DeCo：一种支持泛型数据类型的增量函数式编程核心演算

TIMON BÖHLER，德国达姆施塔特工业大学TOBIAS REINHARD，德国达姆施塔特工业大学
DAVID RICHTER，德国达姆施塔特工业大学MIRA MEZINI，德国达姆施塔特工业大学、德国hessian.AI，以及德国国家应用网络安全研究中心 ATHENE

arXiv:2602.20866v1 [cs.PL] 24 Feb 2026

增量化通过避免不必要的重新计算并高效复用先前结果来加快计算。领域专用技术能够显著提升性能，例如在数据库查询场景中，但难以推广。与此同时，通用方法几乎无法为领域专用运算的增量化提供支持。

在这项工作中，我们提出了 DeCo——一种新颖的核心演算，支持使用多种用户定义数据类型进行增量函数式编程。尽管我们的方法具有通用性，却能对用户定义数据类型上的领域专用运算进行静态增量化。因此，与其他将领域专用运算视为黑盒的通用技术相比，我们的方法粒度更细。

我们在 Lean 中对工作进行了机械化，并证明了其正确性，即增量化执行所得结果与完全重新求值相同。我们还提供了可执行实现，并通过案例研究展示了线性代数、关系代数、字典、树和无冲突复制数据类型中的示例，此外还简要评估了线性代数、关系代数和树方面的性能。

1 引言

增量化 [20] 通过避免完全重计算、仅处理输入变化来加速计算。我们提出DeCo，这是一种用于增量编程的带类型函数式一阶语言，支持静态、细粒度的增量化，并结合用户定义数据类型与内置和类型和积类型。

DeCo 通过以下方式解决现有技术的局限并对其加以补充：不同于选择性重新求值等粗粒度方法 [15]将运算视为黑盒，DeCo能够对用户定义数据类型上的运算进行增量化，同时仅缓存相关的中间结果。不同于针对多维数组等特定结构的细粒度方法 [29] 或关系 [5]，DeCo 通过容器 [1]为不同数据类型的增量计算提供统一基础。与数据类型无关的方法，例如 IλC [7] 在实现新数据类型时支持有限，因为它们没有通用的内置运算；而DeCo 提供数据类型泛化操作，例如映射 和重塑 ，以及增量化组合子，使增量化更容易推导。

对不同数据类型的统一处理。 为了有效地增量化用户定义数据结构上的运算，DeCo 必须利用特定于数据结构的属性，同时支持种类繁多的数据结构。我们的演算通过将类型构造子表示为容器 [1] 并要求基类型构成变化结构 [7]来实现这一点。容器提供了一种统一表示数据结构的方式，即将其表示为从索引到值的映射，从而能够简单地进行原地更新——与代数数据类型不同，后者需要递归遍历结构中未变化的部分。变化结构提供值类型、变化类型，以及将

作者联系信息：Timon Böhler，达姆施塔特工业大学，德国达姆施塔特，timon.boehler@tu-darmstadt.de；TobiasReinhard，达姆施塔特工业大学，德国达姆施塔特，tobias.reinhard1@tu-darmstadt.de；David Richter，达姆施塔特工业大学，德国达姆施塔特，david.richter@tu-darmstadt.de；MiraMezini，达姆施塔特工业大学，德国达姆施塔特；hessian.AI，德国达姆施塔特；ATHENE国家应用网络安全研究中心，德国达姆施塔特，mezini@informatik.tu- darmstadt.de。

变更应用于值的运算。用户可以通过定义 (a) 其数据结构的容器，(b) 其基类型的变化结构，以及 (c) 领域专用操作及其增量化，将 DeCo 实例化到自己的领域中。DeCo 还内置了积类型和和类型，可以自动为基类型、和类型、积类型及容器的任意组合派生变化结构。

增量化组合子。 领域专用操作的增量化通常遵循某些模式，而且可能会重复。然而，细微的错误就可能使整个计算失效。DeCo 通过提供一组遵循这些模式、由构造保证正确的增量化组合子，减轻用户负担并增强对用户的保障。这些组合子使用户能够利用函数的线性或自可维护性。最重要的是，它们保证可靠性：通过我们的组合子构建的任何领域专用增量化，都会产生与原始计算相同的结果。

机械化可靠性证明。我们在依赖类型编程语言 Lean 中对 [11]DeCo 进行机械化。具体而言，我们对演算的指称语义和增量化进行了机械化。最重要的是，我们以机械化方式证明，增量化相对于形式化的指称语义是可靠的。也就是说，将增量程序应用于一系列变更，所得结果与重新计算原始程序相同。此外，我们还证明，只要满足每个组合子的前置条件，使用我们的增量化组合子构建的每种增量化都由构造保证正确。

实现与案例研究。 基于对我们演算的机械化，我们实现了一个泛型增量化框架，并将其应用于源自线性代数、关系代数、字典、树和无冲突复制数据类型的增量操作。由此，我们展示了如何利用 DeCo 的泛型数据变换和增量化组合子，对多个领域中的运算进行增量化。此外，我们还对若干程序进行了增量化，展示了与完全重新求值相比，增量化能够带来预期的加速效果：一个稠密神经网络层、两个涉及连接与聚合的关系查询，以及对玫瑰树求和。

贡献。 总而言之，我们提出了 DeCo，这是一种用于增量函数式编程的新型核心演算，具有以下关键特性：(i) 支持内置的和类型与积类型，以及多种用户提供的数据结构。(ii) 支持通过泛型操作和组合子，便于将其实例化到新的应用领域。(iii) 自动增量化，且正确性由构造保证。具体而言，我们作出以下贡献。

- **核心演算：** 一种用于带缓存的增量函数式编程的最小形式化基础，支持泛型数据变换，可实例化为不同的数据类型。
- **增量化组合子：** 可复用的构建模块，能够简化用户定义运算的构造即正确的增量化定义。
- **机械化可靠性证明：** 增量化能够保持程序语义。该证明已在 Lean 中完全机械化。
- **实现：** 一个实现我们形式化核心演算的库。
- **案例研究：** 我们将该库实例化，用于实现涉及树、线性代数与关系代数，以及一个示例性无冲突复制数据类型（CRDT）的增量运算。

论文概述。本文其余部分的组织结构如下。首先，在第2节中，我们介绍缓存增量化及其应满足的正确性属性。在第3节中，我们以稠密层程序为主要示例，解释我们方法的不同组成部分。随后，我们将正式定义DeCo、其类型系统、指称语义和增量化，以及

第4节中的增量化组合子。第5节概述了这些概念在 Lean 中的机械化与验证。第6节和第7节分别介绍库实现及其在我们案例研究中的应用。[1]

2 缓存增量化简述

本节介绍增量化、其缓存变体，以及正确增量化的概念。增量化旨在通过复用先前计算的结果，加速对相似数据的重复计算。我们采用组合式增量化方法，通过将运算映射到增量运算，将常规程序转换为增量程序。

一种朴素的增量化方法如下：考虑类型 A 和 B，并设A' 和 B' 分别为变更在 A 和 B上的类型（在我们的设定中，可以设想 A' 包含稀疏数据结构，其中的非零条目描述了如何更新 A中各值的条目）。我们用 $x \oplus x'$ 表示将变更 $x' \in A'$ 应用于值 $x \in A$。那么， $f\colon A \to B$ 的一种增量化就是函数 $f'\colon A' \to B'$，它将输入变化映射为输出变化：

$$f\ (x \oplus x') = f\ x \oplus f'\ x'. \tag{1}$$

请注意，在本文中，我们使用上标 $'$表示与变更相关的变量，正如上文中的 A', x', f' 一样。满足上述定律的增量化 f' 使我们能够在求值 $f(x \oplus x')$ 时复用先前计算出的值 $f\ x$。为了计算后者，我们只需将差集 $f'\ x'$ 加到 $f\ x$上。请注意，上述公式假设输出变化仅取决于输入变化，因为 f' 只接收 x'，而不接收 x。然而，我们通常会依赖更多信息来计算变化。最直接的解决方案是也将 x传递给 f'。但这样一来， f' 可能需要重新求值 f的部分内容，以确定中间结果发生了哪些变化。为了避免这种重新求值， f' 应接收一个包含中间结果的数据结构。这一见解构成了增量化的缓存 方法 [14]的核心。

一般而言，缓存增量化通过为程序中的每个运算配备缓存来记住中间值。因此，为了对任意函数进行增量化， $f\colon\ A \to B$我们需要定义三个组成部分（图1）：(i) 一个缓存类型 C： 用于描述缓存哪些值的类型。(ii) 一个差分函数 $d\colon\ A' \to C \to B' \times C$ 它接受输入变化和先前的缓存，并返回输出变化和新的缓存。(iii) 一个初始化函数 $i\colon\ A \to B \times C$它会同时生成函数的实际结果和初始缓存。之后，便可在此初始缓存上调用差分函数 d。

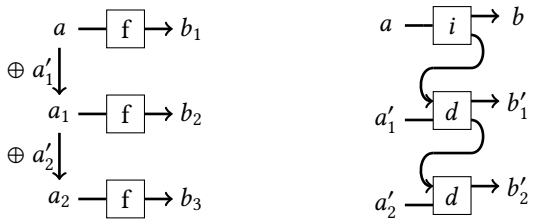
图2 展示了这种方法。考虑一个值 a，它通过变更 a'_1、a'_2、.....反复修改；函数 f以 a 为输入。非增量方法（图1a）每次都必须重新求值 f，每当 a 发生变化时都是如此。相比之下，增量方法（图1b）只会对 f进行一次完全求值：在第一步中对其求值。之后的每一步，我们都会在缓存和输入变化d 上调用增量函数，而不是使用完整输入 a'_i（ a_i）。这样便能避免完全重新求值。

注意， d 也可以视为米利机的转移函数，其输入类型为 A'，输出类型为 B'，内部状态为 C。米利机实现因果流函数 [13] 这一事实意味着，我们也可以将增量化解释为类型为 Stream $A' \to$ Stream B'的函数，即将输入变化流映射为输出变化流。这个视角将 Giarrusso 等人 [14] 的基于缓存的增量化方法与 Budiu等人 [5]的基于流的方法联系起来。我们采用前者，因为更新缓存的思路更接近实际实现。相比之下，因果流函数需要接收先前的全部输入变化

[1]制品的初步版本可在https://figshare.com/s/abdad549da026b4dbca5获取。

$$
\begin{array}{lcl}
f: & & A \to B \\
\\
C & : & \mathsf{Type} \\
i & : & A \to B \times C \\
d & : & A' \to C \to B' \times C
\end{array}
$$

图 1. 增量化接口。



(a) 完全重新求值。 (b) 缓存增量求值。

图 2. 概览。

历史记录才能生成当前的输出变化，这显然无法直接用于实现。此外，米利机明确指出了缓存中存储的是哪些数据。

我们的缓存增量化必须遵循以下定律（另见定义 5.2）：（我们用 $i_1\ x$ 表示 $i\ x$的第一个分量，i_2、d_1和 d_2也同理。）

$$
\begin{array}{lll}
\forall x. & i_1\ x = f\ x & \text{(Law-1)} \\
\forall x\ x'. & f\ (x \oplus x') = f\ x \oplus d_1\ x'\ (i_2\ x) & \text{(Law-2)} \\
\forall x\ x'. & i_2\ (x \oplus x') = d_2\ x'\ (i_2\ x). & \text{(Law-3)}
\end{array}
$$

这些定律的直观含义如下：（定律-1）初始化函数的第一个分量等于原始的完整求值。（定律-2）使我们能够避免对变更后的值 $f(x \oplus x')$进行完全重计算。利用前一步的缓存，我们可以使用差分函数 d 来计算变更。（定律-3）将差分函数 d 应用于变更 x' 和由 i 生成的初始缓存（即 $i_2\ x$），可以一致地更新缓存。具体而言，为更新后的输入 $x \oplus x'$ 初始化缓存，所得结果与使用差分函数 d更新现有缓存的结果相同。

这些定律用于证明我们的关键可靠性性质。在第 5 节的可靠性证明中，我们证明我们的增量化会计算出与完整重新求值相同的结果。更准确地说，给定程序 f、初始值 x和变更列表 $x'_{(\circ)}$，以下两种方式得到的结果相同：（i）将变更 $x'_{(\circ)}$ 应用于 x，并对新值调用 f（即完整重新求值）。（ii）计算$i\ x$ 并将 d 应用于每个变更（即增量化）。这一证明确保 DeCo 程序支持正确性由构造保证的增量化，使其能够用作基础演算和中间表示（参见第6.2节）。

定律 1 – 定律 3 刻画了确保这种可靠性的性质。它们本质上描述了这样一个不变量：每次变更后，缓存都必须与重新求值所得到的当前输出保持一致。为说明这一思想，考虑一个初始输入 a 以及一系列输入变化 $a'_1, \ldots, a'_n$ 将 a转换为 a_n。依次对每个变化应用 d，即可得到相应的一系列输出变化 $b'_1, \ldots, b'_n$（参见图 2）。然后，我们将这些变化依次应用于初始输出 $f\ a$，从而得到更新后的输出 $f\ a_n$。

由于采用内禀静态类型，我们只需考虑类型正确且终止的程序；与无类型方法相比，这使得正确性证明更加简单 [14]。上述定律可通过相对直接的归纳证明得出。

$$
\begin{array}{lcl}
\mathsf{dup} & : & A \rightsquigarrow A \times A \\
\mathsf{cst} & : & [\![A]\!] \to (B \rightsquigarrow A) \\
\mathsf{map} & : & (A \rightsquigarrow B) \to (\mathsf{F}_s\ A \rightsquigarrow \mathsf{F}_s\ B) \\
\mathsf{zip} & : & \mathsf{F}_s\ A \times \mathsf{F}_s\ B \rightsquigarrow \mathsf{F}_s\ (A \times B) \\
\mathsf{map2} & : & (A \times B \rightsquigarrow C) \to (\mathsf{F}_s\ A \times \mathsf{F}_s\ B \rightsquigarrow \mathsf{F}_s\ C) \\
\mathsf{get} & : & [\![s]\!]_I \to (\mathsf{F}_s\ A \rightsquigarrow A) \\
\mathsf{set} & : & [\![s]\!]_I \to (A \times \mathsf{F}_s\ A \rightsquigarrow \mathsf{F}_s\ A) \\
\mathsf{reshape} & : & ([\![s_2]\!]_I \to [\![s_1]\!]_I) \to (\mathsf{F}_{s_1}\ A \rightsquigarrow \mathsf{G}_{s_2}\ A) \\
\mathsf{replicate} & : & (s : S) \to A \rightsquigarrow \mathsf{F}_s\ A \\
\mathsf{tp} & : & \mathsf{F}_{s_1}\ (\mathsf{F}_{s_2}\ A) \rightsquigarrow \mathsf{F}_{s_2}\ (\mathsf{F}_{s_1}\ A) \\
\mathsf{filter} & : & ([\![s]\!]_I \to \mathbb{B}) \to (A \times \mathsf{F}_s\ A \rightsquigarrow \mathsf{F}_s\ A)
\end{array}
$$

图 3. 泛型运算。

$$
\begin{array}{lcl}
\beta & := & \mathbb{R} \\
(\cdot \oplus \cdot) & := & (\cdot +_{\mathbb{R}} \cdot) \\
(\cdot \ominus \cdot) & := & (\cdot -_{\mathbb{R}} \cdot) \\
\mathsf{S} & := & \mathbb{N} \\
[\![n]\!]_I & := & \{0, \ldots, n-1\} \\
[\![\mathsf{relu}]\!]\ x & := & \text{if } x < 0 \text{ then } 0 \text{ else } x \\
[\![*]\!]\ (x, y) & := & x \cdot y \\
[\![\mathsf{sum}]\!]\ x & := & \Sigma_i (x\ i)
\end{array}
$$

F 图 4. 线性代数实例化（未进行增量化）。

3 DeCo 程序示例

本节将（i）讨论我们的核心演算DeCo的特性，（ii）通过为线性代数实现一个领域特定实例化来说明对领域特定实例化的支持，以及（iii）讨论缓存增量化对该实例化的领域专用运算所提出的要求。稍后的第7节中，我们将展示更多实例化示例，这些实例化实现了关系代数和无冲突复制数据类型的增量化。

在DeCo 程序中，我们区分以下几类：(i) 提供适用于所有数据结构的泛型基本操作的核心演算。(ii) 与特定数据结构相关联，并在实例化时提供的领域-专用操作。(iii)派生操作由用户通过核心操作和领域专用操作定义，而DeCo会自动为其推导增量化。[2]

容器。 DeCo 将数据结构表示为容器 [1]。容器由一组形状以及每种形状对应的一组索引定义。形式化地说：

$$
\begin{array}{lcl}
S & : & \mathsf{Type} \\
[\![\cdot]\!]_I & : & S \to \mathsf{Type}
\end{array}
$$

例如，数组定义为一种容器，其形状为自然数（表示数组的长度），而形状 n的索引为 0、……、$n-1$。因此，对于数组，我们有$[\![n]\!]_I := \{0、\ldots\ldots、\ n-1\}$。我们语言的语义是在任意此类容器上定义的。

泛型操作。 DeCo 包含多种泛型操作（见图 3 中的一部分，其余操作见下一节），我们认为这些操作对增量函数式编程很有用。不过，我们并不声称它们是完备的；在第 7 节中，我们将展示这些操作具有足够的表达能力，可以实现线性代数、关系代数和无冲突复制数据类型中的有趣领域专用操作。这些泛型操作始终可用，与所使用的数据结构无关。请注意，$A \rightsquigarrow B$ 表示 DeCo中的一个增量函数，而 $A \to B$表示元语言（在我们的例子中为 Lean）中的任意函数。此外，$[\![A]\!]$ 是对象级类型 A的（元语言）指称，而 $\mathsf{F}_s\ A$ 是一个形状为 s、元素来自 A的容器。在线性代数中，这意味着 $\mathsf{F}_s\ A$ 是一个长度为 s的数组。

我们提供具有通常语义的二阶函数映射、map2和zip 。函数复制 会复制一个值，而cst x 是一个常量函数。我们允许读取和写入单个元素

[2]按照 Patri <style 恒等='1'>gnani [27], 我们分别通过颜色以及无衬线体、粗体和斜体来区分这三种类型。

通过 获取 和 设置。运算 重塑 r 使用函数 r 变换索引，从而执行形状变换。其输出实际上可以是与输入不同的容器类型（此处记作 G）。例如，我们可以通过定义 n 将索引 r 映射到 i，以实现反转长度为$n - i - 1$的列表。接下来，复制 会生成一种数据结构，其中每个条目都包含相同的值。函数 转置通过交换外层和内层类型构造子来转置嵌套数据结构。最后，筛选 接受一个返回布尔值的函数 p，该函数作用于索引；如果 p将某个索引映射为 false，则会将对应条目设为默认值。（下一节将详细介绍这些运算符的指称语义，参见图 9。）DeCo 还提供用于组合程序以及处理积和和的运算。暂且略过这些运算，稍后将在第 4 节讨论。

我们根据泛型容器表示来定义这些运算。这使得 DeCo 具有良好的可扩展性和复用性：只要将一种数据结构定义为容器，核心运算（及其增量化形式）就会自动可用。我们展示了许多数据类型都可以编码为容器，包括矩阵、关系以及 CRDT 的状态向量。因此，当用户希望将某个新的应用领域增量化时，只要证明其底层数据类型是容器，就能立即使用多个已经增量化的运算。 DeCo 用户只需为基本的领域专用操作（例如乘法）提供增量化形式，随后便可使用泛型操作构造更复杂的增量程序。这样既减少了用户的实现工作量，也使领域特定实例的定义十分简洁，我们在第 7 节的案例研究中对此进行了展示。

为线性代数实例化 *DeCo*。 DeCo 支持对核心演算进行模块化的领域特定实例化，从而便于添加对特定应用领域的支持。只要领域专用操作具有正确的增量化形式，所有使用这些操作的 DeCo 程序就都能保证具有正确的增量化形式。此外，DeCo 的增量化组合子有助于领域专用操作的开发者提供正确的增量化形式。下面，我们通过线性代数的示例，非正式地介绍实例化机制，概要见图 4。这意味着需要定义一种基类型、一种数据结构和一组运算。

首先，我们定义其他类型所构建于其上的基类型 β。线性代数实例化的基类型定义为 $\beta := \mathbb{R}$。我们还需要定义变更的集合 β' 以及将变更应用于值的更新操作 $\oplus$。在本例中，$\beta' := \mathbb{R}$，更新操作为加法。我们还定义了一个减法运算，用于计算两个值之间的差集。上述数据决定了一个变化结构 [7],，第4节将对此作进一步解释。接下来，我们定义该实例化的域数据结构为数组。为了复用核心演算中的数据类型泛化操作（图3），这里通过将数组描述为容器来给出这一定义。为此，我们将数组的形状类型定义为 $S := \mathbb{N}$，对应于数组的长度。如上所述，我们将位置类型定义为 $[\![n]\!]_I := \{0, \ldots, n-1\}$，这意味着长度为 n 的数组具有从 0 到 $n-1$ 的有效索引。以这种方式定义数组意味着，长度为 n 的向量表示为 $\mathrm{F}_n\,\beta$，而 $n \times m$ 矩阵表示为 $\mathrm{F}_n\,(\mathrm{F}_m\,\beta)$。

对于线性代数实例化，我们只添加三个原语：修正线性单元（$\mathbf{relu} : \beta \to \beta$）、数乘（$* : \beta \times \beta \to \beta$）和向量求和（$\mathbf{sum} : \mathrm{F}_s\,\beta \to \beta$）。图4中写出了每个运算的语义 o，其写法为 $[\![o]\!]$（例如 $[\![\mathbf{relu}]\!]$）。结合泛型操作，这三个领域专用原语足以定义许多更复杂的线性代数运算。例如，使用泛型操作和数乘，我们可以将矩阵-向量乘法构造为一个派生运算。用数学符号表示，矩阵-向量乘法定义如下：

$$(\mathsf{mvmul}\ M\ v)[i] = \textstyle\sum_j (M[i,j] \cdot v[j])$$

从操作语义上看，我们将每个元素 x 与 v 中的一列配对 M，将 x 与该列中的每个元素相乘，然后将结果求和。在DeCo中，写法如下（这里我们使用map2 f同时遍历两个数组）：

$$mvmul := (\mathsf{id} \times \mathsf{replicate})\ ;\ \mathsf{map2}\ (\mathsf{map2}\ *)\ ;\ \mathsf{map}\ \mathbf{sum}.$$

程序 *mvmul* 以二元组 (M, v) 作为输入，并计算出 $M\,v$。它分三步完成：首先，将复制 应用于 v，通过复制每个元素将其转换为矩阵 v'。然后，对 M 和 v'执行逐元素乘法。最后，对结果按行求和。

我们的 *mvmul* 是一种派生操作，它通过核心演算和线性代数专用实例化中的运算来定义。具体来说，运算 **mul** 和 $*$ 是线性代数原语（定义见图 4）。用于定义 *mvmul* 的大多数构造都来自核心演算，包括恒等函数 id、函数复合运算符 $(f; g)$、并行运算符 $(f \times g)$，它对输入 返回 $(f\ x, g\ y)$，以及二阶运算符 map2。

举例来说，请比较作用于向量的全连接层的数学描述（左）与使用 DeCo 实现的版本（右）：

$$M : \mathsf{F}_m\ (\mathsf{F}_n\ \beta) \qquad b : \mathsf{F}_m\ \beta \qquad \mathsf{dense} : \mathsf{F}_n\ \beta \to \mathsf{F}_m\ \beta$$

$$\mathsf{dense}\ x := \mathsf{map}(\mathsf{relu}, M \cdot x + b) \qquad dense := \begin{array}{l} \langle \mathsf{cst}\ M, \mathsf{id} \rangle ; mvmul ; \\ \langle \mathsf{cst}\ b, \mathsf{id} \rangle ; \mathsf{map2}\ + ; \\ \mathsf{map}\ \mathbf{relu} \end{array}$$

函数 *dense* 使用我们推导出的 *mvmul*，即线性代数原语 **relu**、二阶运算符 映射和常函数组合子 cst（我们还使用了语法糖：$\langle f,\ g \rangle := (\text{复制};\ f \times g)$，它会先复制同一个值，再将两个函数分别应用于该值）。

简单来说，dense 由以下操作组合实现：将矩阵 M与 M函数参数相乘，将 p 加到乘法结果上（使用 map2），然后对每个元素应用 **relu**。

线性代数实例化的缓存增量化。 线性代数实例化的最后一步，是为领域专用操作定义缓存增量化。我们考虑的变更是逐元素修改，即将实数（可以是零或负数）加到值的每个条目上。因此，对于向量，变更就是另一个逐元素相加的向量。

要对 **relu** 和 $*$进行增量化，我们需要缓存两个输入值。在这两种情况下，我们都通过用新输出减去旧输出来计算导数。例如，**relu**的导数为

$$\lambda(x', x).\ (\llbracket \mathbf{relu} \rrbracket\ (x \oplus x') \ominus \llbracket \mathbf{relu} \rrbracket\ x,\ x \oplus x'),$$

其中，x' 表示变化量，x 表示旧输入，$x + x'$ 表示新输入，而 $\llbracket \mathbf{relu} \rrbracket\ (x + x')$ 表示新输出。导数接收变化量和旧输入，并返回新输出与旧输出之间的差集。

注意，对于诸如 ReLU 或乘法本身这样的标量运算，增量化并不能带来性能收益。相反，由于缓存，它会增加内存需求。然而，定义它们的增量化使我们能够应用增量化组合式 地通过组合子表达式的缓存。例如，当将映射 应用于 $*$时，我们借助向量形状的缓存实现逐元素的向量乘法。在这种情况下，乘法的缓存增量化最终实现了渐近加速：如果某一个输入元素发生变更，我们只需更新输出向量中的一个元素，而重新求值则必须遍历整个向量。更具体地说，增量版本的 映射 接受一个变更并映射

作用于其上的 $*$ 的增量版本。由于该变更仅包含一个元素，$*$ 的增量版本只被调用一次。

我们明确写出了 ReLU 的增量化过程，不过也可以使用增量化组合子给出更简洁的定义。我们将在第4.5节中更详细地介绍这些组合子。直观来说，增量化组合子将增量化中的常见模式具体化，从而简化实现。对于函数 f（根据组合子的不同，它可能需要满足某些性质），增量化组合子会返回一个元组(f_c, f_i, f_d)，其中包含缓存类型、初始化和导数。当增量化组合子的前置条件得到满足时，已形式化证明它会输出正确的增量化。

如上所述，relu 和 $*$ 的增量化都通过缓存输入来定义，并通过从新输出中减去旧输出来计算导数。我们用Triv 组合子来概括这种模式；它会创建一种执行重新求值的“平凡”增量化。另一个增量化组合子是Lin，它概括了对线性函数进行增量化的模式，因为线性函数的导数就是其自身。线性函数的一个例子是sum 运算符。因此，增量化可以根据运算符的常规求值语义来定义（我们用 $[\![f]\!]'$ 表示元组(f_c, f_i, f_d)）：

$$[\![\mathbf{ReLU}]\!]' := \textsc{Triv}\ [\![\mathbf{ReLU}]\!] \qquad [\![*]\!]' := \textsc{Triv}\ [\![*]\!] \qquad [\![\mathbf{sum}]\!]' := \textsc{Lin}\ [\![\mathbf{sum}]\!]$$

4 增量化的形式化

本节将正式介绍我们的方法。首先，我们将描述 DeCo 演算的语法和指称语义，并给出用于将 DeCo 实例化为新数据结构和运算的接口。添加新的运算还要求我们定义它们的增量化。我们提出了一组小型组合子，以简化构造即正确的增量化的实现。随后，我们将说明如何通过对 DeCo 项定义增量化变换，使我们的方法能够自动且可靠地对计算进行增量化。

4.1 接口

如前所述，DeCo 的参数取决于容器和基类型的选择。

图 5 展示了 DeCo 的参数化接口。图的上半部分展示了 n元容器，它们表示带有 n 个参数的类型构造子。在此语境下，S 表示容器的形状类型，而 $[\![\cdot]\!]_I$ 接受一个形状，并为每个 n 类型参数给出一个索引类型。为确保证明具有构造性，我们要求形状具有可判定相等性（我们的证明确实依赖命题外延性和函数外延性）。

为便于理解，我们对本文其余部分作两项简化。首先，我们的机械化支持多个容器和基类型。使用多个容器确实可以对不同容器的嵌套进行建模，例如，关系的元素包含数组，就像向量数据库中的情况一样。在本文中，我们只展示单个容器和基类型的情形。其次，我们的机械化支持具有多个类型参数的容器（n元容器）。在本文中，我们只介绍具有单个类型参数的一元容器，因为除转置[3] 之外，所有运算从一元情形推广到 n元情形的方式都显而易见，因此本文仅介绍一元情形。关于多个容器、多个基类型以及 n元容器的详细信息，可参见制品中的文件`Correctness.lean`。

我们要求基类型构成一种变化结构 [7], ，即一种结合了值类型、变化类型以及将变更应用于值的运算的结构。我们用 β 表示基值类型，并用 β'表示其对应的变化类型。我们要求一个更新运算 $\oplus$ 和相应的差分操作 $\ominus$，使得 $x \oplus (y \ominus x) = x$。更新运算允许我们

[3]转置仅适用于一元容器。

容器S ： 类型$[\![\cdot]\!]_I$ ： $S \to$类型 deceq : $(s : S) \to$ 可判定相等性 s

变化结构 β ：类型β' ： 类型 $(\cdot \oplus \cdot) : \beta \to \beta' \to \beta$ $(\cdot \ominus \cdot) : \beta \to \beta \to \beta'$ 完整 : $x \oplus (y \ominus x) = x$

图 5. 容器与变化结构接口。

$s : \mathsf{S}\ r : [\![s_2]\!]_I \to [\![s_1]\!]_I$ $c :$ $\beta i :$ $[\![s]\!]_I$ o : Op $A\ B$ $p :$ $[\![s]\!]_I \to \mathbb{B}$类型$A, B ::=$ b | $\mathsf{F}_s\ A$ | $A \times B$ | $A + B$表达式$e, f ::=$ e_1 ; e_2 | $e_1 \times e_2$ | 恒等 | 复制 | 首分量 | 次分量| 映射 e | zip | 获取 i | 设置 i | 重塑 r| 复制 s | 转置 | 筛选 p | fuse | distr| inl | inr | $e_1 \,||\, e_2$ | cst c | + | op o

图 6. 语法。

$$\frac{\vdash e_1 : A \rightsquigarrow B \qquad \vdash e_2 : B \rightsquigarrow C}{\vdash e_1 \,;\, e_2 : A \rightsquigarrow C} \qquad \frac{\vdash e_1 : A \rightsquigarrow B \qquad \vdash e_2 : C \rightsquigarrow D}{\vdash e_1 \times e_2 : A \times C \rightsquigarrow B \times D} \qquad \frac{\mathbf{o} : \mathrm{Op}\ A\ B}{\vdash \mathrm{op}\ \mathbf{o} : A \rightsquigarrow B}$$

$$\frac{\vdash e : A \rightsquigarrow B}{\vdash \mathsf{map}\ e : \mathsf{F}_s\ A \rightsquigarrow \mathsf{F}_s\ B} \qquad \frac{\vdash e_1 : A \rightsquigarrow B \qquad \vdash e_2 : C \rightsquigarrow D}{\vdash e_1 \,||\, e_2 : A + C \rightsquigarrow B + D}$$

$$\vdash \text{恒等} : A \rightsquigarrow A \quad \vdash \text{首分量} : A \times B \rightsquigarrow A \quad \vdash \text{次分量} : A \times B \rightsquigarrow B \quad \vdash + : A \times A \to A^{\dagger}$$

$$\vdash \mathsf{distr} : \ A \times (B + C) \ \rightsquigarrow A \times B + A \times C \quad \vdash \mathsf{fuse} : \ A + A \rightsquigarrow A \quad \vdash \mathsf{inl} : \ A \rightsquigarrow A + B \quad \vdash \mathsf{inr} : \ B \rightsquigarrow A + B$$

图 7. 核心构造的类型（省略了图3中已经展示的内容）。† + 要求值与变更相同，即 $\beta = \beta'$，因为其语义使用了 $\oplus$，否则类型检查将无法通过。

$$\frac{f : [\![A]\!] \to [\![B]\!] \qquad C : \mathrm{Type} \qquad i : A \to B \times C \qquad d : A' \to C \to B' \times C}{(f, C, i, d) : \mathrm{Op}\ A\ B}$$

图 8. 用户定义运算。

形式化变化的概念。差分操作由 Triv 组合子用于计算输出差分，也用于求输出部分或全部为常量的运算的导数（如 $x \ominus x$是 零变化，对应于 x；也就是说，根据刚才提到的定律，有 $x \oplus (x \ominus x) = x$）。我们的变化结构与 Cai 等人提出的原始版本不同 [7]，因为我们的变化类型是固定的，而非依赖于值。本文中的示例和图示均假设只有一种基类型 β。这只是为了使表述简洁——我们的机械化支持任意数量的基类型。

4.2 语法与语义

本节将介绍DeCo的语法、类型系统和指称语义。图6展示了我们核心演算的语法。我们用 $s : S$ 表示形状变换，用 r 表示常量，用 $c : \beta$ 表示位置，用 i 表示运算，用 o 表示谓词。类型可以是基类型 b、形状为$\mathsf{F}_s\ A$ 的形状 s的积类型 $A \times B$，也可以是和类型 $A + B$。由于形状

是类型的一部分，我们可以静态地知道任何 DeCo 程序计算出的所有值的形状。静态形状类型检查还确保了全定义性，因为像获取这样的运算不会越界。这简化了指称语义。

DeCo 支持以下表达式 e：(i) 两个表达式的顺序组合 $e_1\ ;\ e_2$，(ii) 两个运算的并行复合 $e_1 \times e_2$，(iii) 字面量cst c。(iv) 恒等式id，(v) 复制dup，(vi) 元组投影fst 和snd，以及 (vii) 加法 +。字面量必须是符合下文定义的类型指称的有效值。

除元组外，DeCo还支持和类型。积与和的运算符基于分配范畴的结构 [12]。具体而言，可以使用inl、inr 构造和，并使用 $e_1 \mid\mid e_2$、fuse和distr进行操作。

DeCo 的表达式还包括容器操作[4]：(i) get 用于获取元素，set 用于设置索引 i 处的容器元素：$[\![s]\!]_I$。(ii) replicate 创建一个所有元素都设为同一值的数据结构。(iii) filter 接受一个针对索引 p 的谓词，并将索引 i 处满足 $p\ i$ 为 false 的所有元素置零。(iv) map e 将函数映射应用于容器。(v) zip 组合两个形状相同的容器，并将它们的元素配对。(vi) tp 对嵌套数据结构进行转置。(vii) op 允许调用用户提供的运算 o。(viii) reshape 通过形状变换 r：$[\![s_2]\!]_I \to [\![s_1]\!]_I$重新排列内容，从而改变容器的形状。后者将 s_2 中的索引映射到 s_1。

我们在图DeCo 的类型系统见图7。类型判断的形式为 $\vdash e : A \rightsquigarrow B$，意为 e 将类型为 A 的值转换为类型为 B的值。图8 形式化了我们用于描述领域特定实例化所附带额外运算的方法。用户定义函数o 被打包到Op 类型中。此类型将函数与其缓存增量化关联起来。后者包括一个缓存 C、一个初始化函数 ι和一个导数 d。（我们要求它们遵守相关定律，以确保增量化的可靠性，参见第5.3节。）

图9 展示了DeCo的指称语义。该语义隐式地以此前描述的容器和变化结构接口为参数。类型同时具有值语义 $[\![\cdot]\!]$ 和变化语义 $[\![\cdot]\!]'$，二者仅在基类型的情形下有所不同。在两种情况下，我们都将容器类型解释为从索引到值的函数。和类型的变化语义允许（i）在保持和的同一“侧”不变的情况下改变值（即，从 $\iota_1\ x$变为 $\iota_1\,(x + x')$，或从 $\iota_2\ x$变为 $\iota_2\,(x + x')$）；（ii）设置一个新值。出于篇幅考虑，该内容展示于附录15A的图。

关于术语语义，我们通过说明每个索引处的值来描述数据变换的结果。例如，$[\![\text{zip}]\!]\,(x, y)$会生成一个容器，其中每个索引 i处的条目为 $(x\ i, y\ i)$。另一个例子是，$[\![\text{重塑}\ \ r]\!]\ x$会输出一个数据结构，其索引 i处的值取自 $r\ i$中的索引 x。

4.3 增量化

在本节中，我们将介绍增量化函数。对于每一种值类型 A，我们将其对应的变化类型记为 A'。我们还用 x表示值，用 x'表示变更。函数 $f\colon A \to B$ 的增量化包括：

$$
\begin{array}{lcl}
[\![f]\!]_{\mathrm{C}} & : & \mathrm{Type} \\
[\![f]\!]_{\mathrm{i}} & : & A \to B \times [\![f]\!]_{\mathrm{C}} \\
[\![f]\!]_{\mathrm{d}} & : & A' \to [\![f]\!]_{\mathrm{C}} \to B' \times [\![f]\!]_{\mathrm{C}} \\
[\![f]\!]' & := & ([\![f]\!]_{\mathrm{C}}, [\![f]\!]_{\mathrm{i}}, [\![f]\!]_{\mathrm{d}})
\end{array}
$$

[4]我们省略前面展示过的 map2，因为它可以通过组合 映射 和 zip

$$\frac{\vdash e : A \rightsquigarrow B}{[\![e]\!] : [\![A]\!] \to [\![B]\!]}$$

$$\begin{array}{lcl} [\![\mathsf{b}]\!] & := & \beta \\ [\![\mathsf{F}_s\ A]\!] & := & [\![s]\!]_{\mathcal{I}} \to [\![A]\!] \\ [\![A \times B]\!] & := & [\![A]\!] \times [\![B]\!] \end{array} \qquad \begin{array}{lcl} [\![\mathsf{b}]\!]' & := & \beta' \\ [\![\mathsf{F}_s\ A]\!]' & := & [\![s]\!]_{\mathcal{I}} \to [\![A]\!]' \\ [\![A \times B]\!]' & := & [\![A]\!]' \times [\![B]\!]' \end{array}$$

$$\begin{array}{lcl} [\![\mathsf{恒等}]\!]\ x & := & x \\ [\![\mathsf{常量}\ c]\!]\ x & := & c \\ [\![e_1\ ;\ e_2]\!]\ x & := & [\![e_2]\!]\ ([\![e_1]\!]\ x) \\ [\![e_1 \times e_2]\!]\ (x,\ y) & := & ([\![e_1]\!]\ x,\ [\![e_2]\!]\ y) \\ [\![\mathsf{复制}]\!]\ x & := & (x,\ x) \\ [\![+]\!]\ (x, y) & := & x \oplus y \\ [\![\mathsf{首分量}]\!]\ (x, y) & := & x \\ [\![\mathsf{次分量}]\!]\ (x, y) & := & y \\ [\![\mathsf{左注入}]\!]\ x & := & \iota_1\ x \\ [\![\mathsf{inr}]\!]\ x & := & \iota_2\ x \\ [\![\mathsf{distr}]\!]\ (x, \iota_1\ y) & := & \iota_1\ (x,\ y) \\ [\![\mathsf{distr}]\!]\ (x,\ \iota_2\ y) & := & \iota_2\ (x \text{<style 恒等='3'>,}\ y) \end{array} \qquad \begin{array}{lcl} [\![\mathsf{复制}\ s]\!]\ x & := & \lambda i.\ x \\ [\![\mathsf{映射}\ e]\!]\ x & := & \lambda i.\ [\![e]\!]\ (x\ i) \\ \text{[<style 恒等='2'>zip]} \text{ <style 恒等='5'>(x<style 恒等='8'>, y<style 恒等='11'>)} & := & \lambda i.\ (x\ i,\ \ y\ i) \\ [\![\mathsf{重塑}\ r]\!]\ x & := & \lambda i.\ x\ (r\ i) \\ [\![\mathsf{获取}\ i]\!]\ x & := & x\ i \\ [\![\mathsf{设置}\ i]\!]\ (x,\ a) & := & \lambda j.\mathsf{如果}\ i = j\ \mathsf{则}\ x\ \mathsf{否则}\ a\ j \\ [\![\mathsf{转置}]\!]\ x & := & \lambda i.\ \lambda j.\ x\ j\ i \\ [\![\mathsf{筛选}\ p]\!]\ (x, a) & := & \lambda i.\ \mathsf{如果}\ p\ i\ \mathsf{则}\ a\ i\ \mathsf{否则}\ x \\ [\![\mathsf{fuse}]\!]\ (\iota_1\ x) & := & x \\ [\![\mathsf{fuse}]\!]\ (\iota_2\ x) & := & x \\ [\![e_1\ ||\ e_2]\!]\ (\iota_1\ x) & := & [\![e_1]\!]\ x \\ [\![e_1\ ||\ e_2]\!]\ (\iota_2\ x) & := & [\![e_2]\!]\ x \end{array}$$

图 9. 类型与项的语义。

$[\![f]\!]_{\mathrm{C}}$ 是缓存类型。初始化函数 $[\![f]\!]_{\mathrm{i}}$ 接收一个值 x，并返回 $(f\ x, c)$，其中 c 是初始缓存。导数 $[\![f]\!]_{\mathrm{d}}$ 接收一个输入变化 x' 和当前缓存，并返回一个输出变化以及缓存的新状态。我们用 $[\![f]\!]'$ 作为 $([\![f]\!]_{\mathrm{C}}, [\![f]\!]_{\mathrm{i}}, [\![f]\!]_{\mathrm{d}})$ 的简写。

我们在图10中给出增量化。基本运算借助Self和Add 组合子定义，我们将在第4.5节中解释。对顺序复合$(f;g)$ 进行增量化时，会合并已增量化子表达式的缓存。为计算该复合的导数，我们将缓存拆分为 f 和 g 各自的部分，分别计算它们的导数，然后重新合并更新后的缓存。同一原则适用于并行复合$(f \times g)$。当我们将映射$_s$ f 应用于数据结构时，本质上是在执行重复的并行复合。因此，缓存类型为 $[\![s]\!]_{\mathcal{I}} \to [\![f]\!]_{\mathrm{C}}$，并包含一份 f 的缓存类型副本，用于数据结构中的每个位置。与并行复合类似，对 f 的每次调用都会获得其自身版本的缓存。fuse、distr || 的增量化较为冗长，因此置于附录A。

4.4 自可维护性

对于某些函数，输出变化仅取决于输入变化，而不取决于输入本身。形式上，导数 d 具有以下性质：

$$\forall x\ y\ x'.\ d\ x\ x' = d\ y\ x'.$$

这类函数称为可自维护 [7]。可自维护函数可以在不存储中间结果的情况下进行增量化，也就是说，缓存类型可以是 Unit。这是因为只有在需要输入变化以外的额外信息时，才有必要进行缓存。我们的泛型操作是可自维护的，并且核心演算中的组合子能够保持自可维护性。

在增量化中，可以利用这一性质来提高效率，因为可自维护函数无需缓存，只需对变更进行运算，而非处理整个值。Cai 等人 [7]使用惰性求值实现了自可维护性的效率优势，但也提到了死代码消除这一替代方案。在我们的方法中，由可自维护操作构成的程序会自动产生缓存为空的增量化，因此只对变更进行运算。

$$[\![f]\!]' := \textsc{Self}^{\diamond}\ [\![f]\!] \qquad f \in \{\text{恒等}, \text{复制}, \text{首分量}, \text{次分量}, \text{zip}, \text{转置}, \text{获取}\ i, \text{设置}\ i, \text{重塑}\ r, \text{复制}\ s\text{筛选}\ p, \text{cst}\ c, \text{inl}, \text{inr}\}$$

$$\begin{array}{lcl}
[\![\text{映射}_s\ f]\!]_{\mathrm{C}} & := & [\![s]\!]_{\mathcal{I}} \to [\![f]\!]_{\mathrm{C}} \\
[\![\text{映射}_s\ f]\!]_{\mathrm{i}}\ x & := & \text{令}\ a = \lambda i.\ [\![f]\!]_{\mathrm{i}}\ (x\ i);\ (\lambda i.\ (a\ i)_1,\ \lambda i.(a\ i)_2) \\
[\![\text{映射}_s\ f]\!]_{\mathrm{d}}\ x'\ c & := & (\lambda i.\ ([\![f]\!]_{\mathrm{d}}\ (x'\ i)\ (c\ i))_1,\ \lambda i.\ ([\![f]\!]_{\mathrm{d}}\ (x'\ i)\ (c\ i))_2)
\end{array}$$

$$[\![\text{常量}\ c]\!]' := \textsc{Self}\ [\![\text{常量}\ c]\!]\ [\![\text{常量}\ 0]\!] \qquad [\![+]\!]' := \textsc{Add}^{\dagger} \qquad [\![\text{操作}\ \mathbf{o}]\!]' := \mathbf{o}$$

$$\begin{array}{lcl}
[\![f\ ;\ g]\!]_{\mathrm{C}} & := & [\![f]\!]_{\mathrm{C}} \times [\![g]\!]_{\mathrm{C}} \\
[\![f\ ;\ g]\!]_{\mathrm{i}}\ x & := & \text{让}\ (y, c_1) = [\![f]\!]_{\mathrm{i}}\ x;\ \text{让}\ (z, c_2) = [\![g]\!]_{\mathrm{i}}\ y;\ (z, (c_1, c_2)) \\
[\![f\ ;\ g]\!]_{\mathrm{d}}\ x'\ c & := & \text{让}\ (y', c_1') = [\![f]\!]_{\mathrm{d}}\ x'\ c_1;\ \text{让}\ (z', c_2') = [\![g]\!]_{\mathrm{d}}\ y'\ c_2;\ (z', (c_1', c_2')) \\
[\![f \times g]\!]_{\mathrm{C}} & := & [\![f]\!]_{\mathrm{C}} \times [\![g]\!]_{\mathrm{C}} \\
[\![f \times g]\!]_{\mathrm{i}}\ (x_1, x_2) & := & \text{让}\ (y_1, c_1) = [\![f]\!]_{\mathrm{i}}\ x_1;\ \text{让}\ (y_2, c_2) = [\![g]\!]_{\mathrm{i}}\ x_2;\ ((y_1, y_2), (c_1, c_2)) \\
[\![f \times g]\!]_{\mathrm{d}}\ (x_1', x_2')\ c & := & \text{让}\ (y_1', c_1') = [\![f]\!]_{\mathrm{d}}\ x_1'\ c_1;\ \text{令}\ (y_2', c_2') = [\![g]\!]_{\mathrm{d}}\ x_2'\ c_2;\ ((y_1', y_2'), (c_1', c_2'))
\end{array}$$

图 10. DeCo项的增量化。Self组合子对可自维护（即无缓存）运算进行增量化。我们将 $\textsc{Self}^{\diamond}\ x$ 记作 $\textsc{Self}\ x\ x$的简写。†增量加法需要结合律和交换律（见第 4.5节）。

$$\frac{f : A \to B}{\text{Triv}\ f := (A,\ \lambda x.(f\ x,\ x)\ \lambda(x',\ x).\ (f(x \oplus x') \ominus f\ x,\ x \oplus x'))}$$

$$\frac{f : A \to B}{\text{Triv2}\ f := (A \times B,\ \lambda x.(f\ x,\ (x,\ f\ x))\ \lambda(x',\ (x,\ y_1)).\ \text{let}\ y_2 := f(x \oplus x');\ (y_2 - y_1,\ (x \oplus x',\ y_2)))}$$

$$\frac{f : A \to B \qquad d : A' \to B' \qquad f(x \oplus x') = f\ x \oplus d\ x'}{\text{自身}\ f := (\text{Unit}\ \lambda x.(f\ x,\ \star)\ \lambda(x',\ \star).\ (d\ x',\ \star))} \qquad \frac{f : A \to B \qquad A = A' \qquad B = B' \qquad f(x \oplus x') = f\ x \oplus f\ x'}{\text{Lin}\ f := (\text{Unit}\ \lambda x.(x, \star)\ \lambda(x', \star).(f\ x', \star))}$$

$$\frac{f : A \times B \to C \qquad A = A' \qquad B = B' \qquad C = C' \qquad f(x \oplus x',\ y) = f(x,\ y) \oplus f(x',\ y) \qquad f(\text{x<style 恒等='3'>},\ y \oplus y') = f(\text{x<style 恒等='8'>},\ y) \oplus f(\text{x<style 恒等='13'>},\ y') \qquad \forall(x\ y : C).\ x \oplus y = y \oplus x \qquad \forall(x\ y\ z : C).\ x \oplus (y \oplus z) = (x \oplus y) \oplus z}{\text{BiLin}\ f := (A \times B,\ \lambda(x,\ y).\ (f(x,\ y)\ x,\ y)\ \lambda(x',\ y')\ (x,\ y).\ (f(x',\ y') \oplus f(x',\ y) \oplus f(x,\ y')\ x \oplus x',\ y \oplus y'))}$$

$$\frac{A = A' \qquad \forall(x\ y : A).\ x \oplus y = y \oplus x \qquad \forall(x\ y\ z : A).\ x \oplus (y \oplus z) = (x \oplus y) \oplus z}{\text{Add} := (\text{Unit}\ \lambda(x,\ y).\ (x \oplus y,\ \star)\ \lambda(x',\ y',\ \star).\ (x' \oplus y',\ \star))}$$

图 11. 增量化组合子。

例如，考虑zip 函数。对zip 的某个参数所做的变更，会作为输出的变更传递下去，而无需使用当前值。具体来说，我们有 $[\![\text{zip}]\!]'(x \oplus x',\ y \oplus y') := [\![\text{zip}]\!]\ (x',\ y')$。因此，计算 $[\![\text{zip}]\!]'$ 时不会使用 x 或 y，而只检查变更。

4.5 增量化组合子

将 DeCo 实例化到特定领域时，用户必须为新增的基本操作提供增量化。为帮助实现此类增量化，我们提供了一组增量化组合子，如图 11所示。每个组合子都有一些

输入和前置条件，并返回一个元组 $(C,\ i,\ d)$，使得 d是函数 f的导数，其中 C、 i和 d具有第 4.3 节中声明的正确类型。

对于任意函数 f，我们都可以使用 Triv 组合子来创建一种平凡的增量化，也就是说，在发生变更时只重新计算值。从技术上说，这种构造满足将其视为增量化所需的形式要求，但并未带来任何加速。不过，这类增量化仍然可能有用：考虑实数乘法等数乘运算。由于在这里通过增量化不可能实现渐近加速，我们不妨采用平凡的增量化。Triv 组合子依赖差分函数 $\ominus$。对于任意值 a、b，它都会计算出一个变更 $a \ominus b$，将 b 转换为 a。例如，如果 A 是实数集合，且更新操作是加法，那么 $\ominus$ 就是减法。对于数乘，我们将平凡的增量化写作 Triv $[\![*]\!]$。由此得到的导数就是 $\lambda((x_1', x_2'), (x_1, x_2)).(y \ominus (x_1 \cdot x_2)\ x \oplus x')$，其中 $y := (x_1 \oplus x_1') \cdot (x_2 \oplus x_2')$ 是新的输出。Triv 组合子的缺点是会对 f 求值两次。因此，我们还实现了 Triv2 组合子，它只对 f 求值一次，方法是同时存储 f 的输入和输出。这样就可以以内存换取速度。

组合子Self 简化了自维护函数这一特殊情形的增量化。回顾一下，在增量化语境中，如果一个函数的输出变化只依赖于输入变化，而不依赖于当前值，那么它就是可自维护的。换言之，它具有常导数。更准确地说，对于可自维护函数 $f : A \to B$，存在一个函数 $d : A' \to B'$ 使得 $f(x \oplus x') = f\ x \oplus d\ x'$。例如，函数 (cst c) 是可自维护的，其导数为 0。这本质上是因为 cst 具有恒定的输出，且变更会被丢弃。因此导数为 $[\![\text{cst}\ 0]\!]$，并且我们可以将其增量化定义为 Self $[\![\text{cst}\ c]\!]\ [\![\text{cst}\ 0]\!]$。

Self 组合子构造的增量化无需缓存，这是通过将缓存类型定义为 Unit 来表示的。（我们用 ★ 表示 Unit 类型中唯一的元素。）在 DeCo 中，数量惊人的运算是通过 Self 实现的，参见图 10。例如，很容易看出，dup 是可自维护的，因为我们有 $[\![\text{dup}]\!]\ (x \oplus x') = (x \oplus x', x \oplus x') = (x, x) \oplus (x', x') = [\![\text{dup}]\!]\ x \oplus [\![\text{dup}]\!]\ x'$。

其余组合子仅适用于值和变更属于同一类型的特殊情况，即 $A = A'$。要证明这一点对所有类型都成立，只需对基类型进行证明。

对线性函数进行增量化会更加简单，因为它们就是自身的导数。从本质上说，当 $f = d$ 时，线性函数就是可自维护函数。请注意，这仅对函数 f 有明确定义： $A \to B$ 其中 $A = A'$ 且 $B = B'$（如果 $\beta = \beta'$，即基值类型和变化类型相同，则此条件成立）。

我们还可以对双线性函数进行增量化。这类函数是二元函数，分别对其第一和第二个参数呈线性。与线性函数不同，对双线性函数进行增量化确实需要缓存 [5]。关系交叉乘积就是双线性函数的一个例子。

最后，如果更新操作（$\cdot \oplus \cdot$）满足结合律和交换律，那么它也可以用作运算，使其成为自身的导数。这是因为对（$\cdot \oplus \cdot$）进行增量化的可靠性属性为

$$(x \oplus x') \oplus (y \oplus y') = (x \oplus y) \oplus (x' \oplus y'),$$

这显然可由结合律和交换律推出。

4.6 在静态形状下扩展数据

我们的核心演算在设计上强制静态容器形状：容器的形状在类型层级固定，不能在执行期间动态改变。这确保了诸如设置、获取、转置等关键运算的全定义性。形状变更操作可以实现，只要形状仍然

静态固定。例如，我们可以将追加实现为从形状 n 到形状$n + 1$的函数：

$$\mathsf{append} : A \times \mathsf{F}_n\ A \rightsquigarrow \mathsf{F}_{n+1}\ A := (\mathsf{id} \times \mathsf{reshape}\ (\lambda i.\ i - 1))\ ;\ \mathsf{set}\ 0$$

将变化结构限制为维持当前形状，也使运算 获取 具有自可增量性。尽管如此，仍可以实现可增长的数据结构，正如我们在第 7 节的关系代数和树示例中所展示的那样。在我们的线性代数案例研究中，我们将张量的维度纳入形状，例如，数组的长度是静态跟踪的；而在关系代数案例研究中，我们只静态跟踪关系的 模式，例如，关系中的项目数量可以动态增长。此外，在树案例研究中，我们将形状简单地视为 Unit类型，这实际上意味着所有形状都是动态处理的。

5 可靠性

本节将介绍我们的 Lean 机械化工作，其中包括一项证明，表明我们的增量化是保值的，也就是说，它计算出的结果与完整重新求值所得的结果相同。这体现了这样一种理念：增量化是一种优化，应当避免不必要的重新计算，同时保留原程序的行为。

本节分为五个小节，系统地确立正确性。首先，我们通过内在类型（§ 可类型性保持，§ 5.1）来证明可类型性保持。具体而言，我们展示了机械化如何利用内在类型，确保指称语义、增量过程和组合子实现均满足可类型性。其次，为了形式化增量化，我们将数据类型变化的正式定义 表述为变化结构（§ 5.2）。如第 4 节所述，我们假设基类型构成一种变化结构。随后，我们证明这保证了所有类型（即任意嵌套的容器、积和和）都构成变化结构，因此任何类型都有有意义的变化概念。第三，我们引入一致增量化 的概念，以刻画增量元组各元素必须满足的要求，并证明所有一致的增量化都保持值，也就是说，其结果与完全重计算得到的结果相同（§ 5.3）。第四，我们只需证明我们的变换组合子和增量化组合子是一致的，从而保持值（§ 5.4）。最后，我们陈述并证明 DeCoDeCo 的运算在何种条件下能确保容器具有有限支撑，这对于关系等无限容器的有限表示很重要（§ 5.5）。

5.1 可类型性保持

我们采用 内在类型，这意味着 Tm 中的项是以输入类型和输出类型为参数的。使用内在类型意味着类型规则直接集成在语法声明中，因此只能构造良类型项。

因此，将一对值（其类型为b这一基类型）转换为单个值的程序，其类型为 Tm (prodb b)b。下面左侧的代码清单简要展示了Ty 和 Tm 的部分定义（分别对应图 6中的对象类型和表达式），并省略了一些类型参数。

归纳定义 Ty : 类型 def Tm.denote :| 基类型 : 类型 | 积 : Ty → Ty → 类型 | . . . Tm A B → A.toCS.V → B.toCS.V 归纳定义 Tm : Ty → Ty → 类型 (u+1) def Tm.incr (e : Tm AB) :| 恒等 : Tm A A | 序列 : Tm A B → Tm B C → Tm A C | . . . ConsIncr A.toCS B.toCS e.denote

结构 ConsIncr（A B : 变化结构）（f : A.V → B.V）： 类型1 其中
C: 类型i: A.V →B.V ×
C d: A.D → C →B.D ×
C

h_0 : ∀ x, (i x).1 = f x
h_1 : ∀ x x', f (x + x') = f x + (d x' (i x).2).1
h_2 : ∀ x x', (d x' (i x).2).2 = (i (x + x')).2

上方右侧的代码清单声明了指称语义和增量化。该声明确保指称和增量化都具有适用于项e的正确类型。使用内在类型可确保Tm.denote 和Tm.incr 都是定义良好的全函数，这是简化后续推理与证明的一项关键性质。

请注意，toCS : Ty →变化结构 是类型的指称语义，它将每种类型与一个变化结构相关联（详见第5.2节）；而ConsIncr 则是一个函数的缓存增量化及其一致性证明，我们将在第5.3节进一步讨论。

5.2 容器与变化结构

我们的开发以一个容器、一个基类型和一组领域专用操作为参数。现在，我们描述这三个参数必须满足的性质，以确保增量化正确。(i) 如第4.1节所述，容器由一个形状类型以及一个将每种形状映射到索引类型的函数构成。在 Lean 形式化中，我们还要求每种形状的索引类型具有可判定相等性。这一要求并不苛刻，因为大多数容器的索引都具有可判定相等性，从而能够区分其元素的位置。(ii) 基类型构成一个变化结构，其中包含值和变更的类型、一个更新运算、一个差集运算以及完备性证明。(iii) 为了添加更多特定于数据类型的操作，语法类型Tm 包含一个操作op，其类型签名为op : OpA B → Tm A B。类型Op 的元素是一个从A 到B的任意增量式化操作。定义该操作时，需给出一个从A 到B 的函数f，以及f的一个正确增量化。

下面定义这三个参数。

结构容器**其中**
S : 类型形状 : S →类型[h: ∀ s, DecidableEq (形状 s)]

结构变化结构**其中**
(A :类型) (A': 类型)
更新 : A → A' →A
差分: A →A →A'
完整: update x (diff y x) =y

结构 Op (AB : Ty) **其中**
f: A.toCS.V → B.toCS.V
i: ConsIncr A.toCS B.toCS f

为了讨论变更，我们需要为每种对象类型配备一种代数结构，使我们能够应用更新。因此，我们的第一个引理是：如果给定的基类型构成一种变化结构，那么我们的对象类型（图 5，图 9）也将构成一种变化结构：

引理 5.1。
如果基类型 ⟦b⟧ 的语义构成一个变化结构，那么所有对象类型 ⟦A⟧（由基类型、容器、积和和构造的类型，图 *6*）的语义也构成一个变化结构。

证明。对 A进行归纳。 □

我们对该证明的 Lean 机械化是构造性的：我们定义了一个函数 toCS，它将每个类型映射到一个变化结构。类型为 变化结构 的元素是一个元组，由 (A, A', $\oplus$, $\ominus$) 以及完备律组成：$x \oplus (y \ominus x) = y$。

5.3 一致增量化保持值

现在，我们正式阐述缓存增量化正确性的含义。首先，我们给出增量化应满足的三条一致性定律。随后，我们定义值保持性这一性质，用以形式化表达这样的观点：应用于变更列表时，增量化应产生与

重新求值相同的结果。随后，我们证明一致性蕴含值保持性。

接下来，我们假设 A 和 B 是变化结构。对于函数 $f\colon A \to B$，我们回顾增量化生成的函数，并陈述它们必须满足的三条定律。

定义 *5.2*（增量化）。
元组$(C$：类型 $i\colon A \to B \times C,\ d\colon A' \to C \to B' \times C)$ 是一致增量，适用于$f\colon A \to B$ 当且仅当

$$\forall x. \qquad i_1\ x = f\ x \tag{Law-1}$$

$$\forall x\ x'. \qquad f\ (x \oplus x') = f\ x \oplus d_1\ x'\ (i_2\ x) \tag{Law-2}$$

$$\forall x\ x'. \qquad i_2\ (x \oplus x') = d_2\ x'\ (i_2\ x). \tag{Law-3}$$

我们将函数结果的第一个分量记为 i_1 和 d_1，第二个分量记为i_2 和 d_2。定律 1 要求初始化 $i\ x$ 的第一个分量等于原函数 $f\ x$。定律 2 要求增量化 d_1 是 f的离散导数。这意味着，如果给定 d_1 一个针对 x 的缓存和一个变化 x'，它将返回 $f\ x$ 和$f(x + x')$之间的差集。定律 3 要求增量化 d_2 也能正确维护缓存。这意味着，如果我们从输入 x 初始化缓存，然后应用一次更新 x'，那么新缓存应等于从 $x + x'$初始化所得到的缓存。

现在我们陈述增量化应满足的主要性质，即值保持性。直观地说，如果对于任意变更序列，增量化计算出的结果与原始计算相同，则称该增量化具有值保持性。更准确地说，我们定义函数 iter，将增量化应用于一个输入变化列表。我们证明，iter 的结果等于将函数应用于发生变化的输入所得的结果。

iter 的定义如下（其中 Incr $A\ B$ 表示增量元组类型（C: 类型 $A \to B \times C,\quad A' \to C \to B' \times C$）：

$$\begin{array}{lcl}
\mathsf{iter} & : & (I : \mathsf{Incr}\ A\ B) \to A \to \mathsf{List}\ A' \to B \times I_C \\
\mathsf{iter}\ (C, i, d)\ x\ [\,] & := & i\ x \\
\mathsf{iter}\ (C, i, d)\ x\ (x' :: xs') & := & \mathsf{let}\ (y, c_1) := \mathsf{iter}\ (C, i, d)\ x\ xs' \\
 & & \mathsf{let}\ (y', c_2) := d\ x'\ c_1 \\
 & & (y \oplus y', c_2)
\end{array}$$

注意，变更是从后往前应用的。为了表示完全重新求值，我们需要使用下面的求和函数计算应用变更列表后的输入值。

$$\begin{array}{lcl}
\mathsf{sum} & : & A \to \mathsf{List}\ A' \to A \\
\mathsf{sum}\ x\ [\,] & := & x \\
\mathsf{sum}\ x\ (x' :: xs') & := & \mathsf{sum}\ x\ xs' \oplus x'
\end{array}$$

现在我们可以定义值保持性质。

定义 *5.3*（值保持增量化）。
如果某个增量化 I: Incr $A\ B$ 对于 $f\colon A \to B$ 具有值保持性，则称其满足

$$\forall (x : A)\ (xs' : \mathsf{List}\ A).\ (\mathsf{iter}\ I\ x\ xs')_1 = f\ (\mathsf{sum}\ x\ xs').$$

最后，我们证明，表明增量化是一致的（根据定义 5.2）足以确保增量化是保值的。

引理 5.4。
如果 $(C, i, d)\colon$ Incr $A\ B$是 $f\colon A \to B$ 的一致增量化，那么相对于 f，它是保值的。

证明。我们通过对 xs'进行归纳法，证明以下更强的性质：

$$\forall x\ xs'.\ \mathrm{iter}\ I\ x\ xs' = i\ (\mathrm{sum}\ x\ xs').$$

该引理由第一条一致性定律（$\forall x.\ i_1\ x = f\ x$）得出。 □

5.4 增量化与组合子保值

现在我们证明，我们的增量化函数和增量化组合子会生成一致的、因而保值的增量化结果。

我们首先 证明增量化函数的结果满足一致性律 .

引理 5.5。
对于一个表达式 e，元组 $(\llbracket e \rrbracket_C, \llbracket e \rrbracket_f, \llbracket e \rrbracket_d)$ 是 $\llbracket e \rrbracket$ 的一致增量。

证明。对 e 使用归纳法。 □

在我们的 Lean 机械化中，这意味着 Tm.incr 具有以下签名（我们已在上文展示过），从而确保它始终返回正确的增量化结果：

```
定义Tm.incr : (e:Tm A B) → ConsIncr A.toCS B.toCS e.denote
```

根据这个签名，值保持性显然成立。函数 Tm.incr 通过对语法进行结构递归来定义。为了确保满足定义 5.2 中的一致性律，我们针对每一种语言构造分别证明这些定律。如图 10所示，我们的形式化采用了增量化组合子。请注意，每当使用一个组合子时，我们都必须证明其所有前置条件均得到满足。以 恒等 运算（图 10）为例，我们使用 Self 组合子（图 11）对其进行增量化。这里，我们必须证明 Self 的前置条件 $\llbracket \mathrm{id} \rrbracket\ (x \oplus x') = \llbracket \mathrm{id} \rrbracket\ x \oplus \llbracket \mathrm{id} \rrbracket\ x'$，而这显然可化简为恒等式。

定理 5.6（值保持增量化）。对于表达式 e，元组 $(\llbracket e \rrbracket_C, \llbracket e \rrbracket_f, \llbracket e \rrbracket_d)$ 是 $\llbracket e \rrbracket$ 的一种保值增量化。

证明。 根据引理 5.5和引理5.4。 □

定理 5.7（保值组合子）。组合子 *Triv*、*Self*、*Lin*、*BiLin* 和 *Add* 都是保值的。

证明。 我们通过等式推理证明每个组合子的一致性，由此根据引理 5.4 可推出值保持性。 □

在 Lean 中，每个增量化组合子的一致性都以与Tm.incr类似的方式表达，即让它们返回ConsIncr类型的值。更准确地说，每个组合子都接受一个函数f:A.V →B.V （以及一些额外数据，对应于图11中的前提），并返回ConsIncrABf类型的值。

5.5 有限支撑

上述结果确保了我们方法的外延正确性。在本小节中，我们希望讨论一个与使用有限数据结构（如哈希映射）表示容器的实现相关的问题：如果容器的某些形状具有无限多个索引，我们需要确保程序只产生具有有限支撑的容器，即只有有限个索引被映射到非零值的容器。因此，我们给出定理，说明在何种条件下我们的内置运算能够保持有限支撑性质。所有有限支撑性质的一个常见条件是：对于容器所应用的类型，都存在一个默认值（下文称为 ε）。该 ε 用于表示值的缺失。

为形式化有限支撑保持这一概念，我们将有限支撑映射定义为函数 $f\colon A \to B$，对于这些函数，存在一个 l：列表 A，使得对于所有 $k \notin l$，$f\ k = \varepsilon$。随后我们证明，返回容器的运算能够保持有限支撑。具体而言，我们将这些运算的指称语义应用于有限支撑映射，从而证明每个运算都会返回一个有限支撑映射。

定理 5.8（有限性保持）。
运算符 设置、 复制、 映射、 重塑、 筛选、 zip、 转置 在以下前置条件下保持容器的有限支撑：*(i)* 对于 映射 f，我们要求 $[\![f]\!]\ \varepsilon = \varepsilon$。*(ii)* 对于 复制 x，我们要求$x = \varepsilon$。*(iii)* 对于 重塑 r，我们要求对于输入中的每个索引 $r\ i$，输出中满足 $r\ j = i$的索引 j只有有限多个。

定理中的条件防止运算设置无限多个非零值。我们的关系代数和树用例满足这些要求，因为我们没有定义从有限关系创建无限关系的运算。线性代数和无冲突复制数据类型用例不受这些前置条件约束，因为它们只使用有限数据结构。

6 实现

第5节中对DeCo 的机械化旨在验证我们的正确性主张，但不能直接用于实际场景。具体而言，使用函数表示数据结构会使性能推理变得困难。因此，我们提出了一种实现，它采用基于哈希映射的表示方式，并且在编写实例时也更加方便。

我们的实现基于无标签终结式风格 [8]。这意味着，我们不将抽象语法定义为归纳数据类型，而是将其定义为类型类，其中每个函数都对应一个构造子。这样，我们就可以通过实例化这个类型类来定义解释器。

6.1 高效表示

我们尽可能使库实现贴近形式化定义。不过，为了确保所有运算都能高效计算，我们的实现与演算在若干细节上略有不同。重要的是，这些差异都不会影响其可靠性。

最显著的区别是，形式化定义将数据结构表示为函数，而我们的库使用哈希映射。这样可以急切地计算结构中的值，并利用其稀疏度。后者在处理变更时尤为重要，因为变更往往是稀疏的（例如，只修改关系中的少数几个元组）。我们可以通过原地更新数据结构，确保稀疏变更得到高效应用。在我们的实现中，Lean 的函数式但原地更新语义 [28], 使这一点成为可能：它允许对线性使用的数据结构进行破坏性更新，同时保持纯函数式行为。在其他语言中，也可以改用支持高效插入的纯函数式数据结构 [26]。

如第 5.5 节所述，将无限数据结构表示为（有限）哈希表时需要格外谨慎。我们的实现假设程序员遵守第 5.5 节所述的前置条件（处理数组等有限数据结构时则无需如此）。

最后，形式化要求容器的索引具有可判定相等，而实现则要求布尔相等和一个哈希函数[5]。

需要注意的是，第 5 节中的正确性论证同样适用于我们的实现。(i) 对索引要求可判定相等而非布尔相等

[5]Lean 中，可判定相等与布尔相等的区别在于，前者需要提供相等或不相等的证明，而后者只是一个返回布尔值的函数。

对正确性而言也没有问题。这是因为，布尔值相等函数的存在可直接由可判定相等的存在推出。(ii) 哈希映射显然是有限支撑函数的一种有效实现。因此，第 5 节中的正确性论证同样适用于我们的实现。

6.2 语法糖

虽然DeCo主要是一种形式演算，但我们添加let绑定作为语法糖。其核心思想是，let绑定let $x = e_1$; e_2 可以翻译为现有的DeCo组合子：复制上下文，将一份副本传递给 e_1，然后将 e_1 的结果连同另一份副本传递给 e_2：

$$[\![\text{let } x = e_1;\ e_2]\!] = \text{dup} \ ; ([\![e_1]\!] \times \text{id}) \ ; [\![e_2]\!]$$

命名变量的翻译分两步：从命名变量到de Bruijn变量，再从de Bruijn到DeCo的组合子。第一步将命名变量翻译为de Bruijn索引。命名项的类型上下文最初是一个变量-类型对列表。去掉变量名后，它们就成为de Bruijn上下文。例如，命名上下文x：A, y：B被翻译为de Bruijn上下文 A, B。相应地，项中的变量出现通过查找其在上下文中的位置而被翻译为索引。因此，x：A, y：$B \vdash y$：B被翻译为 $A, B \vdash 1$：B，因为 y 是上下文中的第二个变量，而我们从0开始计数。

第二步将 de Bruijn 项转换为组合子项。此步骤会完全消除上下文，并以嵌套积取而代之。也就是说，A，B会被转换为 $A \times B$，而 A，$B \vdash e$：C则会转换为 e'：$A \times B \to C$。相应地，它会将 de Bruijn 变量转换为投影。例如，A，$B \vdash 1$：B会被转换为次分量：$A \times B \rightsquigarrow B$。

示例。我们举例说明第3 节中的矩阵-向量乘法和全连接层使用我们的语法糖后是什么样子。这里，NTm 表示一个命名项，# 表示函数应用，而 [m,v] 用于构造命名项（如mvmul）的参数列表。

```
定义mvmul : NTm [("m", V n (V m Float)), ("v", V m Float)] (V n Float) ： =« 映射 求和 #
(map2 (map2 乘法) # (复制 # v, m)) »定义dense : NTm  [("m", V n (V m Float)), ("b",
V n Float), ("x", V mFloat)] (V n Float) ： =« 映射 ReLU # map2 Add # (mvmul #  [m,
x], b) »
```

7 应用

在本节中，我们介绍已实现的实例，展示如何将 DeCo 应用于线性代数、关系代数、树和CRDT。由此可见，DeCo 能够在全新的领域实现增量化，而无需重复造轮子。我们对线性代数、关系代数和树进行了原理验证性的性能评估。此处讨论线性代数示例（一个稠密神经网络层）的结果。对于其他应用，我们仅作简要介绍，更多详情见附录B。

7.1 线性代数

在第3节中，我们介绍了线性代数实例的语法和指称语义的实现。本节将介绍该实例各项运算的增量化，并讨论如何在 DeCo 中编码常见的线性代数运算。得益于增量化组合子，这些增量化的实现十分直接（Map.sum 对以映射表示的容器中的值求和）：

```
定义 mul ： = triv (λ (x, y)  =>x * y)定义
relu ： = triv (λ x  => 若 x  > 0 则 x 否则 0)
```

表 1. DeCo 中的线性代数运算。

运算	线性代数	DeCo
向量求和	$v_1 + v_2$	映射2 +
矩阵和	$M_1 + M_2$	map2 (映射2 +)
Hadamard 积	$v_1 \circ v_2$	映射2 *
点积	$v_1 \cdot v_2$	映射2 *;求和
标量向量乘积	$c \cdot v$	标量向量乘法
矩阵向量乘积	$M\ v$	*mvmul*
矩阵乘积	$M_1\ M_2$	*mmmul*

其中 标量向量乘法 := (复制 n×恒等);映射2 *,
以及 *mmmul* := 复制 n×转置;map2 *mvmul*; 转置

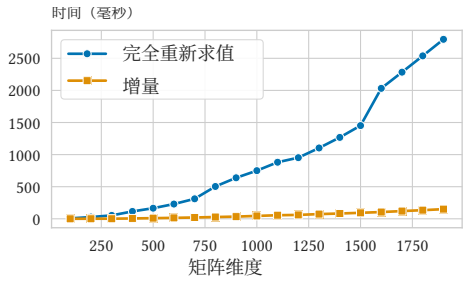


图 12. 线性代数基准测试：在 *dense* 上的完整求值与增量更新。

```
def sum := self Map.sum Map.sum
```

在表 1中，我们展示了如何在 DeCo (再次使用映射2 f作为 zip的缩写；映射 f）。我们假设 *mvmul* 的定义与第 3节相同。

求值。 为了初步定性评估我们的方法所带来的加速效果，我们使用前文定义的全连接神经网络层 *dense* 运行示例进行了一项小规模基准测试。我们比较了完整求值全连接层所需的时间，以及当 1% 的输入元素发生变化时更新输出所需的时间。基准测试的输入向量大小范围从 $n = 100$ 到 $n = 2000$ （权重矩阵的大小始终为 $n \times n$）。每项时间测量均取 5[6]次结果中的最佳值。图12 所示结果表明，与重新求值相比，增量化带来了显著的加速。这些结果旨在证明我们的方法切实可行；请注意，我们当前的实现使用的是未经优化的解释器。因此，目前我们并未声称其性能可与采用编译和专用线性代数库的先进框架相媲美。请注意，增量化所需的缓存可以从增量化转换的定义中直接读出。具体而言，*dense* 的缓存类型（忽略 Unit 类型的缓存）为 $(\mathbb{R}^2)^{n \cdot m} \times \mathbb{R}^n$ 对于一个 $m \times n$ 矩阵，因此缓存包含 $2nm + n$ 个数。

我们还评估了随着变更规模增大，性能会如何下降（更多内容见第 B.1 节）。在我们的示例中，只有当输入变化不超过 70% 时，增量求值才比重新求值快。请注意，无论矩阵的稠密程度如何，我们的评估都使用相同的基于哈希映射的表示；若采用稠密表示，重新求值开始变得更快的临界点很可能会低于 70%。

7.2 关系代数

我们遵循 DBSP [5]，将数据库表表示为从元组到整数的映射，用以表示该元组在表中的重数（负整数用于表示元组的移除）。也就是说，我们以基类型 $\beta := \mathbb{Z}$，并以形状类型 S 作为一组模式。具体而言，我们的实现将模式定义为整数、字符串或两个模式的二元组，但这可以更改而不影响该实例的其余部分。请注意与线性代数示例的对比：在那里，向量的值就是容器的值；而在这里，关系的值是容器的 索引 。更形式化地说，模式为

[6]我们在 ThinkPad T14 Gen 4 上以单线程运行了所有实验。该设备配备 AMD Ryzen 7 PRO 7840U 中央处理器（3.3 GHz），运行 Linux 内核 6.12.63 和 Lean 4.22.0-rc4。

$$
\begin{array}{lcl}
* & : & \mathsf{b} \times \mathsf{b} \rightsquigarrow \mathsf{b} \\
x \, [\![*]\!] \, y & := & x \cdot y \\
[\![*]\!]' & := & \textsc{Triv} \, [\![*]\!] \\
- & : & \mathsf{b} \times \mathsf{b} \to \mathsf{b} \\
x \, [\![-]\!] \, y & := & x - y \\
[\![-]\!]' & := & \textsc{Lin} \, [\![-]\!] \\
\otimes & : & \mathsf{F}_s \, \mathsf{b} \times \mathsf{F}_{s'} \, \mathsf{b} \rightsquigarrow \mathsf{F}_{s,s'} \, \mathsf{b} \\
R \, [\![\otimes]\!] \, S & := & \lambda(i,j).\, x \, i \cdot y \, i \\
[\![\otimes]\!]' & := & \textsc{BiLin} \, [\![\otimes]\!]
\end{array}
$$

图 13. 关系代数实例运算。

表 2. DeCo 中的关系代数运算。

运算	关系代数	DeCo
笛卡尔积	$R \times S$	$\otimes$
Join	$R \bowtie_p S$	$\otimes$; 筛选 p
选择	$\sigma_p(R)$	筛选 p
并集	$R \cup S$	映射2 +
差集	$R - S$	映射2 −
交集	$R \cap S$	映射2 $*$

s 被表示为从 $[\![s]\!]_{\mathcal{I}}$ 到 $\mathbb{Z}$的映射，其中索引 $[\![s]\!]_{\mathcal{I}}$ 是该关系的值。于是，一次变更相当于向表中添加或从表中移除任意数量的元组。

我们在图 13 中形式化地描述该实例。笛卡尔积 $\otimes$ 接受两个模式分别为 s 和 s' 的关系，并返回一个模式为它们的积模式 s 的关系，s'。这是唯一一个未使用增量化组合子定义的实例运算。它的增量化可视为乘法的离散导数：

$$((a + a') \cdot (b + b')) - (a \cdot b) = a \cdot b' + a' \cdot b + a' \cdot b'.$$

在表 2中，我们展示了如何在 [9] 中表达若干关系代数运算 DeCo。将关系代数定义为DeCo的一个实例，比从头定义其增量化更简单，原因有三。首先，在关系代数运算中，只有笛卡尔积必须显式定义为一项附加运算；其他运算都可以在 DeCo 中定义为派生操作，因此不需要额外的证明工作量。其次，借助增量化组合子，无需手动编写任何增量化定义。第三，增量化组合子减少了证明工作量，因为只需证明 $\otimes$ 的双线性，以及整数加法和减法的一些平凡性质。

字典。为了定义分组等聚合操作，我们同时使用关系和字典。字典 [30] 是容器的一种特殊情况，其形状类型只有一个元素，因此索引类型不依赖于形状。具体来说，对于给定的键类型 K，字典可以视为一个函子，将 V映射到 $K \to V$，其中的函数假定为有限函数。因此，对于每个 K，我们定义一个容器 D^K，它只有一个形状 $\bigstar$，索引类型为 $[\![\bigstar]\!]_{\mathcal{I}} := K$。根据我们的容器语义，将 D^K 应用于类型 A 表示 $[\![\mathsf{D}^K_\star \, A]\!] = ([\![\bigstar]\!]_{\mathcal{I}} \to A) = (K \to A)$。

聚合。我们关于聚合的第一个例子是 $\mathsf{count} : \mathsf{F}_s \, \mathsf{b} \rightsquigarrow \mathsf{b}$，它用于计算关系中的行数。我们使用 Self 组合子对其进行增量化。同样，我们还可以定义求和、最大值和最小值，此处略过。我们还定义了运算 $\mathsf{groupBy} : ([\![s]\!]_{\mathcal{I}} \to K) \to \mathsf{F}_s \, \mathsf{b} \rightsquigarrow \mathsf{D}^K_\star \, (\mathsf{F}_s \, \mathsf{b})$，其灵感来自 DBSP。除了关系容器外，它还使用了上文定义的字典容器。[7] 定义 count 和 groupBy 后，我们可以构造派生操作 proj :=groupBy f; map count。我们在附录B.2中介绍了对增量 proj 的原理验证式评估，并展示了它相较于重新求值的加速效果。

[7]使用给定的类型进行**分组**，而不是**分组**：$([\![s]\!]_{\mathcal{I}} \to [\![s']\!]_{\mathcal{I}}) \to \mathsf{F}_s \, \mathsf{b} \rightsquigarrow \mathsf{F}_{s'} \, (\mathsf{F}_s \, \mathsf{b})$ 允许按任意类型进行分组 K，而不只是按可用模式表示的类型进行分组 s'。

S :=Unit　　$[\![\bigstar]\!]_I$: $= \mathbb{I}$　　$\beta := \mathbb{N}$　　$+ := +_{\mathbb{N}}$　　$0 := 0_{\mathbb{N}}$

$[\![\text{最大值}]\!]\ x\ y$	:=	若 $x < y$ 则 y 否则 x	值	:=	sum
$[\![\text{最大值}]\!]'$	:=	Triv $[\![\text{最大值}]\!]$	自然数递增	:=	恒等，常量 1； +
$[\![\text{求和}]\!]\ x$	:=	$\Sigma_i(x\ i)$	inc_i	:=	$\langle$获取; *incNat*, 恒等$\rangle$;设置$_i$
$[\![\text{求和}]\!]'$	:=	Lin $[\![\text{求和}]\!]$	合并	:=	map2 最大值

图 14. GCounter 无冲突复制数据类型实例，包含派生操作。

7.3 树

将树状结构表示为容器的自然方法，是将其表示为从路径到标量值的（部分）映射，类似于 XPath 和 JSONPath。我们希望变更能够添加和移除树节点。我们通过将“形状”的类型设为 Unit 类型（与我们的容器表示类似，也类似于如何将动态类型语言表示为单类型语言），并将索引集设为所有可能路径的集合，来表示形状可变的树。路径到值的映射具有部分性的这一事实，反映在值类型包含一个“空值”上；这确保我们的运算在形式上仍然是全定义的（但它们可能会生成包含空值的结构）

XQuery 示例。 我们实现了一个受《XQuery用例》文档中查询 1 启发的示例[8]，Matsuda 等人也研究过这一示例。 [21]。在此示例中，我们查询书目，找出 1991 年之后由 Addison-Wesley 出版的书籍，并返回其标题和年份。如上所述，我们的增量程序预期书籍以从路径到标量值的映射形式表示。普通树与映射之间的转换在 Lean 中实现。数据和查询见附录B.3。

玫瑰树上的折叠。 现在我们描述如何对玫瑰树上的折叠进行增量化。玫瑰树是每个节点可以有任意数量子节点的树。与 XQuery 示例类似，我们首先在 Lean 中将玫瑰树定义为归纳类型，然后实现转换为从路径到标量值的映射。折叠随后接受一个函数 f，该函数应用于每个路径-值对；一个二元函数 m，用于合并 f 对每个路径-值对的结果；以及中性元z （对应于 m）。为使定义良定义且增量化切实可行，我们进一步假设 m 是结合的、交换的，并且是线性的（即其自身的增量化）。随后我们可以使用折叠来定义树上的聚合，例如高度、大小、最大值和求和。在上述假设下，折叠是自可增量的。我们对分支因子为 2、深度最高达 20 的树进行求和求值，见附录B.4，结果表明相对于重新求值有加速。

7.4 CRDT

基于状态的无冲突复制数据类型（CRDT） [2, 31] 是一种特殊的复制数据类型，其状态满足半格定律。我们以一个经典的 CRDT 示例——由网络参与者共享的全局计数器（GCounter）——来演示增量化。 $\mathbb{I}$其状态是一个自然数向量，每个节点对应一个数值。*inc* 会递增 i位置的条目，*value* 会对所有条目求和，而*merge* 则取逐元素最大值。

图 14 展示了 DeCo 中 GCounter 的实现，它只有一种形状（$\star$），其索引集为参与者集合。增量 求值 无需重新遍历向量即可更新总和，而增量 合并 只处理稀疏变更，从而减少计算量和带宽消耗。这便得到了一种 增量状态 *CRDT* [3]：常规求值表现为基于状态的CRDT（传输完整

[8] https://www.w3.org/TR/xquery-use-cases

状态），而增量求值表现为增量状态 CRDT（仅传输变更）。传统上，增量状态 CRDT 需要分别手动实现；我们则从单个 DeCo 规范推导出这两种实现。

8 相关工作

我们讨论本方法与领域特定增量化框架、增量 λ演算，以及关于容器和数据类型泛型增量化的其他工作之间的关系。最后，我们讨论增量化与流处理之间的关系。

领域特定的增量化。 DBSP [5] 是一种用于增量流计算的语言。它们要求类型构成交换群，而我们使用更为一般的变化结构。例如，我们使用 $\mathbb{N}$，它并非群，用于我们的无冲突复制数据类型示例。

DBSP 使用因果流算子（CSO）对更新进行建模。CSO 是从流到流的函数，其中输出的第 n个元素仅依赖于输入的前 n 个元素。尽管 CSO 抽象在理论上很有吸引力，但它抽象掉了对高效增量化至关重要的实现细节，例如处理更新所需缓存的结构与大小。此外，CSO 被建模为作用于整个流（以函数表示），这使得内存需求与计算成本并不明确。

在我们的工作中，我们将其表示为一个显式管理状态的函数，采用米利机的风格。更准确地说：

$$[\![f]\!]_{\mathrm{d}} : A' \to [\![f]\!]_{\mathrm{C}} \to B' \times [\![f]\!]_{\mathrm{C}}$$

这种方法为每个程序指定一种特定的缓存类型 $[\![f]\!]_{\mathrm{C}}$，不仅明确了缓存的存在，也明确了每项运算所用缓存的具体结构。在本工作中，我们的增量化会系统地将每项运算转换为一个具体的米利机，从而确保我们的形式化与高效实现密切对应。

基于他们的理论，Budiu等人实现了一种用于增量视图维护的数据库查询语言。与我们的关系代数实例类似，他们将关系建模为到 $\mathbb{Z}$的映射。我们还将这种方法应用于其他用例，并提供一组泛型数据变换和组合子。

Differential Dataflow (DD) [22] 是一种较早的方法，与 DBSP 一样，它使用交换群对变更进行形式化。与 DBSP（以及我们的方法）不同，DD 着重于验证增量算子库，而不是对现有程序进行增量化。DD 也没有形式化说明如何在保持可靠性的同时添加新运算 [5]。

LAGO [29] 是一个用于对线性代数程序进行增量化的框架。它利用静态分析和矩阵分解，提高变化表示与计算的效率。相比之下，我们专注于提供一种机械化的增量化形式化方法，以支持范围更广的程序。尝试将他们的优化整合到我们的框架中，或许会很有趣。

增量 λ演算。一系列论文提出了增量 λ演算 [7] (IλC)，将其作为一种涉及高阶函数的增量化语言。这些论文中的形式化工作侧重于增量化 λ抽象和函数应用。具体的数据结构需要单独定义，不属于形式化内容。原始的 IλC [7]是一种简单类型 λ演算，其中每种类型都配备了一个变化结构，为每个值关联一组变更。Giarrusso 等人将 IλC 扩展为支持缓存，[14], 并采用无类型语言。最近，Matsuda 等人采用无标签终结式风格定义了一个支持缓存的 IλC 的类型化版本，但不支持一等函数。 [21], 该版本对增量函数的定义与我们类似，但没有提供证明。我们的方法和 IλC 都是增量编程语言，但它们的功能集大多互不重叠：DeCo 提供数据类型泛化操作，例如映射和重塑以及增量化组合子。相比之下，IλC 专注于

一等函数，但核心语言不提供任何内置操作。向 IλC 添加新的数据类型，需要从头实现所有运算。

缓存与高阶函数。 我们的方法采用缓存传递式，并支持良类型的增量化，但不支持高阶函数。高阶方法要么不支持缓存 [7], 要么是无类型的 Giarrusso 等人。 [14]。一个障碍在于，接受函数类型参数的增量程序无法被安全地定型，因为闭包的环境在静态上未知，因此缓存类型同样未知（正如 Matsuda 等人所指出的 [21], 他们与我们一样实现了一阶语言）。对于自动微分，已有人提出在 AD 之前应用去函数化，而不是对一等函数执行 AD [32]。类似方法对于增量化似乎也很有前景。

容器。 Wagner [33] 也研究了容器上的变更概念，但是在对称透镜的语境中。他使用容器来开发一个用于在数据类型上组合透镜的库。与我们工作的区别在于，透镜关注的是在面对变更时维护两个数据结构之间的一致性，而不是效率提升。此外，Wagner 的方法旨在手动构造透镜，而我们关注的是自动对程序进行增量化。Böhler 等人 [4] 对用于增量化的变化结构与容器进行了初步研究，但只为极少数运算提供了增量化，且这些运算均无缓存。相比之下，本文提出了一种面向缓存增量化的程序变换，以及一种采用通用核心语言与领域专用扩展的模块化方法。他们的工作也不包含增量化组合子或案例研究。

增量化的其他方法。Adapton 库 [15]通过在计算之间构建依赖图，并在叶节点处使用可变引用来实现增量化。这种方法独立于数据结构，并且能够按需惰性地计算变更。不过，它需要承担运行时构建依赖图的开销。更重要的是，这种增量化是全有或全无的：如果某个值发生变化，所有依赖于它的计算都需要重新完整计算。相反，我们旨在高效处理数据结构上的小幅变更。

Morihata [25] 介绍了一种适用于归纳数据结构的增量计算方法，并由此为函数式程序推导出优雅的增量化方案。他们将数据结构定义为由一系列构造子组成的代数数据类型，而我们将数据结构定义为容器。以代数数据类型定义的数据结构，其值是由构造子应用构成的树。因此，要访问任何值，函数式程序都必须递归遍历这棵树。我们基于容器的定义将形状与内容分离开来。这样一来，我们就能用数组存储内容，实现常数时间访问。这对于良好支持随机访问至关重要。此外，Morihata 并未提供自动程序变换。因此，目前尚不清楚如何使用这种方法，除了手动推导增量化方案之外。

流处理。 如第 4 节所述，增量化与流处理之间存在密切联系。增量化将接收完整值的“批处理”运算转换为一种运算，使其能够迭代地接收输入变化，并在每次接收时产生输出变化。这一概念对应于因果流函数（Budiu等人所用） [5]以及米利机（我们所用）。

Flo语言 [18] 形式化描述了流处理程序的正确性属性：流处理进展意味着程序不会因等待一个永不终止的流终止而阻塞；急切执行意味着无论以增量方式执行还是分批大量执行，程序都会产生确定性的输出。在我们的情形中，将增量函数定义为米利机确保了进展，而急切执行大致对应于我们的增量化正确性定理。

增量化即流式处理的观点也可以应用于分布式编程中的半格研究 [10, 17, 23, 24]：一个定义在 A 上的半格也可以解释为从一组膨胀式更新到 A的同态。假设该集合上的变化由添加一项新更新构成，那么对该同态进行增量化，就会得到一个以更新流为输入的流式程序。

9 结论

本文提出了一种用于增量化的模块化泛型方法，适用于广泛的数据结构。我们以一个名为 DeCo 的核心演算来呈现该方法。我们的演算不依赖于特定的数据结构或运算，而是支持实例化为任何可分解为形状和索引集合的数据结构。同时，我们的框架支持比选择性重新求值更细粒度的增量化；选择性重新求值只关注运算之间的依赖关系，而忽略了运算本身的增量化。

DeCo 是一种函数式编程语言，支持一组常见的泛型函数操作，例如映射 和 筛选。我们的目标是实现复杂计算的自动增量化。因此，我们的技术侧重于组合程序中出现的各项运算的导数。我们的框架通过将运算映射到其导数并组合这些导数，推导出程序的增量化结果。由此得到的增量化程序接收一个输入变化 ，并利用它来更新输出。然而，对于许多运算，输出变化不仅取决于输入变化，还取决于之前的输出本身，例如乘法。我们通过缓存来复用并更新先前结果，以解决这一问题 [14]。

请注意，我们的演算为基本运算提供了增量化。将其实例化到特定领域时，我们要求用户为额外的领域特定运算提供增量化。我们提出了一组增量化组合子，以简化这项任务。

我们的工作已在 Lean 定理证明器中实现并验证。通过使用内在类型和我们的增量化组合子，我们保证所有增量化在构造时都是正确的。我们在值保持增量化 5.6和保值组合子定理5.7中形式化了这些论断。

为了展示我们框架的通用性，我们实现了若干案例研究，并进行了相应的性能评估，结果显示相较于重新求值有显著加速。

未来工作。 我们的方法采用一阶语言。如上所述，通过去函数化在 DeCo 中支持一等函数将会很有意义。

DeCo 也可能非常适合对命令式程序进行增量化。特别是，Kumar 等人 [16] 将命令式程序建模为从输入状态到输出状态的函数。这似乎与 DeCo非常契合，因为每个程序都有形如 $A \rightsquigarrow B$ 的类型。我们可以将由多个变量组成的状态表示为嵌套元组，并通过顺序组合（;）来表达语句的顺序执行。进一步的语法扩展可以来自箭头演算 [19]，它允许以有限形式进行函数抽象和应用，并将其转换为一种组合语言，该语言与我们的语言有着惊人的相似之处。

我们当前的 DeCo 实现通过解释器实现增量执行。我们可以通过将增量化实现为源到源转换并添加编译器后端来提升性能。

最后，在本研究中，我们认为变更只会影响数据结构中的单个元素。因此，我们假设形状是静态的。Morihata 关于增量代数数据类型的 [25] 研究成果，或可使动态形状数据上的增量化成为可能。

10 数据可用性声明

我们提供了一个制品 [6]，形式为 Virtual Box 7.2 镜像，其中包含我们用 Lean 完成的证明机械化和实现。该制品说明了论文中的哪些定义和定理对应于代码中的哪些部分。它支持运行 Lean 类型检查器、验证证明检查器是否接受我们的证明，以及运行基准测试。我们还提供了编写新的容器实例和增量操作的说明。

11 致谢

我们感谢匿名审稿人提出的详尽且有帮助的反馈。

本研究工作得到了 ATHENE国家应用网络安全研究中心的支持。ATHENE由德国联邦教育与研究部和黑森州高等教育、研究与艺术部共同资助。本研究得到了 LOEWE 倡议（黑森州，德国） [LOEWE/4a//519/05/00.002(0013)/95], 、德国研究联合会（DFG，德国研究基金会）SFB 1119 项目（236615297）、黑森州高等教育、研究与艺术部通过“人工智能第三次浪潮”（3AI）集群项目的资助，以及德国研究联合会（DFG，德国研究基金会）通过德国卓越战略（EXC-3057/1“合理人工智能”，项目编号：533677015）的资助。

参考文献

[1] Michael Gordon Abbott、Thorsten Altenkirch 和 Neil Ghani。2005。容器：构造严格正类型。《理论计算机科学》 342, 1 2005 3 27 10 1016 2005 06 002 ( ), –。doi: . /J.TCS. . . Paulo Sérgio Almeida。. 无冲突复制数据类型的[2] 2025 方法。《ACM 计算调查》57, 2 (2025), 51:1–51:36。doi:10.1145/3695249[3] Paulo Sérgio Almeida、Ali Shoker 和 Carlos Baquero。2018。增量状态复制数据类型。《并行与分布式计算杂志》 111 2018 162 173 10 1016 2017 08 003 ( ), –。doi: . / J.JPDC. . . Timon Böhler、David Richter 和 Mira Mezini。[4] 2024 . 多项式函子的增量化。载于《第26届 ACM 类 Java 程序形式化技术国际研讨会论文集》，FTfJP 2024，奥地利维也纳，2024年9月20日 *2024,* 44 49 10 1145 3678721 3686231 Luca Di Stefano（编）。ACM，–。doi: . / . Mihai Budiu、Tej Chajed、Frank McSherry、Leonid Ryzhyk 和 Val Tannen。2023。DBSP：丰富查询语言的自动增量视图维护。《VLDB基金会会刊》[5] 16, 7 2023 1601 1614 10 14778 3587136 3587137 ( ), –。doi: . / . Timon Böhler、Tobias Reinhard、David Richter 和 Mira Mezini。[6] 2026 . DeCo：一种用于泛型数据类型增量函数式编程的核心演算。doi: 10 5281 18757667 . /zenodo。Yufei Cai、Paolo G. Giarrusso、Tillmann Rendel 和[7] Klaus Ostermann。2014。高阶语言的变化理论：通过静态微分对 λ演算进行增量化。载于《ACM SIGPLAN 编程语言设计与实现会议》，PLDI ’14，英国爱丁堡，2014年6月9日至11日，Michael F. P. O’Boyle 和 Keshav Pingali（编）。ACM， 145 155 10 1145 2594291 2594304 –。doi: . / . Jacques Carette、Oleg Kiselyov 和 Chung-chieh Shan。2009。终于[8] 实现无标签：用于更简单带类型语言的无标签分阶段解释器。《函数式编程杂志》 19, 5 2009 509 543 10 1017 0956796809007205 ( ), –。doi: . /S E. F. Codd。. [9] 1970 大型共享数据库的数据关系模型。《ACM 通讯》( ), –387。doi:10.1145/362384.362685[10] Neil Conway、William R. Marczak、Peter Alvaro、Joseph M. Hellerstein 和 David Maier。2012。分布式编程的逻辑与格。载于《ACM 云计算研讨会》，SOCC ’12，美国加利福尼亚州圣何塞，2012年10月14日至17日，Michael J. Carey 和 Steven Hand（编）。ACM， 1 10 1145 2391229 2391230 。doi: . / . Leonardo de Moura 和 Sebastian Ullrich。. [11] 2021 4 Lean 定理证明器与编程语言。载于《自动演绎——CADE 28 15，2021，28 12 12699——第28届自动演绎国际会议》，线上会议，2021年7月——论文集（《计算机科学讲义》，第 卷），André Platzer 和 Geoff Sutcliffe（编）。Springer，625–635。 doi:10.1007/978-3-030-79876-5 37_

[12] 2017 1, 2017 27 1 27 27 10 1145 3110271 Conal Elliott。. 编译到范畴。[13] 2025 《ACM 编程语言会刊》。ICFP ( )，: – :。doi: . / Dan R. Ghica、George Kaye 和 David Sprunger。. 时序数字电路的完备理论：指称、操作与代数语义。arXiv: 2201 10456 [ ] 2201 10456 . cs.LO [14] https://arxiv.org/abs/ 。Paolo G. Giarrusso、Yann Régis-Gianas 和 Philipp Schuster。2019 . 缓存传递风格的增量 λ 演算——通过程序变换实现静态记忆化。载于《编程语言与系统——第28届欧洲编程研讨会 λ

编程，ESOP 2019，作为欧洲软件理论与实践联合会议 ETAPS 2019 的一部分举行，捷克共和国布拉格，2019年4月6日至11日，论文集（《计算机科学讲义》，第11423卷），Luís Caires（编）。Springer，553 580 10 1007 978 3 030 17184 1_20 –。doi: . / - - - - Matthew A. Hammer、Yit Phang Khoo、[15] Michael Hicks 和 Jeffrey S. Foster。2014。Adapton：可组合的按需增量计算。载于《ACM SIGPLAN 编程语言设计与实现会议》，PLDI '14，英国爱丁堡，2014年6月9日至11日，Michael F. P. O'Boyle 和 Keshav Pingali（编）。ACM，156–166。doi:10.1145/2594291.2594324[16] Prashant Kumar、André Pacak 和 Sebastian Erdweg。2025。通过微分执行实现增量计算。载于第39届欧洲面向对象编程会议，ECOOP 2025，2025年6月30日至7月2日，挪威卑尔根（LIPIcs，第333卷），Jonathan Aldrich 和 Alexandra Silva（编）。Schloss Dagstuhl - Leibniz-Zentrum für Informatik，20:1–20:24。doi:10.4230/LIPICS.ECOOP.2025.20[17] Lindsey Kuper 和 Ryan R. Newton。2013。LVars：用于确定性并行的基于格的数据结构。载于《第2届 ACM SIGPLAN 函数式高性能计算研讨会论文集》（美国马萨诸塞州波士顿）（FHPC '13 71 84 10 1145 2502323 2502326 ）。计算机协[18] 2025 会，美国纽约州纽约市，–。doi: . / . Shadaj Laddad、Alvin Cheung、Joseph M. Hellerstein 和 Mae Milano。. Flo：渐进式流处理的语义基础。《ACM 编程语言会刊》9, 2025 241 270 10 1145 3704845 POPL ( )，–。doi: . / Sam[19] 2010 Lindley、Philip Wadler 和 Jeremy Yallop。. 箭头演算。《函数式编程杂志》20, 1 (2010), 51–69。doi:10.1017/S095679680999027X[20] Yanhong A. Liu。2000。通过增量化提高效率：简介。《高阶与符号计算》13, 4 (2000), 289–313。doi:10.1023/A:1026547031739[21] Kazutaka Matsuda、Samantha Frohlich、Meng Wang 和 Nicolas Wu。2023。通过反嵌入实现嵌入。《ACM 编程语言会刊》7, 2023 1 47 10 1145 3607830 ICFP ( )，–。doi: . / Frank McSherry、Derek Gordon Murray、[22] Rebecca Isaacs 和 Michael Isard。2013。Differential Dataflow。载于第六届创新数据系统研究双年会，CIDR 2013，美国加利福尼亚州阿西洛马，2013年1月6日至9日，在线论文集。www.cidrdb.org。http://cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf [23] Christopher Meiklejohn 和 Peter Van Roy。2015。Lasp：一种用于分布式无协调编程的语言。载于《第17届声明式编程原理与实践国际研讨会论文集》，意大利锡耶纳，2015年7月14日至16日，Moreno Falaschi 和 Elvira Albert（编）。ACM，184–195。doi:10.1145/2790449.2790525[24] Mae Milano、Rolph Recto、Tom Magrino 和 Andrew C. Myers。2019。Gallifrey 导览：一种用于地理分布式编程的语言。载于2019年3月17日，2019，16 136rd 编程语言进展峰会，SNAPL，美国罗德岛州普罗维登斯，5月——（LIPIcs，第 卷），Benjamin S. Lerner、Rastislav Bodík 和 Shriram Krishnamurthi（编）。Schloss Dagstuhl - Leibniz-Zentrum für Informatik，11:1–11:19。doi:10.4230/LIPICS.SNAPL.2019.11[25] Akimasa Morihata。2018。使用数据结构进行增量计算。《科学计算编程》164 (2018), 18–36。doi:10.1016/J.SCICO.2017.04.001[26] Chris Okasaki。1999。纯函数式数据结构。剑桥大学出版社。[27] Marco Patrignani。2020。为什么要使用颜色？或者，超越代码片段的语法高亮。arXiv abs/2001.11334 (2020)。[28] Alex Reinking、Ningning Xie、Leonardo de Moura 和 Daan Leijen。2021。Perceus：通过复用实现无垃圾回收的引用计数。载于《第42届 ACM SIGPLAN 编程语言设计与实现国际会议论文集》（线上会议，加拿大）（PLDI 2021）。计算机协会，美国纽约州纽约市，96–111。doi:10.1145/3453483.3454032[29] Amir Shaikhha、Mohammed Elseidy、Stephan Mihaila、Daniel Espino 和 Christoph Koch。2020。增量线性代数程序合成。《ACM 数据库系统汇刊》45, 3 2020 12 1 12 44 10 1145 3385398 ( )，：– :。doi: . / [30] 2022 Amir Shaikhha、Mathieu Huot、Jaclyn Smith 和 Dan Olteanu。. 使用半环字典进行函数式集合编程。《ACM 编程语言会刊》6, 1 2022 1 33 10 1145 3527333 OOPSLA ( )，–。doi: . / [31] 2011 Marc Shapiro、Nuno M. Preguiça、Carlos Baquero 和 Marek Zawirski。. 无冲突复制数据类型。载于《分布式系统的稳定性、安全性与安全保障——第13届国际研讨会，SSS 2011，法国格勒诺布尔，2011年10月10日至12日。论文集》（《计算机科学讲义》，第6976卷），Xavier Défago、Franck Petit 和 Vincent Villain（编）。Springer，386 400 10 1007 978 3 642 24550 3_29 –。doi: . / - - - - Tom Smeding 和 Matthijs Vákár。. 高效[32] 2024 CHAD。《 ACM 编程语言会刊》 8, POPL (2024), 1060–1088。doi:10.1145/3632878[33] Daniel Wagner。2014。对称编辑透镜：双向语言的新基础。宾夕法尼亚大学。

$$
\begin{aligned}
&[\![\mathrm{F}_s\ A]\!] := \quad [\![s]\!]_{\mathcal{I}} \to [\![A]\!] [\![\mathrm{F}_s\ A]\!]' := \quad [\![s]\!]_{\mathcal{I}} \to [\![A]\!]' \\
&f \oplus g := \quad \lambda i.\ f\ i \oplus g\ i f \ominus g := \quad \lambda i.\ f\ i \ominus g\ i [\![A \times B]\!] : \\
&= \quad [\![A]\!] \times [\![B]\!] [\![A \times B]\!]' := \quad [\![A]\!]' \times [\![B]\!]' (x, y)\ \oplus (x', y') := (\\
&x \oplus x', y \oplus y')\ (x, y)\ \ominus (x', y') := (x \ominus x', y \ominus y') [\![A + B]\!] : \\
&= \quad [\![A]\!] + [\![B]\!] [\![A + B]\!]' := \mathrm{cl}\ [\![A]\!]' \mid \mathrm{cr}\ [\![B]\!]' \mid \mathrm{sl}\ [\![A]\!] \mid \mathrm{sl} \\
&[\![B]\!] \mid \text{空值} \iota_1\ x \oplus \mathrm{cl}\ x' := \quad \iota_1\ (x \oplus x') \iota_1\ x \oplus \mathrm{cr}\ y' := \quad \iota_1\ x \\
&\iota_2\ y \oplus \mathrm{cl}\ x' := \quad \iota_2\ y \iota_2\ y \oplus \mathrm{cr}\ y' := \quad \iota_2\ (y \oplus y') x \oplus \mathrm{sl}\ y := \quad \iota_1\ x \\
&x \oplus \mathrm{sr}\ y := \quad \iota_2\ x x \oplus \text{空值} := \quad x \iota_1\ x \ominus \iota_1\ y := \mathrm{cl}\ (x \ominus y) \\
&\iota_1\ x \ominus \iota_2\ y := \mathrm{sl}\ x \iota_2\ x \ominus \iota_1\ y := \mathrm{sr}\ x \iota_2\ x \ominus \iota_2\ y := \mathrm{cr}\ (x \ominus y)
\end{aligned}
$$

图 15. 变化结构。

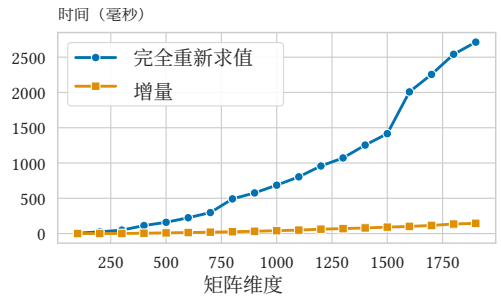
完整清单

在图 15中，我们定义了容器、积和和的变化结构。容器和积的变化结构逐元素应用其运算；和的情况则更为复杂，其中包含一种自定义的和变化类型。局部变化 $\mathrm{cl}\ x'$ 将一个值更新 $\iota_1\ x$为 $\iota_1\ (x \oplus x')$，否则保持不变（cr 对 ι_2执行相同操作）。变化 $\mathrm{sl}\ x$则用新值替换旧值 x。这种非局部变化用于确保完备性，即能够将任意值变为任意其他值：局部变化（cl、cr）不允许我们从$\iota_1\ x$变为 $\iota_2\ y$，因为它们会保持构造子不变。我们还添加了一种空值 变化，用于对 ||进行增量化。

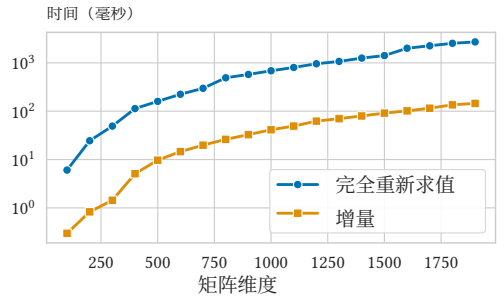
求和运算的增量化见图16。

$$[\![\mathsf{fuse}_A]\!]_{\mathsf{C}} := \quad [\![A]\!] + [\![A]\!][\![\mathsf{fuse}]\!]_{\mathsf{i}}\ (\iota_1\ x) := (x, \iota_1\ x)[\![\mathsf{fuse}]\!]_{\mathsf{i}}\ (\iota_2\ x) := (x, \iota_2\ x)[\![\mathsf{fuse}]\!]_{\mathsf{d}}\ (\mathsf{cl}\ \ x')\ (\iota_1\ x) := (x', \iota\ (x \oplus x'))[\![\mathsf{fuse}]\!]_{\mathsf{d}}\ (\mathsf{cr}\ \ y')\ (\iota_1\ x) := (x \ominus x, \iota_1\ x)[\![\mathsf{fuse}]\!]_{\mathsf{d}}\ (\mathsf{cl}\ \ x')\ (\iota_2\ y) := (y \ominus y, \iota_2\ y)[\![\mathsf{fuse}]\!]_{\mathsf{d}}\ (\mathsf{cr}\ \ y')\ (\iota_2\ y) := (y', \iota_2\ (y \oplus y'))[\![\mathsf{fuse}]\!]_{\mathsf{d}}\ (\mathsf{sl}\ \ x)\ (\iota_1\ x_0) := (x \ominus x_0, \iota_1\ x)[\![\mathsf{fuse}]\!]_{\mathsf{d}}\ (\mathsf{sl}\ \ x)\ (\iota_2\ y_0) := (x \ominus y_0, \iota_1\ x)[\![\mathsf{fuse}]\!]_{\mathsf{d}}\ (\mathsf{sr}\ \ y)\ (\iota_1\ x_0) := (y \ominus x_0, \iota_2\ y)[\![\mathsf{fuse}]\!]_{\mathsf{d}}\ (\mathsf{sr}\ \ y)\ (\iota_2\ y_0) := (y \ominus y_0, \iota_2\ y)[\![\mathsf{fuse}]\!]_{\mathsf{d}}\ \text{空值}\ (\iota_1\ x) := (x \ominus x, \iota_1\ x)[\![\mathsf{fuse}]\!]_{\mathsf{d}}\ \text{空值}\ (\iota_2\ y) := (y \ominus y, \iota_2\ y)[\![\mathsf{distr}_{A,B,C}]\!]_{\mathsf{C}} := \quad [\![A]\!] \times ([\![B]\!] + [\![C]\!])[\![\mathsf{distr}]\!]_{\mathsf{i}}\ (x, \iota_1\ y) := (\iota_1\ (x, y), (x, \iota_1\ y))[\![\mathsf{distr}]\!]_{\mathsf{i}}\ (x, \iota_2\ y) := (\iota_2\ (x, y), (x, \iota_2\ y))[\![\mathsf{distr}]\!]_{\mathsf{d}}\ (x', \mathsf{cl}\ \ y')\ (x, \iota_1\ y) := (\mathsf{cl}\ (x', y'), (x \oplus x', \iota_1\ (y \oplus y')))[\![\mathsf{distr}]\!]_{\mathsf{d}}\ (x', \mathsf{cr}\ \ y')\ (x, \iota_1\ y) := (\mathsf{cl}\ (x', y \ominus y), (x \oplus x', \iota_1\ y))[\![\mathsf{distr}]\!]_{\mathsf{d}}\ (x', \mathsf{cl}\ \ y')\ (x, \iota_2\ y) := (\mathsf{cr}\ (x', y \ominus y), (x \oplus x', \iota_2\ y))[\![\mathsf{distr}]\!]_{\mathsf{d}}\ (x', \mathsf{cr}\ \ y')\ (x, \iota_2\ y) := (\mathsf{cr}\ (x', y'), (x \oplus x', \iota_2\ (y \oplus y')))[\![\mathsf{distr}]\!]_{\mathsf{d}}\ (x', \mathsf{sl}\ \ y)\ (x, _) := (\mathsf{sl}\ (x \oplus x', y), (x \oplus x', \iota_1\ y))[\![\mathsf{distr}]\!]_{\mathsf{d}}\ (x', \mathsf{sr}\ \ y)\ (x, _) := (\mathsf{sr}\ (x \oplus x', y), (x \oplus x', \iota_2\ y))[\![\mathsf{distr}]\!]_{\mathsf{d}}\ (x', \text{空值})\ (x, \iota_1\ y) := (\mathsf{cl}\ (x', y \ominus y), (x \oplus x', \iota_1\ y))[\![\mathsf{distr}]\!]_{\mathsf{d}}\ (x', \text{空值})\ (x, \iota_2\ y) := (\mathsf{cr}\ (x', y \ominus y), (x \oplus x', \iota_2\ y))[\![f \mid\mid_{A_1,A_2,B_1,B_2} g]\!]_{\mathsf{C}} := \quad [\![f]\!]_{\mathsf{C}} \times [\![B_1]\!] + [\![g]\!]_{\mathsf{C}} \times [\![B_2]\!][\![f \mid\mid g]\!]_{\mathsf{i}}\ (\iota_1\ x) := (\iota_1\ y, \iota_1\ (c, y))\ \text{其中}\ (y, c) := f_{i\,\mathsf{i}}\ x[\![f \mid\mid g]\!]_{\mathsf{i}}\ (\iota_2\ x) := (\iota_2\ y, \iota_2\ (c, y))\ \text{其中}\ (y, c) := g_{i\,\mathsf{i}}\ x[\![f \mid\mid g]\!]_{\mathsf{d}}\ (\mathsf{cl}\ \ x')\ (\iota_1\ (c, y)) := (\mathsf{cl}\ \ y', \iota_1\ (c', y \oplus y'))\ \text{其中}\ (y', c') := f_{i\,\mathsf{d}}\ x'\ c[\![f \mid\mid g]\!]_{\mathsf{d}}\ (\mathsf{cr}\ \ x')\ (\iota_1\ (c, y)) := (\text{空值}\ \iota_1\ (c, y))[\![f \mid\mid g]\!]_{\mathsf{d}}\ (\mathsf{cl}\ \ x')\ (\iota_2\ (c, y)) := (\text{空值}\ \iota_2\ (c, y))[\![f \mid\mid g]\!]_{\mathsf{d}}\ (\mathsf{cr}\ \ x')\ (\iota_2\ (c, y)) := (\mathsf{cr}\ \ y', \iota_2\ (c', y \oplus y'))\ \text{其中}\ (y', c') := g_{i\,\mathsf{d}}\ x'\ c[\![f \mid\mid g]\!]_{\mathsf{d}}\ (\mathsf{sl}\ \ x)\ (\iota_1\ (c, y_0)) := (\mathsf{cl}\ (y \ominus y_0)\ \ \iota_1\ (c, y))\ \text{其中}\ (y, c) := f_{i\,\mathsf{i}}\ x[\![f \mid\mid g]\!]_{\mathsf{d}}\ (\mathsf{sl}\ \ x)\ (\iota_2\ _) := (\mathsf{sl}\ \ y, \iota_1\ (c, y))\ \text{其中}\ (y, c) := f_{i\,\mathsf{i}}\ x[\![f \mid\mid g]\!]_{\mathsf{d}}\ (\mathsf{sr}\ \ x)\ (\iota_1\ _) := (\mathsf{sr}\ \ y, \iota_2\ (c, y))\ \text{其中}\ (y, c) := g_{i\,\mathsf{i}}\ x[\![f \mid\mid g]\!]_{\mathsf{d}}\ (\mathsf{sr}\ \ x)\ (\iota_2\ (c, y_0)) := (\mathsf{cr}\ (y \ominus y_0)\ \ \iota_2\ (c, y))\ \text{其中}\ (y, c) := g_{i\,\mathsf{i}}\ x[\![f \mid\mid g]\!]_{\mathsf{d}}\ \text{空值}\ \ x := (\text{空值}\ \ x)$$

图 16. 求和算子的增量化。

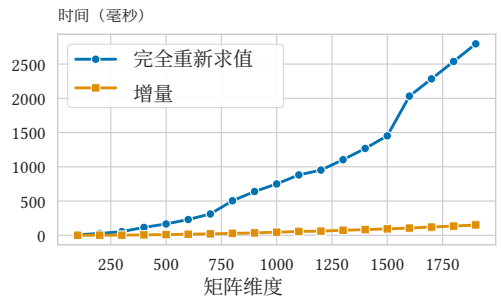


(a) 执行时间与矩阵维度的关系（线性刻度）。

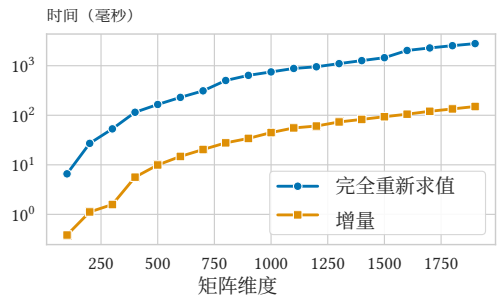


(b) 执行时间与矩阵维度的关系（对数刻度）。

图 17. 矩阵-向量乘法基准测试执行时间与矩阵维度的关系。



(a) 执行时间与矩阵维度的关系（线性刻度）。



(b) 执行时间与矩阵维度的关系（对数刻度）。

图 18. 全连接层（带偏置和 ReLU 激活的矩阵-向量乘法）基准测试执行时间与矩阵维度的关系。

B 评估详情

我们展示了案例研究的库实现及求值结果。所有时间测量均取 5 次测试中的最佳值。基准测试所用的机器配备 AMD Ryzen 7 PRO 7840U 中央处理器，拥有 16 个核心和 32 个线程，运行 Debian 13 trixie，内核版本为 6.12.63。Lean 版本为 4.22.0-rc4。

B.1 线性代数

在线性代数求值中，我们对矩阵-向量乘法和全连接层（带偏置和 ReLU 激活的矩阵-向量乘法）进行基准测试。我们对输入向量中 1% 的元素施加变更。结果绘制于图18和图 17。

为了解输入的稀疏度如何影响性能，我们还对稀疏度（非零元素的百分比）各不相同的情况下的矩阵-向量乘法进行基准测试，所用矩阵固定1000×1000。结果绘制于图 19。

B.2 关系代数

我们评估两种关系代数运算：一种聚合操作（投影），以及一种非聚合操作（连接，然后筛选）。

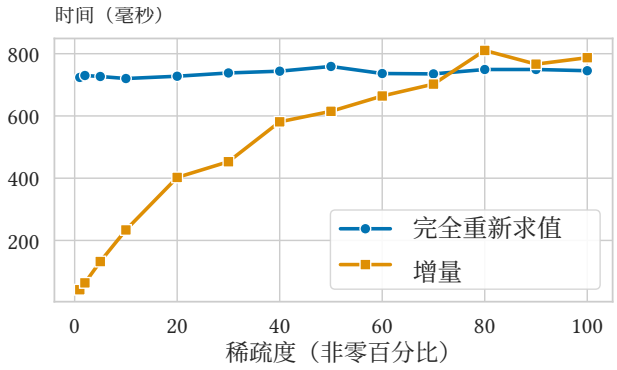


图 19. 矩阵-向量乘法的基准测试执行时间与输入稀疏度（非零元素的百分比）的关系，矩阵维度固定为 1000×1000。

```
def countD (m : Z α) : Int := m.折叠 (λ c _ v => c + v) 0 def count : Incr (Z α) Int := .self countD
countD def 分组 (f : α → β) (m : Z α) : D β (Z α) :=
Id.run do let mut res := ∅for (k, v) in m do res :=
res.alter (f k) (λ| .some m => m.Addk v| .none =>
some {(k, v)}) return res def 分组 (f : α → β) : Incr
(Z α) (D β (Z α)) :=Incr.self (groupByD f)
(groupByD f) def 投影 (f : α → β) : Incr (D α Int) (
D β Int) :=分组 f ;; Incr.映射count
```

图 20. 计数、分组和投影的实现。

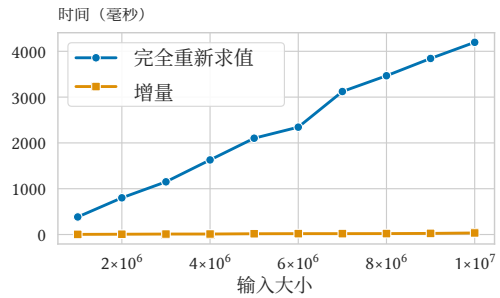
聚合。在图 20 中，我们展示了关系代数聚合案例研究的库实现（注意，Z 是关系类型构造子）。对于 计数 和 分组，我们在 Lean 中编写直接实现，并使用 Incr.self 组合子将其增量化。然后，我们定义一个派生操作 投影，它根据输入函数 f对输入关系进行投影。

结果绘制于图 21中。变化量为输入大小的 1%。

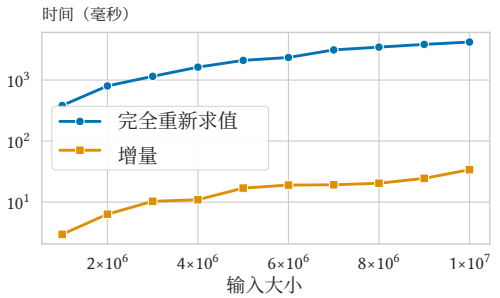
连接。对于非聚合运算，我们对连接进行基准测试 $R \bowtie_p S$： = ⊗; 筛选 p。结果绘制于图 22中。变化量为输入大小的 1%。

B.3 JSON

在图23 中，我们展示了 JSON 案例研究中使用的示例数据；在图24中，我们展示了 Q1 查询的库实现。程序首先移除所有非 Addison-Wesley 出版的书籍，然后移除所有出版年份不晚于 1991 年的书籍，最后将结果投影到

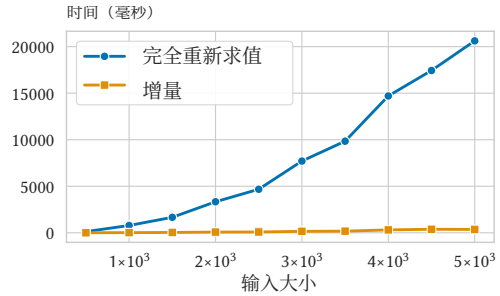


（a）执行时间与输入大小的关系（线性刻度）。

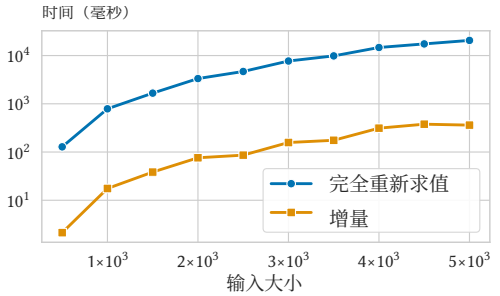


（b）执行时间与输入大小的关系（对数刻度）。

图 21. 关系代数聚合（投影）基准测试执行时间与输入大小的关系。



（a）执行时间与输入大小的关系（线性刻度）。



（b）执行时间与输入大小的关系（对数刻度）。

图 22. 关系代数连接基准测试执行时间与输入大小的关系。

的年份和标题字段。辅助函数checkPath 接收一个路径和一个谓词，并返回一个增量函数，用于删除在给定路径处不满足该谓词的 JSON 值。

B.4 玫瑰树

对于玫瑰树评估，我们生成分支因子为 2 的树，并定义树上的增量求和。变化量为树大小的 1 %。结果绘制在图25中。

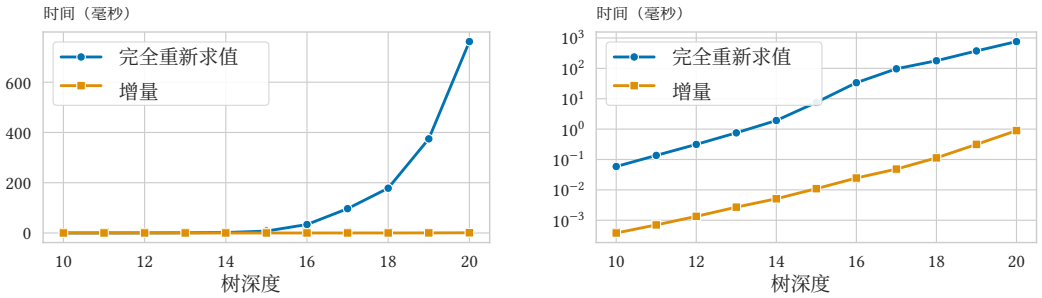
收稿日期：2025-10-10；录用日期：2026-02-17；修订日期：2007年2月20日；修订日期：2009年3月12日；录用日期：2009年6月5日

```
结构 人物 其中名 : 字符串 姓 :
字符串 所属机构 : 可选类型 字
符串 : = 无

结构 书籍其中
 年份 : 自然数 标题 : 字
 符串 作者 : 列表 人物:
 = []编辑 : 列表 人物:= []
 出版商 : 字符串 价格 :
 浮点数

定义 书目 : 数组 书籍 := #[{ 年份 := 1994, 标题 := "
TCP/IP Illustrated", 作者 := [{ 名 := "W.",姓:= "Stevens" }
],出版商 := "Addison-Wesley", 价格 := 65.95 }, { 年份 :
= 1992, 标题 := "Advanced Programming in the
Unixenvironment", 作者 := [{ 名 := "W.",姓:= "Steve
ns" }],出版商 := "Addison-Wesley", 价格 := 65.95 }, { 年
份 := 2000, 标题 :="Data on the Web", 作者 := [{ 名 := "
Serge", 姓 := "Abiteboul"}, { 名 := "Peter", 姓 := "
Buneman"}, { 名 := "Dan", 姓 := "Suciu" }],出版商 := "
Morgan Kaufmann Publishers", 价格 := 39.95 }, { 年份 :
= 1999, 标题 := "The Economics of
TechnologyandContentforDigi
                                                    tal TV",
  编辑者 : = [{ first : = "Darcy",last: = "Gerbarg",affiliation : = " CITI" }],
    出版商: ="Kluwer Academic
Publishers", 价格: = 129.95 }]
```

图23。来自 https://www.w3.org/TR/xquery-use-cases/ 的 Q1 测试数据。Lean 可以自动推导JSON序列化和反序列化；我们进一步将 JSON 转换为我们的容器表示（此处均省略）。

```
定义 checkPath (p : JsonPath) (q : JSON 标量 → Bool) : Incr
Json Json： =.triv λ v => 如果 q (v.获取 p) 则 v 否则 ∅

def q1I : Incr (D 自然数 JSON) (D 自然数 JSON) :=let f1 :
Incr (D 自然数 JSON) (D 自然数 JSON) :=.map (checkPath [.
dict "publisher"] λ v =>v ==(.str "Addison-Wesley")) let
f2 : Incr (D 自然数 JSON) (D 自然数 JSON) :=.map
(checkPath [.dict "year"] λ| .num n => n > 1991 _ => false)
f1 ;; f2 ;; .map (.filter λ k => k == [.dict "year"] || k == [.
dict "title"])
```

Fig. 24. Q1 的增量实现。“D”是一种字典类型，“Json”是 JSON 容器 r.



(a) 执行时间与树深度（线性刻度）。 (b) 执行时间与树深度（对数刻度）。

图 25. 玫瑰树基准测试执行时间与树深度。