状态下，执行沿列表向下递归：

$$\begin{array}{lcl} \texttt{Recurse}\ (0 :: \ell)\ k & \mapsto & \texttt{Recurse}\ \ell\ (\texttt{Cons1}\ k) \\ \texttt{Recurse}\ (1 :: \ell)\ k & \mapsto & \texttt{Recurse}\ \ell\ (\texttt{Cons0}\ k) \end{array}$$

然后，当执行到达列表末尾时，它会切换到弹出状态，从栈中弹出元素并构建输出列表 ℓ_2:

$$\begin{array}{lcl} \texttt{Recurse Nil } k & \mapsto & \texttt{Pop Nil } k \\ \texttt{Pop } \ell \ (\texttt{Cons0 } k) & \mapsto & \texttt{Pop } (0 :: \ell)\ k \\ \texttt{Pop } \ell \ (\texttt{Cons1 } k) & \mapsto & \texttt{Pop } (1 :: \ell)\ k \\ \texttt{Pop } \ell \ \texttt{Empty} & \mapsto & \texttt{Done } \ell \end{array}$$

Fi最后，机器到达完成状态，这表示 f 的执行结束。

这种操作语义的一个重要性质是，它由纯粹的重写规则定义。$\Sigma \mapsto \Sigma'$ 关系中的每一步都没有前提或递归子步骤，因此，这种语义中的执行仅包括寻找一条其左侧与当前状态匹配的规则，并转移到该规则右侧给出的新状态。事实上，每条规则左侧的变量总是互不相同的，而且其右侧只包含左侧中的变量。因此，从概念上讲，执行只涉及匹配构造子和绑定变量。

此外，这些重写规则可以组合起来。例如，任意状态 Recurse $(1 :: 0 :: \ell)\ k$ 都会先转移到 Recurse $(0 :: \ell)$ (Cons0 k)，然后转移到 Recurse ℓ(Cons1 (Cons0 k))，因此，下面这条重写规则虽然不属于上述操作语义，但同样有效:

$$\texttt{Recurse } (1 :: 0 :: \ell)\ k \mapsto \texttt{Recurse } \ell\ (\texttt{Cons1 } (\texttt{Cons0 } k))$$

这条新的重写规则称为快捷方式，它将两个执行步骤概括为一条规则。将其加入操作语义不会改变任何程序的含义，同时还能减少程序执行所需的步骤。Chordata 正是这样做的:通过观察程序执行来推导快捷方式，然后用这些学习得到的规则扩展其操作语义。

1.3 贡献

简而言之，Chordata 是一种基于快捷方式而非依赖图的增量式解释器。其核心贡献如下:

- 用于创建快捷方式的形式语义，并通过 Agda 中的机械化证明，证明规则组合是正确且最一般的（第2节）。
- 用于学习高价值快捷方式的启发式方法（第3.4节）
- 用于匹配快捷方式的高效插入/查找过程（第3.5节）
- 一种用于加速快捷方式应用的数据表示（第 4.2节）
- 一种用于处理复杂重写规则的优化判别树数据结构（第 4.2节）

- 融入 Chordata 的其他优化的集合（第 4节）
- 使用 Hazel 的实时程序轨迹对 Chordata 进行评估，展示了显著的加速效果（第 5节）

2 形式化

Chordata 方法适用于任何由一组模式匹配规则定义了小步归约语义的语法树语言。本节首先展示一种适用的语义类型——CEK 机器，然后将其推广到任意语法树和归约语义。[1] 接着，本节定义了 Chordata 用于寻找快捷方式的组合操作，并证明快捷方式确实存在且有效：也就是说，应用快捷方式所得的结果与

[1]传统的小步语义涉及环境、替换以及递归遍历，以查找可归约表达式；这些操作并非局部模式匹配，因此与 Chordata 并不相契合。

非增量求值所得的结果相同。本节中的每个定理都有 Agda 证明助手中的机械化证明作为依据（该证明已随补充材料一并提供）。

2.1 CEK 抽象机

CEK 抽象机 [7] 是一种通过对由一组构造子构成的语法树应用一组重写规则，为函数式编程语言定义归约语义的方法。这使其非常适合 Chordata 的快捷方式记忆化。在 CEK 机器中，语言语义是机器状态上的步进关系，该机器状态由三个部分组成：*C*ontrol（控制）表达式，即待求值的表达式；*E*nvironment（环境），用于将变量绑定到值；以及*K*ont（续延），即要返回的续延；其中每一部分都有许多可能的构造子。例如，下面定义了一个用于列表构造和析构这一最小语言的 CEK 机器：

$$x \in \text{变量} \quad v \in \text{值} ::= \mathrm{Nil} \mid \mathrm{Cons}\ v\ v \quad c \in \text{控制} ::= x \mid \mathrm{Nil} \mid \mathrm{Cons}\ c\ c \mid \text{匹配}\ c\ [\mathrm{Nil} \to c]\ [\mathrm{Cons}\ x\ x \to c] \quad e \in \text{环境} ::= x := v \quad k \in \text{续延} ::= \mathrm{OK} \mid \mathrm{Cons}\ _\ c\ e\ k \mid \mathrm{Cons}\ v\ _\ k \mid \text{匹配}_\ [\mathrm{Nil} \to c]\ [\mathrm{Cons}\ x\ x \to c]\ e\ k \quad s \in \text{状态} ::= \text{求值}\ c\ e\ k \mid \mathrm{Apply}\ v\ k$$

请注意，虽然 Apply 状态并非标准形式，但它们在文献中早已为人熟知，并确保步进关系完全由一组固定的重写规则定义。实际上，这些额外状态将递归遍历展开为多个原子步骤。[2]

尽管此处的语言十分精简，但大多数函数式语言都可以定义 CEK 机器。CEK 机器程序的执行由状态上的步进关系 $s \mapsto s'$ 定义。关键在于，在 CEK 机器中，整个步进关系都由有限数量的简单模式定义：

$$\begin{array}{lcl}
\text{求值}\ x\ [y := v; e]\ k & \mapsto & \text{求值}\ x\ e\ k \\
\text{求值}\ x\ [x := v; e]\ k & \mapsto & \mathrm{Apply}\ v\ k \\
\text{求值}\ \mathrm{Nil}\ e\ k & \mapsto & \mathrm{Apply}\ V\mathrm{Nil}\ k \\
\text{求值}\ (\mathrm{Cons}\ c_1\ c_2)\ e\ k & \mapsto & \text{求值}\ c_1\ e\ (\mathrm{Cons}\ _\ c2\ e\ k) \\
\mathrm{Apply}\ v_1\ (\mathrm{Cons}\ _\ c_2\ e\ k) & \mapsto & \text{求值}\ c_2\ e\ (\mathrm{Cons}\ v_1\ _\ k) \\
\mathrm{Apply}\ v_2\ (\mathrm{Cons}\ v_1\ _\ k) & \mapsto & \mathrm{Apply}\ (\mathrm{Cons}\ v_1\ v_2)\ k \\
\text{求值}\ (\text{匹配}\ c_1 \cdots)\ e\ k & \mapsto & \text{求值}\ c_1\ e\ (\text{匹配}\ _ \cdots\ e\ k) \\
\mathrm{Apply}\ \mathrm{Nil}\ (\text{匹配}\ _\ [\mathrm{Nil} \to c1] \cdots\ e\ k) & \mapsto & \text{求值}\ c1\ e\ k \\
\mathrm{Apply}\ (\mathrm{Cons}\ v_1\ v_2)\ (\text{匹配}\ _ \cdots\ [\mathrm{Cons}\ x_1\ x_2 \to c_2]\ e\ k) & \mapsto & \text{求值}\ c_2\ [x_1 := v_1;\ x_2 := v_2; e]\ k
\end{array}$$

没有规则用于将一个值应用于 OK 续延，也没有规则用于在空环境中求值 x；前者表示程序成功终止，后者表示错误状态。

2.2 重写规则

更一般地说，Chordata 的输入是一种编程语言定义，由一个包含构造子的 sortConstructor 和一组定义其归约语义的重写规则给出。对于 CEK 机器，构造子集合必须包括控制、值、环境、续延和状态的定义中包含的每个构造子，而重写规则集合则恰好是上面显示的集合。我们

[2]求值 x 的规则似乎依赖于 x 和 y 之间的不等式；然而，在实际程序中，变量只有有限数量，因此这两条规则可以展开为有限数量的纯重写规则。

定义一种项语言，其中每一项都是将一个构造子应用于一串（可能为空的）参数：

$$\begin{array}{rcll} k & \in & \texttt{Constructor} & \\ t & \in & \texttt{Term} & ::= \quad k\ \overline{t} \end{array}$$

这些项上的重写规则由模式定义。模式是一些项，其中部分子项被替换为模式变量：

$$\begin{array}{rcll} x & \in & \texttt{PatternVar} & \\ p & \in & \texttt{Pattern} & ::= \quad k\ \overline{p} \mid x \end{array}$$

我们假设模式变量集合 PatternVar 是无限的。具体而言，我们假设存在一个从 $\mathbb{N}$ 到 PatternVar 的单射，以及 PatternVar 与 PatternVar+PatternVar 之间的一个双射（其中 + 表示不交并）。这些操作用于生成新鲜变量。因此，归约规则由两个模式组成，即右侧和左侧，记作 $p \mapsto p$:

$$r \in \texttt{Rule} ::= p \mapsto p$$

我们还假设每条规则都是函数性的，也就是说，出现在其右侧（RHS）的每个变量也都出现在其左侧（LHS）。Chordata 的输入是一种构造子 Constructor，以及一组规则 $R \subseteq$ Rule。这些规则通过单条推理规则定义语言的语义

$$\frac{r = p_1 \mapsto p_2 \qquad \sigma[p_1] = t_1 \qquad \sigma[p_2] = t_2}{t_1 \overset{r}{\mapsto} t_2} \tag{1}$$

$t_1 \overset{r}{\mapsto} t_2$ 读作“t_1 通过 r步进到 t_2 ”。该推理规则指出，如果存在一个代换 σ，将其应用于归约规则的两侧后可以分别得到这两个项，那么一个项就可以依据某个归约规则步进到另一个项。请注意，为了使第二个和第三个前提有意义，我们将项视为特殊的模式：其中不含模式变量。

不依据任何规则步进到其他项的项 t_1 被视为最终的，或终止的程序状态。如果 $t_1 \overset{r}{\mapsto} t_2$ 和 $r \in R$，我们记作 $t_1 \overset{R}{\mapsto} t_2$，并将 $\overset{R}{\mapsto}$ 定义为 $\overset{R}{\mapsto}$的自反传递闭包。我们称 t_1 在 R 下求值为 t_2（记作 $t_1 \overset{R}{\Rightarrow} t_2$），当且仅当 $t_1 \overset{R^*}{\mapsto} t_2$，并且对每个 t_3都有 $t_2 \not\overset{R}{\mapsto} t_3$。

公式（1）中的假设涉及代换，下面我们将对其进行定义。

2.3 代换与合一

代换定义为从模式变量到模式的映射。在本节余下部分中，我们将项视为不含模式变量的模式，并一般性地讨论模式，而非专门讨论项。对模式应用代换（记作 $\sigma[p]$）时，将每个模式变量替换为与之对应的模式。

$$\begin{array}{rcll} \sigma & \in & \texttt{Substitution} & := \texttt{PatternVar} \to \texttt{Pattern} \\ & & \sigma[k\ \overline{p}] & = k\ \sigma[\overline{p}] \\ & & \sigma[x] & = \sigma(x) \\ & & \sigma[p\overline{p}] & = \sigma[p]\sigma[\overline{p}] \\ & & \sigma[\cdot] & = \cdot \end{array}$$

现在，我们为一对模式定义合一子和最一般合一子。请注意，我们考虑的是用于合一模式对的替换对，而不是用单个替换来合一某组约束这一更为标准的概念。

定义 *2.1*。 $\sigma_1 \circ \sigma_2$ 定义为：对于所有 x，$(\sigma_1 \circ \sigma_2)(x) = \sigma_1[\sigma_2(x)]$。

定义 *2.2*。 $\sigma_1 \sqsubseteq \sigma_2$ 当且仅当 $\sigma_1 = \sigma_3 \circ \sigma_2$ 对于某个 σ_3。

定义 *2.3*。 (σ_1, σ_2) 合一 一对模式 (p_1, p_2) 当且仅当 $\sigma_1[p_1] = \sigma_2[p_2]$。

定义 *2.4*。 (σ_1, σ_2) 是一对模式 (p_1, p_2) 的最一般合一子，当且仅当 (σ_1, σ_2) 合一 (p_1, p_2)，并且对于所有能合一 (p_1, p_2) 的 (σ'_1, σ'_2)，$\sigma'_1 \sqsubseteq \sigma_1$ 且 $\sigma'_2 \sqsubseteq \sigma_2$。

最一般合一子的概念对于定义规则组合至关重要。

2.4 快捷方式

现在，代换和合一已经定义，我们可以引入快捷方式的概念，它允许将多个归约步骤合并为一个。考虑由两个归约步骤组成的序列：

$$t_1 \overset{r_1}{\mapsto} t_2 \overset{r_2}{\mapsto} t_3$$

我们希望得到一条新的“快捷方式”归约规则 r，使得 $t_1 \overset{r}{\mapsto} t_3$。当然，平凡规则$r = t_1 \mapsto t_3$也具有这一性质，但它只适用于归约与 t_1相同的项。我们寻求一条更一般的规则，称为 r_1 和 r_2 的组合（记作 $r_1 \circ r_2$）；该规则仅取决于 r_1 和 r_2，而不取决于特定的项 t_1、t_2和 t_3。这条更一般的规则可能适用于许多后续的程序状态，而不仅仅是 t_1，并且通过将两个步骤合并为一个来加快它们的执行。

我们通过以下推理规则定义规则组合。

$$\frac{(\sigma_1, \sigma_2) \text{ is a most general unifier for } (p_2, p_3)}{(p_1 \mapsto p_2) \circ (p_3 \mapsto p_4) = \sigma_1[p_1] \mapsto \sigma_2[p_4]}$$

在这种判断形式中使用等号，理由并不明显，因为 $\sigma_1[p_1] \mapsto \sigma_2[p_4]$可能无法由 $p_1 \mapsto p_2$ 和 $p_3 \mapsto p_4$唯一确定。为了证明这种记法以及将规则组合视为（偏）函数的做法是合理的，我们已经证明规则组合保持规则上的一种等价关系。与本文中的所有定理一样，我们这里只给出该定理证明的概要；完整的机械化证明见随附的 Agda 制品。

定义 *2.5.* 两个规则等价，当且仅当它们在变量重命名后相同。形式化地，$p_1 \mapsto p_2 \equiv p'_1 \mapsto p'_2$ 当且仅当存在 σ_1 和 σ_2，使得 $\sigma_1[p_1] = p'_1$、$\sigma_1[p_2] = p'_2$、$\sigma_2[p'_1] = p_1$，且$\sigma_2[p'_2] = p_2$。

定理 2.6（规则组合的兼容性）。如果 $r_1 \equiv r_{\prime 1}$，$r_2 \equiv r_{\prime 2}$，$r_1 \circ r_2 = r$，且$r_{\prime 1} \circ r_{\prime 2} \equiv r'$，那么 $r \equiv r_{\prime}$。

证明概要。证明定理时，首先将每条规则 r解释为一个从模式到模式的偏函数 $[\![r]\!]$，然后证明这种解释的若干性质：$[\![r_1]\!] = [\![r_2]\!]$当且仅当 $r_1 \equiv r_2$，并且如果 $r_1 \circ r_2 = r$，那么 $[\![r_1]\!] \circ [\![r_2]\!] = [\![r]\!]$。结合这些性质可以证明，由于偏函数组合是唯一的，规则组合在等价意义下也是唯一的。同样的思路可以证明，由于偏函数组合满足结合律，规则组合在等价意义下也满足结合律。□

我们已经证明了规则组合的另外两个对 Chordata 至关重要的性质：相关的规则组合存在，并且可以安全地扩展归约规则集以将其纳入其中。第一个性质是定理2.7，它保证归约序列中连续规则的组合存在。该定理的证明是构造性的，因此该证明还提供了一种计算规则组合的算法。

定理 2.7（快捷存在性）。若 $t_1 \overset{r_1}{\mapsto} t_2 \overset{r_2}{\mapsto} t_3$，则 $r_1 \circ r_2$ 存在。

证明概要。证明的核心是如下引理：如果一对模式（p_1，p_2）具有合一子，那么它就具有最一般合一子。该最一般合一子是通过迭代地构建一对替换（σ_1，σ_2）获得的，任何合一子都必须经由这对替换进行分解，直到这对替换本身也是一个合一子。σ_1和 σ_2 最初都为恒等替换。在每一步中，比较 $\sigma_1[p_1]$ 和 $\sigma_2[p_2]$。如果它们相等，则完成。否则，找到它们不一致的位置。如果该位置在两个模式之一中具有模式变量 x（不失一般性，设为 $\sigma_1[p_1]$），而在另一个中具有构造子项 $k\ p$，那么我们通过扩展 σ_1 来更新替换对，使其包含一个将 x映射到 $k\{x_i\}_{i<n}$的替换，其中 x_i是互不相同的新鲜变量，且 n是 p的长度。每个合一子都必须经由这些新替换进行分解，因为每个合一子都必须将 x 赋值为具有相同构造子 k和相同数量子节点 n的某个模式。此后，合一过程继续迭代。这样会朝着终止取得进展，因为 $\sigma_1[p_1]$或 $\sigma_2[p_2]$中的构造子数量在每一步都会增加，但两者都不能超过原始合一模式中的构造子数量，因为每个合一子都经由（σ_1，σ_2）进行分解。

现在考虑上述形式的 $\sigma_1[p_1]$和 $\sigma_2[p_2]$之间不存在分歧点的情况：即变量与构造子对应的情况。它们不可能在对应位置上因构造子不同或子节点数量不同而产生分歧，因为那样它们就不可能存在合一子。因此，问题已简化为两个模式仅在变量出现处存在分歧的情况。我们使用“已合一变量”的归纳定义概念来构造最一般合一子。如果两个变量 x_1 和 x_2 要么满足 $x_1 = x_2$，要么分别出现在例如 $\sigma_1[p_1]$中与变量 y_1和 y_2对应的位置，且 $\sigma_2[p_2]$和 y_1和y_2 已合一，则称它们已合一。我们的最一般合一子是键值对 (σ_1', σ_2')，其中每个代换都将已合一的变量映射到同一个变量，并将未合一的变量映射到不同变量。换言之，每个代换都从“已合一”关系下的变量等价类中单射地映射出去。定义 σ_1'和 σ_2'的第二个约束是：如果 x出现在 $\sigma_1[p_1]$中与变量 y在 $\sigma_2[p_2]$中对应的位置，那么 $\sigma_1'(x) = \sigma_2'(y)$。证明这是最一般合一子，可由变量合一的归纳定义推出；该定义给出了能够使这些模式合一的变量的最细划分。□

第二个是定理 2.9，它指出，可以在规则集中加入其组成规则的复合，而不影响程序语义。

定义 *2.8*。规则集 R和 R'等价，当且仅当它们对相同输入求值后得到相同输出。形式化地，$R \equiv R'$当且仅当对所有 t_1和 t_2，$t_1 \overset{R}{\Rightarrow} t_2$当且仅当 $t_1 \overset{R'}{\Rightarrow} t_2$。

定理 2.9（快捷规则有效性）。若 r_1，$r_2 \in R$且 $r_1 \circ r_2$存在，则 $R \cup \{r_1 \circ r_2\} \equiv R$。

证明概要。证明的核心是这样一个引理：如果 $r_1 \circ r_2$ 能为从 t_1 到 t_2的一步归约提供依据，那么 t_1也可以通过两步归约到达 t_2：第一步由 r_1 提供依据，第二步由 r_2提供依据。这确保了归约序列中任何出现的 $r_1 \circ r_2$ 都可以替换为仅涉及 R中规则的归约；反过来，如果某个项在 R 中无法归约一步，那么使用 $r_1 \circ r_2$也无法归约一步。□

这些定理共同确保，在归约序列中总是存在快捷方式规则，并且 Chordata 可以在求值时纳入这些规则，而不影响计算结果。现在我们来介绍 Chordata 用于查找和应用这些快捷方式的数据结构和算法。

3 设计与实现

3.1 Chordata语言

尽管我们的快捷方式理论适用于各种编程语言和归约规则集，但我们将 Chordata 语言具体设定为一种带有 Hindley–Milner类型的函数性一阶编程语言。Chordata 的形式语法如下：

$$
\begin{array}{l}
x \in \text{变量} \quad C \in \text{Constructor} \quad D \in \text{数据类型} \quad F \in \text{函数} \quad i \in \text{整数} \quad t \in \text{类型} ::= \text{整数} \\
\mid D\ tv \in \text{值} ::= i \mid C\ vc \in \text{控制} ::= x \mid \text{Let}\ x = c\ \text{In}\ c \mid i \mid c + c \mid C\ c \mid \text{匹配} \\
c\ [C\ x \to c] \mid F\ ce \in \text{环境} ::= x := v k \in \text{续延} ::= \text{OK} \mid \text{Let}\ x = _\ \text{In} \\
c\ k \mid C\ v _ e\ c\ k \mid \text{匹配}\ _\ [C\ x \to c]\ e\ k \mid \text{Add}\ _\ c\ e\ k \mid \text{Add}\ v\ _ \mid F\ v _ e\ c\ kd \in \text{定义} \\
::= D\ x = C\ t \mid F\ x = cs \in \text{状态} ::= \text{求值}\ c\ e\ k \mid \text{Apply}\ v\ k
\end{array}
$$

该语言是第 2.1节所述抽象机语言的推广，允许用户定义新的数据类型和函数。其语义非常直接：

$$
\begin{array}{l}
\text{求值}\ (\text{Let}\ x = c_1\ \text{In}\ c_2)\ e\ k \mapsto \text{求值}\ c_1\ e\ (\text{Let}\ x = _\ \text{In}\ c_2\ e\ k) \\
\text{Apply}\ v\ (\text{Let}\ x = \text{In}\ c_1\ e\ k) \mapsto \text{求值}\ c_1\ [x := v]\ e\ k \text{求值}\ x\ [y := v; e]\ k \mapsto \\
\text{求值}\ x\ e\ k \text{求值}\ x\ [x := v; e]\ k \mapsto \text{Apply}\ v\ k \text{求值} \\
C\ e\ k \mapsto \text{Apply}\ C\ k \text{求值}\ (C\ c_1\ c_2) \\
e\ k \mapsto \text{求值}\ c_1\ e\ (C\ _\ e\ c_2\ k)\ \text{Apply}\ v_2\ (C\ v_1\ _\ e\ c_1\ c_2\ k) \mapsto \text{求值} \\
c_1\ e\ (C\ v_1\ v_2\ _\ e\ c_2\ k)\ \text{Apply}\ v_2\ (C\ v_1\ _\ e\ k) \mapsto \text{Apply}\ (C\ v_1\ v_2)\ k \text{求值}\ (\text{匹配}\ c_1\ [C\ x \to c_2]) \\
e\ k \mapsto \text{求值}\ c_1\ e\ (\text{匹配}\ _\ [C\ x \to c_2]\ e\ k)\ \text{Apply}\ (C_1\ v_1)\ (\text{匹配}\ _\ [C_2 \cdots \\
]\ [C_3\ x_3 \to c_3]\ e\ k) \mapsto \text{Apply}\ (C_1\ v_1)\ (\text{匹配}\ _\ [C_3\ x_3 \to c_3]\ e\ k)\ \text{Apply}\ (C_1\ v_1)\ (\text{匹配} \\
_\ [C_1\ x_1 \to c_1] \cdots\ e\ k) \mapsto \text{求值}\ c_1\ (v_1 := x_1; e)\ k \text{求值} \\
i\ e\ k \mapsto \text{Apply}\ i\ k \text{求值}\ (c_1 + c_2) \\
e\ k \mapsto \text{求值}\ c_1\ e\ (\text{Add}\ _\ c_2\ e\ k)\ \text{Apply}\ i_1\ (\text{Add}\ _\ c_2\ e\ k) \mapsto \text{求} \\
\text{值}\ c_2\ e\ (\text{Add}\ i_1\ _\ k)\ \text{Apply}\ i_2\ (\text{Add}\ i_1\ _\ k) \mapsto \text{Apply}\ (i_1 + i_2)\ k \text{求值} \\
F\ e\ k \mapsto \text{求值}\ F.c\ k \text{求值}\ (F\ c_1\ c_2) \\
e\ k \mapsto \text{求值}\ c_1\ e\ (F\ _\ c_2\ e\ k)\ \text{Apply}\ v_2\ (F\ v_1\ _\ e\ c_1\ c_2\ k) \mapsto \text{求值} \\
c_1\ e\ (F\ v_1\ v_2\ _\ e\ c_2\ k)\ \text{Apply}\ v_2\ (F\ v_1\ _\ e\ k) \mapsto \text{求值}\ (F.c\ v_1\ v_2)\ e\ k
\end{array}
$$

Chordata 的输入是一系列定义，每个定义用于定义一种代数数据类型或一个函数。随后，Chordata 编译器会将这些定义编译为 OCaml 接口

文件，其中导出相应的类型定义，以及针对这些类型优化后的函数定义。这些函数使用快捷方式记忆化执行，因此重复调用可通过缓存获得加速比。

3.2 编译

然而，直接将快捷方式记忆化应用于此语义会带来无法接受的开销，这是由于分支过多（每一步都要检查程序状态的形式），以及变量实例化时需要线性遍历环境。Chordata 编译器采用了多种优化来解决这些性能障碍。

编译后的程序将程序状态表示为一个控制-环境-延续记录，其中 Apply 表示为一种特殊的控制状态，环境中只有一个值。控制由一种程序计数器表示，它是步骤数组中的一个整数索引；每个步骤都是一个从程序状态映射到程序状态的函数。环境表示为一个动态值数组，程序变量通过索引访问该数组；延续则表示为一个栈。这大幅降低了解释执行的开销。

我们还会压缩程序，使每个执行步骤包含多条基本指令。变量查找、let 定义、构造器应用和整数操作的序列会合并为一个步骤，因为这些操作都不会分配延续帧。中间值不再存储为延续，而是被压入环境，并从环境中弹出。只有调用非尾递归函数时，才需要向延续栈压入一个帧。在这些步骤中，未使用的变量会从延续 [35], 中移除，从而减少学习到的快捷方式中的变量数量，并使快捷方式的应用速度更快。Chordata 中的构造子和模式匹配会编译为对应的 OCaml 构造，这些构造比 Chordata 语言归约规则所暗示的线性扫描更高效。这些优化不仅缩短了执行时间，还使每条原始重写规则更大，从而减少了所需的规则操作次数。

因此，我们编译后的表示形式更像栈机或寄存器机，而不是纯粹的 CEK 机器；由于它与实际硬件相似，因此能够实现更高效的执行。

3.3 运行 Chordata

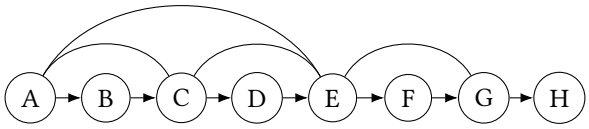
要以这种编译后的表示形式执行函数，Chordata 会设置一个抽象机：控制指向函数的起始位置，环境包含函数参数，延续则初始化为特殊的 Done 值。随后，Chordata 会循环运行，直到抽象机尝试将某个值 Apply 到 Done 延续上，此时 Chordata 就可以从环境中提取结果。在循环中，Chordata 首先会查找适用于当前抽象机状态的已学习快捷方式。如果找到，就用该快捷方式重写抽象机状态；如果没有找到（例如必须执行新的算术运算时），则生成适用的原子重写规则并应用它。这些原子重写是在需要时生成的，用于模拟基于重写的操作之外的运算，例如对机器整数进行算术运算。为生成原子重写，Chordata 会正常执行程序指令，同时将检查过的任何环境槽或延续槽标记为已使用。然后，Chordata 根据初始和最终程序状态构造左侧和右侧模式：对未标记的槽使用变量；对已标记的槽，则用构造子实例化模式，直到遇到未标记的值。

3.4 捷径学习

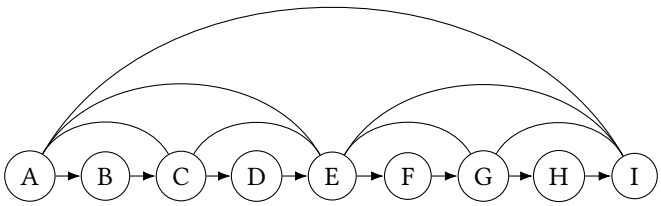
应用原子重写或捷径重写后，Chordata 可以将其与之前的重写组合，以学习新的快捷方式。由于任意连续规则序列都可以组合，因此可以从长度归约序列中推导出 $O(n^2)$

种可能的快捷方式。因此，生成所有可能的快捷方式会比实际运行程序耗时更长。为此，Chordata 会更有选择性地推导并保留快捷方式，并采用各种优化，以紧凑的方式存储快捷方式并快速查找适用的快捷方式。

为避免快捷方式数量激增，Chordata 将快捷方式划分为若干“层级”：图 2a中，同一层级的规则对应的弧长度相同，并且只组合相邻且处于同一层级的规则对。这样，学到的快捷方式数量便限制为随归约序列长度线性增长，同时仍能学到一些非常大的（因而非常有用的）快捷方式。此外，在这组快捷方式中，没有任何快捷方式会跳过另一条快捷方式的起点。这一点至关重要，因为它使快捷方式应用可以采用贪心策略；在当前程序状态应用快捷方式，不太可能越过某个中间状态，而该状态处可能存在更好、更长的快捷方式。



(a) 执行七条重写规则后得到的快捷方式。水平箭头表示原子归约步骤，弯曲的弧线表示通过规则组合得到的快捷方式重写规则。



(b) 从 H 到 I 新增一个归约步骤后的快捷方式，因为此时步骤数是 2 的幂。新增了三条快捷方式：一条从 G 到 I，一条从 E 到 I，最后一条从 A 到 I。

快捷方式树的层级结构使其在整个求值过程中都易于维护：它从空树开始，并随着每个后续归约步骤不断增长。图2b展示了图 2a 中的快捷方式集合如何随着又一个归约步骤的加入而增长。在图中，从 H 到 I 的原子归约步骤与其前一步从 G 到 H 的步骤组合，形成一条从 G 到 I 的快捷方式。由于同一层级上已有一条从 E 到 G 的前置快捷方式，这两条快捷方式便组合成一条从 E 到 I 的快捷方式；这一过程不断重复，最终形成一条层级更高、从 A 到 I 的快捷方式。求值过程中会持续添加快捷方式，并且添加后即可立即使用。这意味着，在求值开始时通过组合规则得到的快捷方式，可以在求值稍后的阶段作为一个单独步骤使用。如果某次程序执行反复在相似数据上执行相似程序（例如递归过程中），那么快捷方式甚至能加速这一次执行。

3.5 判别树

这组学习到的快捷方式必须建立索引，以便高效搜索。更具体地说，在每个执行步骤中，Chordata 都必须找到一条适用于当前程序状态的归约规则。这样的规则可能有多条，此时 Chordata 倾向于应用“长度”最大的规则。

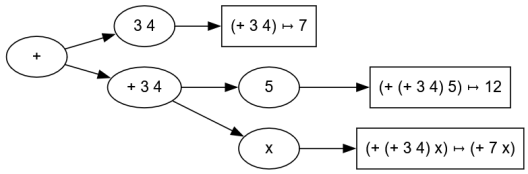


图 3. 包含三条重写规则的判别树。

规则的长度是它跳过的原子归约步骤的数量。该数量与每条规则一同存储，并可通过 $\text{length}(r_1 \circ r_2) = \text{length}(r_1) + \text{length}(r_2)$ 轻松维护。

可用的归约规则集合，包括原子规则和学习到的快捷方式，存储在判别树中，这是一种定理证明器中常用的数据结构。判别树是一种字典树，用于存储所有已知快捷方式或原子步骤的左部（LHS）。由于字典树的键是序列，因此每个左部都会被展平为其先序遍历。此外，由于归约规则查找应在 α 等价（变量重命名）下保持不变，因此归约规则使用 De Bruijn 指数表示。左部中的每个变量都是一个无名绑定符（一种特殊的零元构造子），右部中的每个变量则是一个数字，用于索引其对应的左部绑定符。只有当重写规则是线性的（即左部没有重复变量）时，这种方法才可行，而 CEK 抽象机的规则确实满足这一条件。

在字典树数据结构中，每个键都是一条贯穿整棵树的完整路径，而每个键对应的值都存储在该路径的叶节点上。我们的字典树是一种基数树，也就是说，每个节点存储的是一串元素，而非单个元素。每个节点所存储的序列尽可能长，其长度通过最长公共前缀算法计算得出。因此，在给定节点处，每个子节点的序列都以不同的元素开头，父节点便利用该元素对子节点进行索引。在 Chordata 中，子节点根据数量存储在关联列表或哈希表中。[3]图 3展示了一个示例。

Chordata 将程序状态展平为其先序遍历序列，并在字典树中匹配该序列，从而找到与当前程序状态匹配的最长归约规则。匹配过程从根节点开始遍历字典树，依次将字典树中的节点与展平后的程序状态中的片段进行匹配。当在字典树中遇到变量时，它会与展平后的程序状态中对应于完整子项的前缀匹配。由于程序状态既可以与字典树中的构造器匹配，也可以与树中的变量匹配，因此遍历过程会产生分支，进而到达不同的子节点。分支遍历会到达一组叶节点，并返回其中长度最长的规则。

4 优化

必须进行若干优化，才能确保快捷方式的查找和应用足够迅速，从而真正加快程序执行。

4.1 搜索剪枝

搜索判别树的开销很大，因为可能有许多快捷方式与给定的程序状态匹配；事实上，我们的快捷方式学习策略基本上保证了这一点。为了缩短搜索时间，我们使用分支定界算法。判别树中的每个节点都会存储其子树中最长规则的长度。遍历判别树时，Chordata 会维护搜索过程中目前找到的最长规则长度，并跳过不包含更长最大规则长度的子树

[3]当子节点数量不超过阈值时，使用哈希表所带来的开销并不值得。

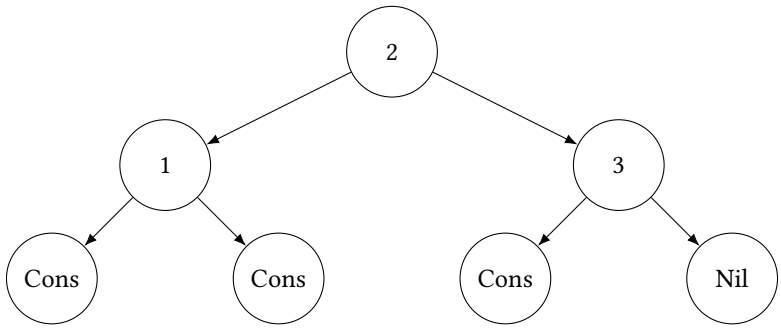


图 4. Chordata 的内部项表示，支持在一次组合操作中高效地包装和拆解多个构造子。在此示例中，将项包装在 *Cons*(0, _) 中会在前面添加两个构造子节点（*Cons* 和 0），而二叉树结构则提供了对数时间复杂度的追加操作。

。此外，为了尽可能多地剪枝，会优先搜索最大规则长度较长的分支。

4.2 快速规则匹配

随着学习到的快捷方式变得越来越复杂，相应的规则也会变得越来越复杂。在项和规则的标准数据表示中，应用规则的成本与左侧和右侧的大小成正比。因此，语法上很大的快捷方式可能无法比直接计算带来有意义的加速比。为此，Chordata 使用另一种重写规则编码方式，使模式匹配的开销主要取决于变量的数量。

回顾一下，判别树存储着学习到的快捷方式的左侧前缀遍历结果。为了高效搜索，程序状态也被存储为字符串，即其前序遍历结果；我们将得到的字符串称为“程序状态字符串”。在判别树中搜索程序状态字符串时，只需检查程序状态是否以固定字符串开头，即可匹配常量字符串；并在程序状态字符串中查找完整子项，即可匹配变量。为了加速这两个步骤，程序状态字符串存储在增强型平衡二叉树中。[4] [5] 在这棵二叉树中，子串匹配和完整项匹配都需要 $O(\log n)$时间，因此应用含有 k个变量的规则需要 $O(k \log n)$时间。这些增强结构通过幺半群解析和幺半群哈希，加速字符串和子项匹配。应用规则还涉及构造新节点，这需要为每个快捷方式的右侧拼接子项和常量字符串。平衡二叉树也支持 $O(\log n)$时间复杂度的拼接操作，因此应用规则同样很快，其时间复杂度取决于变量数量，而与规则大小无关。[6] 向判别树中插入新规则的速度也得到了提升。

幺半群解析。 幺半群解析可以快速找到程序状态字符串中从给定索引开始的完整子项。在幺半群解析中，弧度为 a 的每个构造器都会被赋予一个度$1 - a$。这个量表示该构造器对序列中子项总数的贡献：它为自身构造的子项贡献 $+1$ ，并为其所需的 a 个 $-a$ 子节点贡献相应数量。例如，每个零元构造子度为 1，并贡献其自身的子项，

[4]判别树中的常量字符串采用相同的平衡二叉树表示，以加快更新判别树时所需的常见前缀检查。[5]该树以优化后的指状树实现，通常能提供更优的常数因子和算法复杂度上界。[6]最终，每个变量都绑定到程序状态字符串中的一个子串，该子串包含 $O(\log n)$ 棵子树，因此应用规则需要拼接 $O(k \log\ n)$ 棵子树，耗时为 $O(k \log n \log \log\ n)$。重要的不是精确的渐近复杂度，而是其不受规则大小影响。

而每个三元构造器的度为 −2，并将三个子项合并为一个。于是，S 中从给定索引 i 开始的完整子项，是满足以下条件的最短 j：对 $S[i$ 中的度求和：$j]$，结果为 1，这恰好对应一个子项。随后可以利用二叉树结构高效地找到 j。每个二叉树节点都扩充了两个值：它所存储序列中的度之和（总_度），以及该序列所有前缀中最大的总_度（最大_度）。序列的每个前缀都对应二叉树中的一片子树森林。可以利用这片森林中每棵树根节点的总_度和最大_度，计算整个前缀的最大_度。由于序列前缀中的最大_度不会递减，因此可以通过二分查找找到最大_度为 = 1的最短前缀。

幺半群哈希。幺半群哈希函数是定义在序列上的哈希函数 h，它具有一个作用于哈希值的拼接运算符 $\oplus$，满足 $h(x{+\!\!+}y) = h(x) \oplus h(y)$。这意味着，可以通过拼接各元素的哈希值来构建序列的哈希值。二叉树中的每个节点都存储其内容的长度和幺半群哈希值。要匹配常量字符串，首先利用长度信息将常量字符串缩减为二叉树节点 $O(\log n)$，再对这些节点进行幺半群哈希并与常量字符串的幺半群哈希值进行比较。

Chordata 使用 Castagnoli CRC (CRC32C) 作为其幺半群哈希，该哈希通过 x86 SSE4.2 实现。幺半群哈希的拼接运算通过为每个节点同时存储校验和 p 以及依赖于长度的因子 $x^n \bmod P$来实现。组合结果的校验和只需结合两个校验和与依赖于长度的因子即可得到。我们的实现利用了硬件支持：使用硬件 CRC 指令逐字节折叠，并使用无进位乘法（x86_64 上的 PCLMULQDQ、AArch64 上的 PMULL）进行幺半群哈希组合。

4.3 实例化

由于匹配和应用快捷方式的成本现在由模式变量的数量决定，Chordata 倾向于学习变量较少的快捷方式。为此，系统将快捷方式中的模式变量数量限制为最多十个，分别适用于延续和环境中的每个条目。如果合一导致组合规则中的模式变量超过十个，则会用这些额外变量在当前程序状态中的值来实例化它们。具体而言，在延续和环境中的每个条目里，都会保留最右侧的十个变量，其余变量则进行实例化。之所以优先保留最右侧的十个变量，是因为许多常见的数据结构（如列表）右侧不平衡，而且函数应用（通常涉及常函数）位于左侧。

5 求值

为了评估 Chordata，我们设计了两个基准测试，以检验系统的不同方面：RQ1. Chordata 能否在程序发生变化时加速求值？RQ2. Chordata 能否通过重复计算模式加速求值?

5.1 RQ1：Chordata 能否加速程序更改时的求值?

Hazel 是一种纯函数式、按值调用的编程语言，通过其“洞”结构支持实时编程。Hazel 系统允许用户编辑程序，并在每次按键后重新求值部分程序，在用户输入时实时显示结果。我们在 Chordata 中实现了一个（无类型的）Hazel 解释器，并用它求值由 Hazel 捕获的执行轨迹，这些轨迹记录了三名学生完成函数式编程练习的过程。每条执行轨迹都包含每次按键后的完整程序；我们像 Hazel 一样执行轨迹中的所有程序，同时保留学习到的快捷方式。我们忽略重复程序，否则 Chordata 可借此大幅加快执行速度

。我们比较了 Chordata 与其一个变体的求值时间，后者既不学习也不应用快捷方式。

该基准测试由八个规模和复杂度不断增加的程序组成。我们要求参与者根据规格说明实现这些程序。这些程序包括：

- 基本列表函数（映射、过滤、追加）。
- pair：以列表作为输入，并返回输入列表中相邻元素组成的键值对列表，这需要使用非结构递归。
- reverse：通过 append 辅助函数实现。
- 快速排序、归并排序和插入排序的函数。

24 条基准测试轨迹共包含 4615 个记录的程序。每项实验都在全新的缓存中运行，使用 Ubuntu 24.04.4 系统，配备 Intel i7-8700K CPU 和 32 GiB 内存。

所有执行轨迹的整体加速比为 13.03×。这一加速效果显著受到所用输入列表规模的影响。本节的所有结果均是在含 400 个元素的输入列表上运行程序得到的；由于 Chordata 能够为重复计算提供渐近加速，列表越长，加速效果就越显著。我们认为，Chordata 即使面对只有几百个元素的较短列表，也能提取出加速效果，这是它的一项优势。

图 5a 以散点图展示各项求值时间，图 5b 则展示基准测试中每个数据点的加速比累积分布函数。图 5a 中每个点的横坐标表示未使用学习到的快捷方式时的求值时间，纵坐标表示使用学习到的快捷方式时的求值时间。粗对角线表示 $y = x$等时曲线；曲线下方的点表示增量求值更快的情况。大多数点都位于曲线下方，且往往相差显著——请注意，图表使用对数刻度——这表明增量求值通常能够带来显著的加速比。

散点图中包含许多垂直带，以及一条水平带。垂直带源于编辑前后计算基本不变的情况：基线耗时近乎恒定，但随着系统在多次执行中学习快捷方式，增量计算会变得更加高效。水平带则源于重复计算，在这种情况下，记忆化使增量耗时不受程序变化的影响。

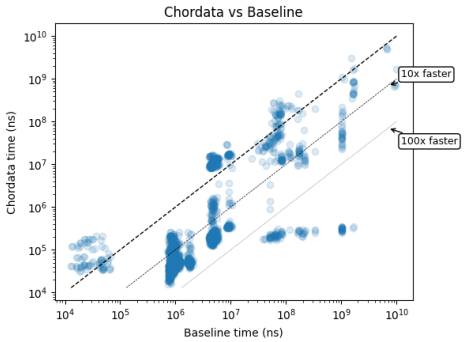
图 5b 展示了数据的汇总视图，纯粹聚焦于相对于基线的加速比。图中有三条显著的带：一条对应约 2× 的减速，影响约 6% 的程序；另一条对应约 25× 的加速，影响约 70% 的程序。就减速而言，这些情况主要涉及递归计算：由于输入发生变化，Chordata 会花时间记录未来迭代中不会用到的快捷方式。最大的减速约为 8×。在这些情况下，用户对程序所做的编辑要求进行全新的计算，而这些计算此前从未执行过，例如首次调用某个函数。此时，Chordata 需要付出前期成本：花时间学习新的快捷方式，而这些快捷方式要到执行轨迹中后续的程序才会用到。这个过程的开销导致了这些点上的减速。幸运的是，这些减速大多出现在执行轨迹的开头，此时程序尚未完成，运行速度较快。

每个解释函数的净加速比记录在表 1a中。由于编码风格不同，不同用户获得的加速效果也有所差异。例如，用户在先实现基本情况还是归纳情况，以及主函数和辅助函数的排列顺序上各不相同。卡姆的两个[7] 程序包含无限循环，因此无法为其分配加速因子。[8] 我们还测量了 Chordata 各组件所占的比例时间，

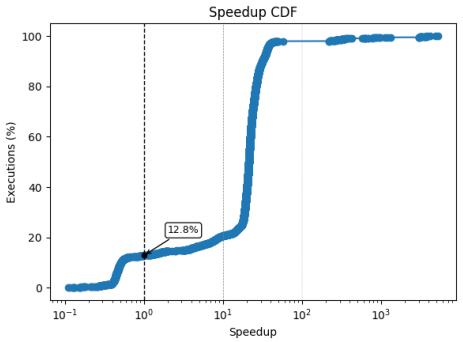
[7]这不是真实名称[8]快捷方式的一个有趣副作用是，它们可以检测出某些无限循环：如果快捷方式的左侧 $\mapsto$ 右侧中，左侧与右侧匹配，就能保证发生非终止。

记录在表 1b中。规则查找和规则组合耗时最多，因为前者需要分支判断，而后者会执行复杂的合一逻辑。这些结果证明，用于加速查找的优化是合理的。

表 1a 还显示了每个函数的内存开销。该开销相当高，有时甚至达到 100×。更复杂的函数具有更高的开销，因为求值过程更长，因此学习到的快捷方式也更多。数值较高的主要原因是 Chordata 目前尚未实现缓存淘汰策略。与任何缓存一样，判别树中的规则具有截然不同的命中次数，我们预计，即使是简单的缓存淘汰策略，也能显著降低内存开销。



(a) 增量求值时间与基线求值时间的散点图。



(b) 每个状态加速比的累积分布函数。

图5。实时编程基准测试的结果。

基准测试	爱丽丝	Bob	Cam	爱丽丝	Bob	Cam
	时间加速比			内存开销		
追加	18.2	19.43	22.43	6.931	12.16	14.88
过滤	2.558	11.49	11.32	12.73	8.693	10.29
Map	12.64	4.564	7.209	5.674	9.07	7.654
快速排序	15.14	8.868	8.91	48.2	37.1	73.98
插入排序	13.27	18.56	16.99	58.58	28.85	33.39
归并排序	5.079	15.7	X	121.1	186.9	X
Pair	8.231	12.7	X	9.399	11.03	X
reverse	4.775	12.83	21.45	29.95	17.17	8.007

(a) 基准测试的加速比和内存开销 rk.

Name	百分比
组合规则	28.97%
查找规则	28.88%
实例化	18.36%
插入规则	12.87%
Apply 规则	10.07%
misc	0.86%

(b) 按组件划分的时间分解。

表 1。实时编程基准测试的详细结果。

5.2 RQ2: C Chordata 能否通过重复计算模式加速求值?

Hazel 的求值涵盖了我们预期的 Chordata 使用场景：实时编程中，相似的程序会被反复求值，且延迟至关重要。然而，在求值过程中，我们注意到，Chordata 有时能在空缓存的情况下加速对单个程序的求值。这是因为程序本身可能会产生重复计算，例如

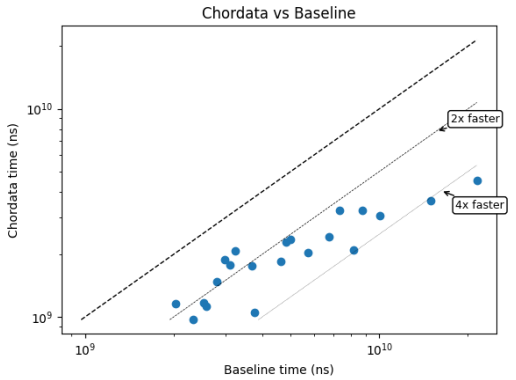


图 6. 代数化简基准测试的散点图

通过将输入转换为重复性更高的形式。（例如，归并排序会生成许多大小相近的已排序列表！）为了清晰地展示这一现象，我们又开展了一项案例研究，使用符号微分器和代数化简器对一个数学表达式求两次导数，然后化简结果。化简过程依据加法和乘法各自的恒等律约简运算，将同一加法或乘法序列中的常数合并，并提取加法序列中的公因子。这个过程会反复进行，直到达到不动点。符号微分经常会产生结构重复的相似子项（空间上的重复），借助快捷方式，可以加速对这些子项的微分和化简。与此同时，化简的迭代特性会在各次迭代之间形成重复结构（时间上的重复）。由于每次化简只会改变表达式的一部分，借助前一次迭代中学到的快捷方式，处理表达式中未改变的部分可以加快。

利用这两种重复模式，Chordata 能够在大小为 100 到 200 个节点的随机生成输入表达式上获得 2.464× 加速比。为了最大限度地消除共享结构，每次运行之间都会清空缓存。尽管这一加速比远小于我们在 Hazel 基准测试中获得的结果，但仍然显著：Hazel 基准测试使用共享缓存对许多相似程序进行了求值，而本实验不会在不同实验之间复用缓存。尽管将 Chordata 应用于通用计算很可能会造成无法接受的内存开销，但我们很期待它能够利用输入数据中的重复、程序执行中的重复，以及编辑之间的重复。此外，尽管加速比不高，但这是在计入快捷方式查找开销的情况下取得的；对于更大或重复性更强的表达式，加速效果还会更显著。例如，执行重复求导的应用（如泰勒展开器）会产生更多共享结构，从而取得更大的加速效果。

6 相关工作

自调整计算。 自调整计算 [4] 和 Adapton [12] 是两种重要的增量计算系统，它们会记录计算图中各个值之间的依赖关系。随后，当计算的输入发生变化时，它们会使用图算法重新计算图中的最小子集。然而，这类系统难以处理计算图的变化。一些后续系统采用混合方法，Acar 等人 [3], 依靠记忆化与计算图之间的紧密协作来维护身份并登记变化；例如，“nominal Adopton” [11] 使用用户指定的 名称 作为计算图中的稳定标识符。

此外，计算图的设计往往会限制所支持的增量计算类型。例如，自调整计算无法自然地支持跨越重新排序或共享的子计算进行增量化，因为它使用顺序维护数据结构，为计算维护全局线性顺序。在 Adapton 中，脏节点与重新计算值之间的阶段分离也会带来类似的限制。这种设计可能导致不必要的重新计算，而更具需求驱动性的求值策略则可以避免这些计算。

增量λ演算。 其他工作，如增量λ演算， [6], 补丁协调 [45], 和差分数据流 [17] 沿依赖图传播程序值的变化。例如，在差分数据流中， [17] 程序值是多重集，变化则是多重集元素的添加和删除。增量λ演算 [6] 为跟踪任意λ演算程序中的此类变化提供了统一框架。然而，与计算图方法类似，它只响应变化，因此也面临远距离相似性问题。

记忆化。 记忆化通过记录函数的输入/输出键值对，并在输入与相应的先前输入匹配时重用先前输出，从而跳过计算，提升函数执行速度。记忆化的一个问题是其通用性有限：记录记忆化计算的所有先前输入，并不能解决新值只有细微差异的情况。快捷方式记忆化是一种记忆化，但它通过学习可泛化的规则，而非简单的输入输出键值对，提升了传统记忆化的性能和通用性。依赖跟踪方案 [1, 14] 也通过识别记忆化函数所不依赖的输入部分，改进了记忆化。从某种意义上说，快捷方式记忆化对合一的运用正是这类依赖跟踪方案。

记忆化的另一个问题，尤其是在递归函数中，是朴素的记忆化必须遍历输入的主干以定位并处理更改，这可能会很慢。函数缓存 [28] 等方法引入了“块状分解”算法，将任意列表转换为平衡二叉树，使相似列表共享树结构。这缩短了主干长度，并提升了记忆化性能。Chordata 使用类似的思路加快其判别树中的查找速度，并通过幺半群哈希和解析将这一方法推广到列表以外的数据。

部分求值。 部分求值和分阶段是编译技术，它们将函数输入划分为编译时已知和运行时已知的部分。随后，函数会利用编译时部分尽可能地求值，从而生成一个用于求值运行时输入的残余程序。Sundaresh [36] 通过构建一种可将任意程序增量化的算法，探究部分求值与增量计算之间的关系；用户可以指定哪些输入在编译时固定，哪些输入可在运行时变化。Chordata 与部分求值有一些相似之处：快捷左部模式同时包含固定输入和模式变量。一个关键区别在于，典型的部分求值器采用“推送”方式，而记忆化通常采用“拉取”方式。也就是说，在部分求值中，指定一些输入后，函数会在编译时尽可能地化简，并生成一个供单独运行时阶段使用的残余程序。而在 Chordata 中，快捷方式会被学习和组合，输入依赖则通过合一在运行时发现。

增量程序分析。增量技术为程序分析带来了巨大的机遇，并具有广泛的潜在应用。其中一个重要应用是与集成开发环境（集成开发环境）集成。在这类环境中，每次程序修改后都必须及时更新分析结果，以便提醒用户当前程序中可能存在的问题。

关于增量程序分析的既有研究，已在多个截然不同的分析框架中展开，包括不动点迭代 [5, 22, 25, 32], 基于 Datalog 的分析 [37, 38, 42], 以及

代数程序分析 [43]。然而，由于这些方法依赖不同的底层分析框架，所得到的技术往往具有异构性，并且与其所构建的特定框架紧密耦合。除了框架的异构性之外，增量的来源也各不相同：(1)。输入程序可能发生变化，而集成开发环境可能需要及时响应的分析结果，因此可以将输入增量化。(2)。在程序分析中常见的不动点计算过程中，可以记录发生了哪些变化，并仅传播这些变化，而不是重新计算所有内容 [29]。(3)。Zhu 和 Calman 等作者[44] 指出，程序中的不同变量可能共享许多分析结果，而这种相似性可以通过二元决策图等方法加以利用 [20]。Chordata 并不专门关注程序分析，但可能适用于同一类问题。

Hazel。 Hazel 有许多子系统，它们都面临由交互性和实时性带来的类似延迟约束，因此可以通过增量化来提升性能。这些子系统包括类型检查 [27] 和语法错误纠正 [19], , Chordata 或许也适用于这些子系统，以提升增量性。近期关于 Hazel 中增量式类型检查的研究采用了基于依赖图的方法 [27], , 但在以我们观察到的方式编辑程序时，沿依赖图传播，同时也将结果存入缓存并从中读取的混合方法，或许会更加有效。

简洁数据结构。 简洁数据结构能够紧凑地表示重复数据；它们通常将输入拆分为多个块，将计算结果存入查找表，并重复利用这些结果，这与 Chordata 有些相似。此外，Chordata 用于快速规则匹配和应用的数据表示借鉴了简洁表示的思想，尤其是将带有度信息的先序遍历编码为字符串的简洁树。Gawrychowski 等人 [9], 、Sadakane [33], 以及 He 和 Munro [13] 等作者都研究过类似的字符串表示。例如， [9] 通过直线文法对一族字符串进行编码。直线文法是一种无环上下文无关文法，其中每个非终结符都只有一条产生式规则，因此每个非终结符恰好表示一个字符串。这种编码能够压缩并去重相同的子串，从而降低 Chordata 的总内存消耗，同时消除对哈希的概率性依赖。我们希望在 Chordata 中进一步探索简洁数据结构的思想。

项索引。 判别树 [16]属于一类用于自动定理证明和逻辑编程的数据结构与算法。这类通用的项索引技术用于高效存储并检索适用于某个项的重写规则；其他广泛使用的技术还包括替换树 [10], 上下文树 [8], 和路径索引 [16]。据我们所知，Chordata 中结合使用判别树、幺半群解析和幺半群哈希的方法尚属首创，也可能适用于其他项索引场景。

7 结论与未来工作

快捷方式记忆化是一种增量计算方法，通过组合并应用重写规则来加速重复执行。我们提出的规则组合方法能够生成语义上有效且最通用的快捷方式，并已在 Agda 中形式化，包括对所有定理的证明。随后，我们介绍了 Chordata，这是一种纯函数式编程语言，它会编译为高效的抽象机表示；该表示能够通过观察执行来学习快捷方式，将这些快捷方式存储在紧凑的判别树中，并将它们应用于未来相似的程序状态。我们通过自定义的项表示来加速这棵判别树，该表示结合了幺半群解析和幺半群哈希。评估结果表明，在实时编程场景的一组程序轨迹上，Chordata 实现了 13.03× 的加速比。此外，在一项符号

微分案例研究中，Chordata 能够在单次执行中利用快捷方式，凭借重复的程序结构实现 2.464× 的加速比。

在未来的工作中，我们希望通过允许除归约规则之外还可指定方程，丰富 Chordata 的理论。例如，可以将加法交换律纳入规则匹配过程，从而增加适用快捷方式的情形。这或许可以借助等式图 [21]。我们还可以通过试验简洁数据结构和项索引数据结构来改进 Chordata 的数据结构，从而获得更大的加速效果。最后，我们计划添加缓存淘汰策略并试验持久化缓存，以减少内存占用，并在多次执行中获得更大的加速效果。

数据可用性声明

我们将提交制品以供制品评估。我们的制品包含主要定理的 Agda 形式化、Chordata 的实现以及评估案例研究。执行该制品后，评估人员应能看到 Agda 对我们的定理进行了形式化验证，并生成一个包含评估图表的网页（尽管由于执行环境不同，数值会略有差异）。

参考文献

[1] Martín Abadi、Butler Lampson 和 Jean-Jacques Lévy。1996 年。依赖关系的分析与缓存。SIGPLAN Not.（1996 年 6 月），31 卷第 6 期，83–91 页。doi:10.1145/232629。[2] Robin Abraham、Margaret Burnett 和 Martin Erwig。2009 年。电子表格编程。doi:10.1002/9780470050118。ecse415[3] Umut A. Acar、Guy E. Blelloch 和 Robert Harper。2003 年。自适应记忆化。https://api.semanticscholar.org/ CorpusID:[4] Umut A. Acar、Guy E. Blelloch 和 Robert Harper。2006 年。自适应函数式编程。ACM Trans. Program. Lang. Syst. 28, 6 2006 990 1034 10 1145 1186632 1186634（2006 年 11 月），第 28 卷，6 期，– 页。doi:–。Steven Arzt 和 Eric Bodden。–。Reviser：针对增量程序变更，高效更新基于 IDE/IFDS 的数据流分析。载于第 36 届国际软件工程会议论文集。288–298 页。[6] Yufei Cai、Paolo G. Giarrusso、Tillmann Rendel 和 Klaus Ostermann。2013 年。高阶语言的变更理论——通过静态微分实现 λ 演算的增量化。arXiv:1312.0658 [cs.PL] https://arxiv.org/abs/ 1312.0658[7] Matthias Felleisen 和 Daniel P. Friedman。1987 年。控制运算符、SECD 机器与 λ λ演算。载于《编程概念的形式化描述》。https://api.semanticscholar.org/CorpusID:[8] Harald Ganzinger、Robert Nieuwenhuis 和 Pilar Nivela。–。使用编码上下文树进行快速项索引。《自动推理杂志》32 卷第 2 期（2004 年 2 月），103–120 页。doi:10.1023/B:JARS.0000029963.64213.ac[9] Paweł Gawrychowski、Adam Karczmarz、Tomasz Kociumaka、Jakub Lacki 和 Piotr Sankowski。2015 年。最优动态字符串。载于 ACM-SIAM 离散算法研讨会。https://api.semanticscholar.org/CorpusID: [10] 1995 Peter Graf。–。替换树索引。载于国际重写技术与应用会议。https://api.semanticscholar.org/CorpusID: [11] Matthew A. Hammer、Jana Dunfield、Kyle Headley、Nicholas Labich、Jeffrey S. Foster、Michael Hicks 和 David Van Horn。2015 50, 10 2015 748 766 10 1145 2858965 2814305。带名称的增量计算。[12] SIGPLAN Not.（10 月），– 页。doi:–。Matthew A. Hammer、Khoo Yit Phang、Michael Hicks 和 Jeffrey S. Foster。–。Adapton：可组合的按需增量计算。载于第 35 届 ACM SIGPLAN 编程语言设计与实现会议论文集（爱丁堡，英国）（PLDI '14）。计算机协会，纽约州纽约市，美国，156–166 页。doi:10.1145/2594291.2594324[13] Meng He 和 J. Ian Munro。2010 年。动态字符串的简洁表示。ArXiv abs/1005.4652（2010）。https: //api.semanticscholar.org/CorpusID: [14] 2000 Allan Heydon、Roy Levin 和 Yuan Yu。–。使用精确依赖关系缓存函数调用。载于 ACM SIGPLAN 2000 年编程语言设计与实现会议论文集（温哥华，不列颠哥伦比亚，加拿大）（PLDI ' 00 311 320 10 1145 349299 349341）。计算机协会，纽约州纽约市，美国，– 页。doi:–。Marisa Kirisame、Tiezhi Wang 和 Pavel Panchekha。–。用于布局失效的无脊柱遍历。ACM《编程语言学报》 9, 219 2025 23 10 1145 3729322 编程语言设计与实现，文章（6 月），页码。doi:–。William McCune。–。用于项检索的判别树索引和路径索引实验。《自动推理杂志》9 卷第 2 期（1992 年 10 月），147–167 页。doi:10.1007/BF00245458

[17] Frank McSherry、Derek Gordon Murray、Rebecca Isaacs 和 Michael Isard。2013 年。差分数据流。载于创新数据系统研究会议。https://api.semanticscholar.org/CorpusID:18593675

[18] Andrey Mokhov、Neil Mitchell 和 Simon Peyton Jones。2018 年。模块化构建系统。ACM《编程语言学报》2 卷，ICFP，文章 79（2018 年 29 月），页码。doi:10.1145/3236774。

[19] David Moon、Andrew Blinn、Thomas J. Porter 和 Cyrus Omar。2025 年。带有实质义务的句法补全。ACM《编程语言学报》9，OOPSLA，文章 404（2025 年 2 月），27 页。doi:10.1145/3763182。

[20] Mayur Naik、Alex Aiken 和 John Whaley。2006 年。Java 的有效静态竞态检测。载于 ACM SIGPLAN 编程语言设计与实现会议。ACM，308–319 页。doi:10.1145/1133981.1134018。

[21] Greg Nelson 和 Derek C. Oppen。1980 年。基于同余闭包的快速判定过程。《ACM 汇刊》27 卷第 2 期（4 月 1980），356–364 页。doi:10.1145/322186.322198。

[22] Aman Nougrahiya 和 V Krishna Nandivada。2025 年。IncIDFA：一种高效通用的增量迭代数据流分析算法。ACM《编程语言学报》9 卷，OOPSLA1（2025 年），617–648 页。

[23] Cyrus Omar、Ian Voysey、Ravi Chugh 和 Matthew A. Hammer。2019 年。使用带类型空洞的实时函数式编程。ACM《编程语言学报》3，POPL，文章（1 月 14 日），32 页。2019。doi:10.1145/3290327。

[24] Fernando Perez 和 Brian E. Granger。2007 年。IPython：交互式科学计算系统。《计算机与科学和工程》9, 3（2007 年 21 月），29 页。doi:10.1109/MCSE.2007.53。

[25] Lori L. Pollock 和 Mary Lou Soffa。1989 年。迭代数据流分析的增量版本。《IEEE 软件工程汇刊》15, 12（1989 年 12 月），1537–1549 页。doi:10.1109/32.58766。

[26] Thomas J. Porter、Marisa Kirisame、Ivan Wei、Pavel Panchekha 和 Cyrus Omar。2025 年。通过顺序维护实现增量双向类型推断。ACM《编程语言学报》9，OOPSLA，文章 339（2025 年 2 月），28 页。doi:10.1145/3763117。

[27] Thomas J. Porter、Marisa Kirisame、Ivan Wei、Pavel Panchekha 和 Cyrus Omar。2025 年。通过顺序维护实现增量双向类型推断。ACM《编程语言学报》9，OOPSLA，文章 339（2025 年 2 月），28 页。doi:10.1145/3763117。

[28] W. Pugh 和 T. Teitelbaum。1989 年。通过函数缓存实现增量计算。载于第 16 届 ACM SIGPLAN-SIGACT 编程语言原理研讨会论文集（美国得克萨斯州奥斯汀）（POPL '89）。计算机协会，纽约州纽约市，美国，315–328 页。doi:10.1145/75277.75305。

[29] Benjamin Quiring 和 David Van Horn。2024 年。通过导数推导：优化高阶流分析的增量不动点。ACM《编程语言学报》8 卷（2024 年），728–755 页。https://api.semanticscholar.org/CorpusID:[redacted]。

[30] Russell W. Quong 和 Mark A. Linton。1991 年。增量式链接程序。ACM Trans. Program. Lang. Syst. 13 卷第 1 期（1991 年 1 月 1991 年 1 月 20 日），10.1145/114005.102804。doi:10.1145/114005.102804。

[31] Patrick Rein、Stefan Ramson、Tom Beckmann 和 Robert Hirschfeld。2025 年。活跃性的信息觅食解释。载于 2025 年 IEEE 可视化语言与以人为中心的计算研讨会（VL/HCC）。128–138 页。doi:10.1109/VL-HCC65237.2025.00022。

[32] Barbara G Ryder。1983 年。增量数据流分析。载于第 10 届 ACM SIGACT-SIGPLAN 编程语言原理研讨会论文集。167–176 页。

[33] Kunihiko Sadakane。2007 年。用于灵活文本检索系统的简洁数据结构。《离散算法杂志》5 卷（2007 年），12 月 22 日，15923596。doi:[redacted]。https://api.semanticscholar.org/CorpusID:

[34] Patrick W. Sathyanathan、Wenlei He 和 Ten H. Tzen。2017 年。增量整程序优化与编译。载于 2017 年国际代码生成与优化研讨会论文集（美国得克萨斯州奥斯汀）（CGO '17）。IEEE 出版社，221–232 页。

[35] Zhong Shao 和 Andrew W. Appel。2000 年。高效且节省空间的闭包转换。ACM Trans. Program. Lang. Syst. 22, 1（2000 年 1 月），129–161 页。doi:10.1145/345099.345125。

[36] R. S. Sundaresh。1992 年。通过部分求值实现增量计算。https://api.semanticscholar.org/CorpusID:209401075。

[37] Tamás Szabó、Sebastian Erdweg 和 Gábor Bergmann。2021 年。使用格在 Datalog 中进行增量整程序分析。载于第 42 届 ACM SIGPLAN 编程语言设计与实现国际会议论文集。1–15 页。

[38] Tamás Szabó、Sebastian Erdweg 和 Markus Voelter。2016 年。Inca：用于定义增量程序分析的领域专用语言。载于第 31 届 IEEE/ACM 自动化软件工程国际会议论文集。320–331 页。

[39] Steven L. Tanimoto。2013 年。关于实时编程演进的观点。载于第 1 届实时编程国际研讨会论文集（美国加利福尼亚州旧金山）（LIVE '13）。IEEE 出版社，31–34 页。

[40] Vercel。2026年。增量静态再生。https://vercel.com/docs/incremental-static-regeneration。Vercel 文档。最后更新于2026年1月27日。

[41] Tim A. Wagner 和 Susan L. Graham。1998 年。高效灵活的增量解析。ACM Trans. Program. Lang. Syst. 20, 5（1998 年 9 月），980–1013 页。doi:10.1145/293677.293678。

[42] David Zhao、Pavle Subotic、Mukund Raghothaman 和 Bernhard Scholz。2021 年。迈向 Datalog 的弹性增量化。载于国际声明式编程原理与实践研讨会。ACM，20:1–20:16。

doi:10.1145/3479394.3479415

[43] Chenyu Zhou、Yuzhou Fang、Jingbo Wang 和 Chao Wang。2025 年。一种用于代数程序分析的增量算法。*ACM*《编程语言学报》9，POPL（2025），1934–1961。[44] Jianwen Zhu 和 Silvian Calman。2004 年。重新审视符号指针分析。载于*ACM-SIGPLAN* 编程语言设计与实现研讨会。https://api.semanticscholar.org/CorpusID:5905331

[45] 2025 Parker Ziegler、Justin Lubin 和 Sarah E. Chasins。. 借助补丁协调对应关系实现快速直接操纵式编程。*ACM*《编程语言学报》9，PLDI，文章 175（2025 年 6 月），26 页。doi:10.1145/3729278